
NeuroML Documentation

NeuroML contributors

Aug 02, 2022

CONTENTS

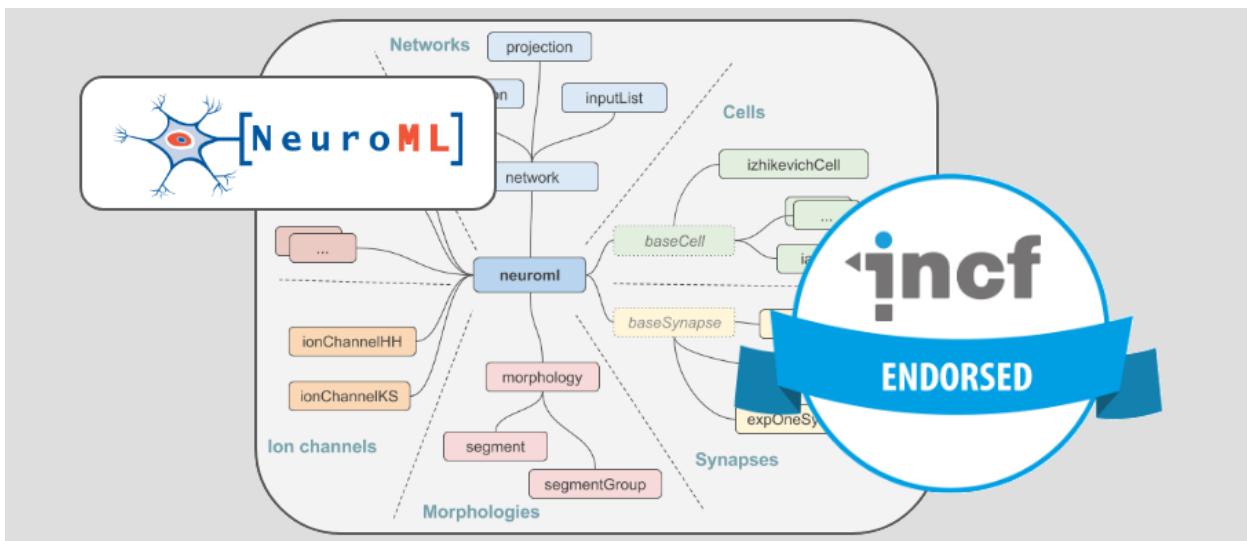
I User documentation	5
1 Mission and Aims	7
2 How to use this documentation	9
2.1 Structure and navigation	9
2.2 Using Jupyter notebooks included in the documentation	9
2.3 Reporting bugs and issues	12
3 Getting started with NeuroML	13
3.1 Simulating a regular spiking Izhikevich neuron	13
3.2 Interactive single Izhikevich neuron NeuroML example	22
3.3 A two population network of regular spiking Izhikevich neurons	25
3.4 Interactive two population network example	35
3.5 Simulating a single compartment Hodgkin-Huxley neuron	41
3.6 Interactive single compartment HH example	62
3.7 Simulating a multi compartment OLM neuron	70
3.8 Interactive multi-compartment OLM cell example	102
4 Finding and sharing NeuroML models	115
4.1 NeuroML-DB: The NeuroML Database	115
4.2 Open Source Brain	116
4.3 Other related projects	116
4.4 NeuroMorpho.Org	116
4.5 OpenWorm	117
4.6 Allen Institute	117
4.7 Blue Brain Project	117
5 Creating NeuroML models	119
5.1 Converting cell models to NeuroML and sharing them on Open Source Brain	119
6 Validating NeuroML Models	125
6.1 Using the command line tools	125
6.2 Using the Python API	125
7 Visualising NeuroML Models	127
7.1 Get a quick summary of your model	127
7.2 View the 3D structure of your model	128
7.3 View the connectivity graph of your model	131
7.4 View the connectivity matrices of the model	133
7.5 View graph of the simulation instance of the model	135
7.6 Viewing/analysing ion channel dynamics	137

7.7	Visualising and analysing ion channel models	137
8	Simulating NeuroML Models	143
8.1	Using Open Source Brain	143
8.2	Using jNeuroML/pyNeuroML	144
8.3	Using NEURON	145
8.4	Using NetPyNE	145
8.5	Using Brian2	146
8.6	Using MOOSE	146
8.7	Using EDEN	146
8.8	Using Arbor	146
9	Optimising/fitting NeuroML Models	147
9.1	Loading data and calculating metrics to use for optimisation	157
9.2	Running the optimisation	169
9.3	Viewing results	179
10	Schema/Specification	187
10.1	NeuroML v2	188
10.2	NeuroML v1	455
11	Using NeuroML 2 and LEMS	457
11.1	Conventions	457
11.2	Units and dimensions	459
11.3	Paths	459
11.4	Quantities and recording	463
11.5	LEMS Simulation files	463
11.6	Extending NeuroML with LEMS	466
12	Software and Tools	473
12.1	Core NeuroML Tools	473
12.2	Other NeuroML supporting applications	474
12.3	pyNeuroML	474
12.4	libNeuroML	479
12.5	pyLEMS	482
12.6	NeuroMLlite	484
12.7	jNeuroML	487
12.8	jLEMS	490
12.9	NeuroML C++ API	491
12.10	MatLab NeuroML Toolbox	491
12.11	Tools and resources with NeuroML support	492
13	Citing NeuroML and related publications	509
13.1	Citing NeuroML	509
13.2	Other publications	512
14	Frequently asked questions (FAQ)	515
14.1	Are length 0 segments allowed in NeuroML?	515
II	NeuroML events	517
15	June 2022: NeuroML tutorial at CNS*2022 satellite tutorials	519
15.1	Times and dates	519
15.2	Target audience	519

15.3 Where	519
15.4 Agenda	519
16 April 2022: NeuroML development workshop at HARMONY 2022	521
16.1 Agenda	521
16.2 Times and dates	521
16.3 Registration	521
16.4 Open an issue beforehand!	522
16.5 Slack	522
17 October 2021: NeuroML development workshop at COMBINE meeting	523
17.1 Times and dates	523
17.2 Target audience	523
17.3 Agenda/minutes	523
18 August 2021: NeuroML tutorial at INCF Training Weeks	525
18.1 Times and dates	525
18.2 Target audience	525
18.3 Agenda	525
19 July 2021: NeuroML tutorial at CNS*2021	527
19.1 Why take part?	527
19.2 Times and dates	527
19.3 Registration	527
19.4 Pre-requisites	527
19.5 Slack	528
20 March 2021: NeuroML hackathon at HARMONY 2021	529
20.1 Why take part?	529
20.2 Times and dates	529
20.3 Registration	530
20.4 Open an issue beforehand!	530
20.5 Slack	530
21 Past NeuroML Events	531
III The NeuroML Initiative	533
22 Getting in touch	535
22.1 Mailing list	535
22.2 Chat channels	535
22.3 Issues related to the libraries or specification	535
22.4 Social media	535
23 Overview of standards in neuroscience	537
23.1 NeuroML as a standard	537
24 A brief history of NeuroML	539
24.1 The early days	539
24.2 NeuroML v1.x	539
24.3 NeuroML v2.x - introducing LEMS	540
24.4 The future	540
25 NeuroML Editorial Board	541
25.1 Current Editorial Board	541

25.2	Procedures	543
25.3	Responsibilities of NeuroML Editors	543
25.4	History of the NeuroML Editorial Board	543
25.5	Workshop and Meeting reports	545
26	NeuroML Scientific Committee	547
26.1	Current Members	547
26.2	Past Members	550
27	Funding and Acknowledgements	551
28	NeuroML contributors	553
28.1	Repositories	554
29	Code of Conduct	557
IV	Developer documentation	559
30	Overview	561
30.1	Contribution guidelines	561
30.2	Release Process	564
30.3	Making changes to the NeuroML standard	565
31	Interaction with other languages and standards	567
31.1	PyNN	567
31.2	SBML	567
31.3	Sonata	567
31.4	NineML & SpineML	567
31.5	ModECI MDF	567
31.6	SWC	568
V	Reference	569
32	Glossary	571
33	Bibliography	573
	Bibliography	575

A model description language for computational neuroscience.



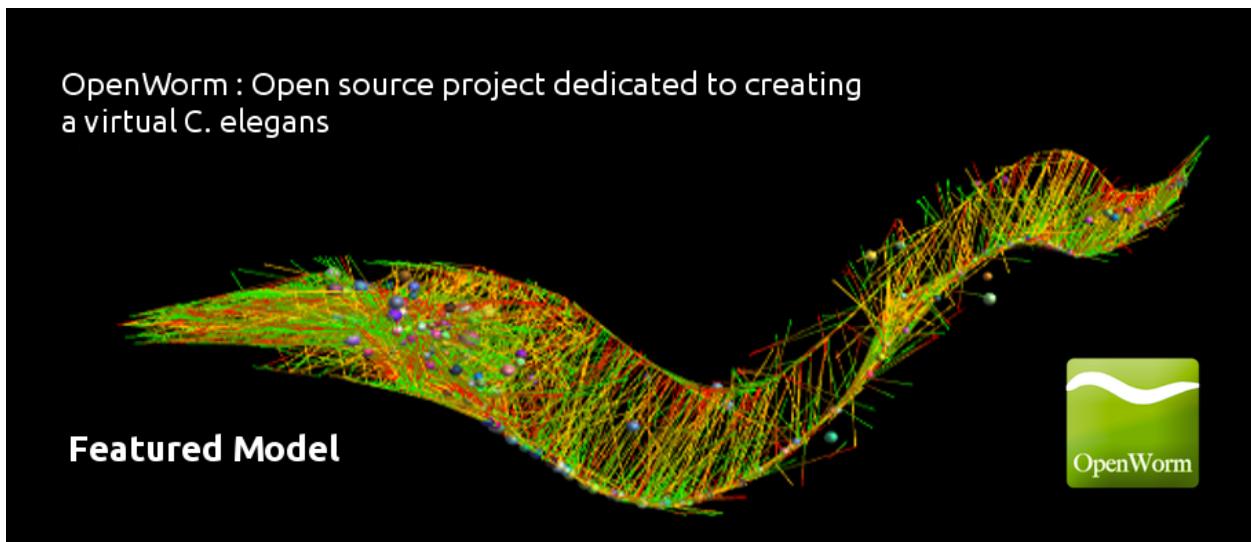
<p>Core Standards</p> <p>the  mbine computational modeling in biology network</p>	<p>Standards for Knowledge</p> <p>Projects</p> <p>Associated Standards</p> <p>Used by core standards</p>	<p>Standards for Visual</p> <p>BIOMODELS.NET</p> <p>qualifiers</p>	<p>Standards for Models and their Analyses</p> <p>S3ML  [NeuroML] </p>
---	--	---	--

Modelling the brain, together



[Explore OSB](#)

Open Source Brain is a resource for sharing and collaboratively developing computational models of neural systems.



NeuroML is an *international, collaborative initiative* to develop a language for describing detailed models of neural systems, which will serve as a standard data format for defining and exchanging descriptions of neuronal cell and network models. NeuroML is:

- modular
- standardised
- structured

and this allows you to:

- easily *build* and *optimise* detailed models of neural systems
- easily *validate* your models
- easily *visualise* your models
- easily *simulate* your models using a variety of simulators
- easily *analyse* your simulations

all using a *well supported set of tools* in the powerful `Python` programming language.

In this documentation, you will find information on [*using NeuroML*](#), [*developing with NeuroML*](#), its [*specification*](#), and the [*community*](#) that maintains it.

For any queries, please contact the NeuroML community using any of our [*communication channels*](#).

Part I

User documentation

**CHAPTER
ONE**

MISSION AND AIMS

Computational models, based on detailed neuroanatomical and electrophysiological data, are heavily used as an aid for understanding the nervous system. NeuroML is an international, collaborative initiative to develop a language for describing detailed models of neural systems, which will serve as a standard data format for defining and exchanging descriptions of neuronal cell and network models.

NeuroML specifications are developed by the *NeuroML Editorial Board* and overseen by its *Scientific Committee*. NeuroML is endorsed by the INCF, and is also an official COMBINE standard.

The NeuroML project community develops an [XML \(eXtensible Markup Language\)](#) based description language where [XML Schemas](#) are used to define model specifications. The community also develops and maintains a number of libraries (in *Python, Java and other languages*) to facilitate use of these specifications.

The **aims of the NeuroML initiative** are:

- To create specifications for an XML-based language that describes the biophysics, anatomy and network architecture of neuronal systems at multiple scales
- To facilitate the exchange of complex neuronal models between researchers, allowing for greater transparency and accessibility of models
- To promote software tools which support NeuroML and support the development of new software and databases for neural modeling
- To encourage researchers with models within the scope of NeuroML to exchange and publish their models in this format

HOW TO USE THIS DOCUMENTATION

This documentation is generated using [Jupyter books](#). You can learn more about the project on their website.

2.1 Structure and navigation

Close the left hand side bar by clicking the left facing arrow in the top panel.

You can close the left hand side bar by clicking the left facing arrow in the top panel. This increases the width of the middle section of the documentation and can be helpful on smaller screens. You can click the hamburger menu that replaces the left facing arrow to open the left hand side bar when required.

The documentation is divided into a few parts that can be seen in the *left hand side navigation bar*:

- **User documentation:** this includes documentation for anyone looking to use NeuroML
- **NeuroML events:** any events related to NeuroML will be listed here
- **The NeuroML Initiative:** this includes documentation on the NeuroML community
- **Developer documentation:** this includes information for individuals looking to contribute to NeuroML (either the standard or the software)
- **Reference:** this includes the glossary of terms and the bibliography.

Each part contains different chapters, which can each contain different sections. Each page in the documentation also has its own navigation in the *right hand side bar*.

2.2 Using Jupyter notebooks included in the documentation

Familiar with Jupyter Notebooks? Skip ahead to the next section.

If you are familiar with Jupyter Notebooks, you can skip ahead to the [Getting started with NeuroML](#) section.

The most important feature of Jupyter books is that it allows you to include [Jupyter notebooks](#) in the documentation. This allows us to write documentation which includes code examples that can be modified and executed by users interactively in their browsers *without having to install anything on their local machines*. For example, these are used in the [Getting Started](#) section.

Each Jupyter notebook in the documentation includes a rocket icon  in the top bar:

Clicking this icon will allow you to run the Jupyter Notebook:



Interactive single Izhikevich neuron NeuroML example

Fig. 2.1: Click the rocket icon in top panel of executable pages to execute them in Binder or Google Collaboratory.

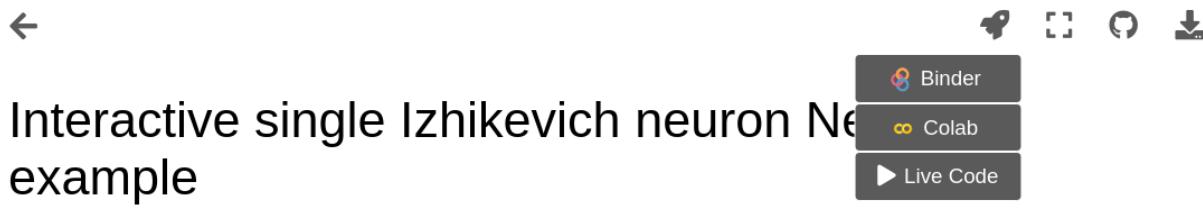


Fig. 2.2: You can run the Jupyter Notebook on Binder or Google Colaboratory.

You can choose from freely available services such as [Binder](#) and [Google Colaboratory](#). Both Binder and Google Colaboratory will take you to these services and load the Jupyter Notebook for you to use. The Live code option uses Binder but allows you to run the code in the current tab itself. However, please note that this option does not include the full Jupyter Notebook features that Binder and Google Colaboratory provide.

Run Binder and Google Colaboratory in a new tab.

It is suggested to right click and select “Open in new tab” so that the tab with the NeuroML documentation remains open. In most browsers, you can also use `Ctrl + click` to open links in a new tab:

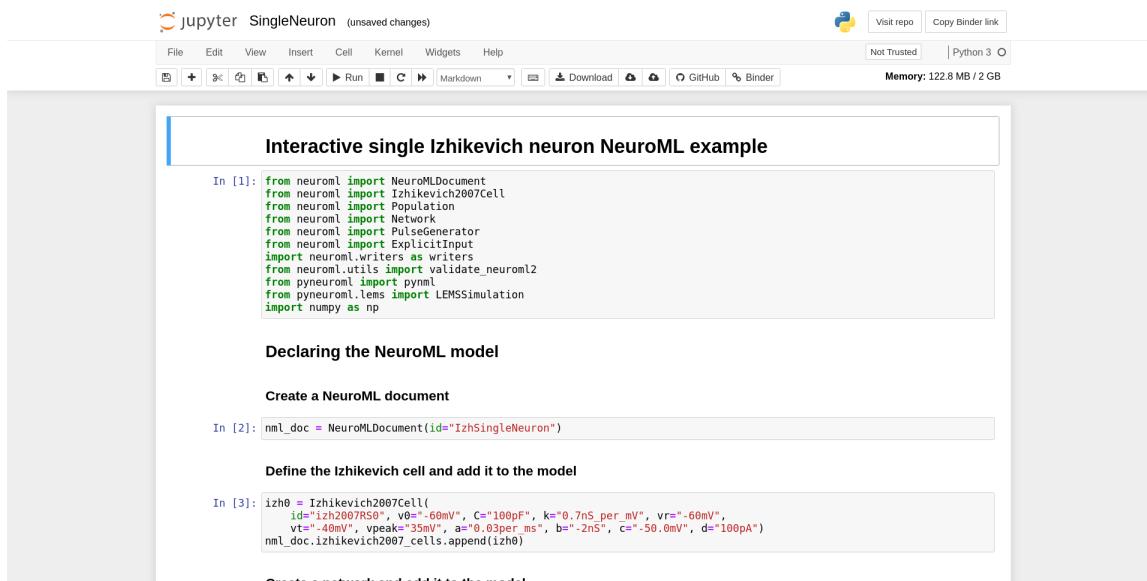
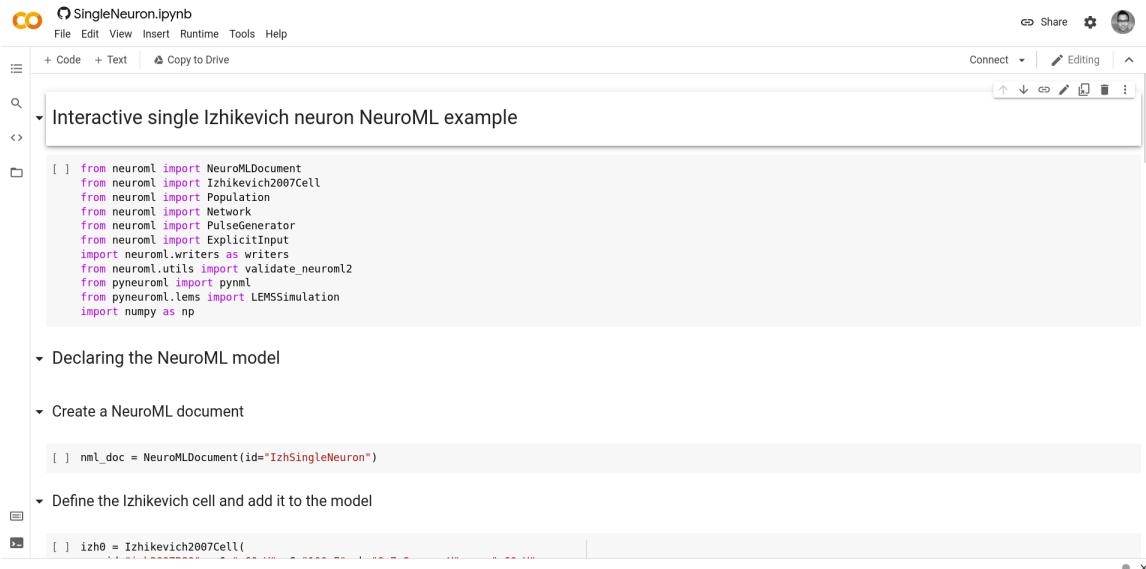


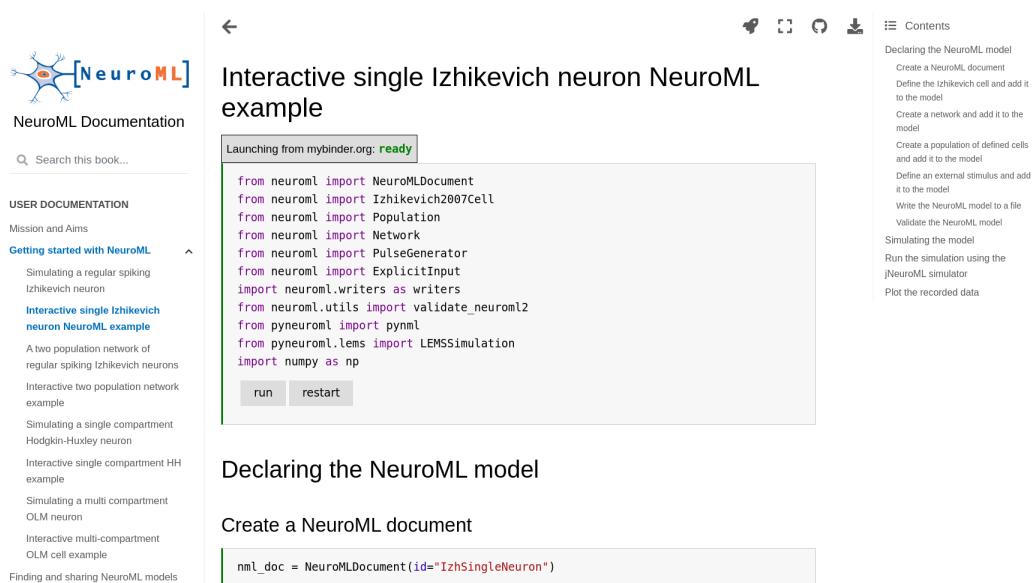
Fig. 2.3: Izhikevich example running in Binder

When running the Jupyter notebooks using these services, you can make changes to the code and re-run it as required.



The screenshot shows a Google Colaboratory notebook titled "SingleNeuron.ipynb". The code cell contains Python imports for NeuroML and Izhikevich models, along with PyNML and PyNEUML libraries. Below the imports, there are sections for "Declaring the NeuroML model" and "Create a NeuroML document". The "Create a NeuroML document" section includes a code cell with the line `nml_doc = NeuroMLDocument(id="IzhSingleNeuron")` and another cell starting with `izh0 = Izhikevich2007Cell(`.

Fig. 2.4: Izhikevich example running in Google Colaboratory.



The screenshot shows the NeuroML Documentation website with the "Interactive single Izhikevich neuron NeuroML example" page selected. On the left is a sidebar with "USER DOCUMENTATION" sections like "Getting started with NeuroML", "Interactive single Izhikevich neuron NeuroML example", and "Declaring the NeuroML model". The main content area displays the same Izhikevich code as in Fig. 2.4, with "run" and "restart" buttons below it. To the right is a "Contents" sidebar listing various NeuroML examples and documentation sections.

Fig. 2.5: Izhikevich example running in Binder but using the Live Code option.

On Binder and Google Colaboratory, which provide the full range of Jupyter Notebook features, you can also run all the code cells at once in sequence. Please see the documentation pages to learn more about using Binder and Google Colaboratory [here](#) and [here](#) respectively. General information on using Jupyter Notebooks and the interface can be found in the documentation [here](#).

2.2.1 Downloading Jupyter Notebooks to run locally on your machine

Jupyter Notebooks can also be downloaded and run locally on your machine. To download the notebooks, use the Download link in the top panel:

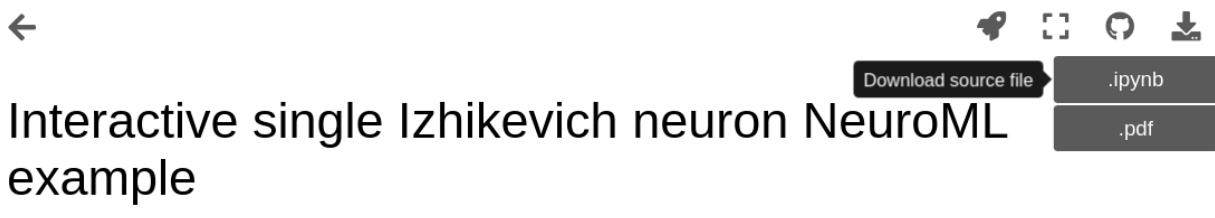


Fig. 2.6: Jupyter notebooks can be downloaded using the Download link in the top panel.

You will need to install the Python Jupyter Notebook packages to do so. Please refer to the Jupyter Notebook [documentation](#) to see how you can install Jupyter Notebooks. Additionally, you will also need to install the [NeuroML software](#) to run these notebooks. Information on using Jupyter Notebooks and the interface can be found in the documentation [here](#).

2.3 Reporting bugs and issues

Please report any issues that you may find in the documentation so that it can be improved. To report an issue on a particular page, you can use the “open issue” link under the GitHub icon in the top panel. Additionally, you can also suggest edits by editing the page in a fork and opening a pull request using the “suggest and edit” link.

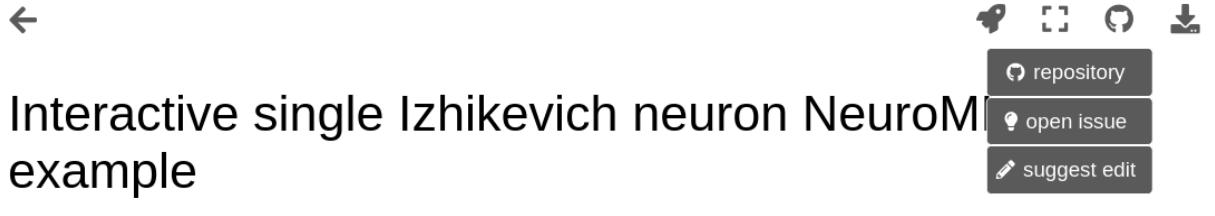


Fig. 2.7: You can report issues and suggest edits to the documentation to help us improve it using the options in the GitHub icon in the top panel.

You can also always contact the NeuroML community using our [communication channels](#) if required.

GETTING STARTED WITH NEUROML

The best way to understand NeuroML is to work through NeuroML examples to see how they are constructed and what they can do. In this chapter, we will step through increasingly complex models to see how they are written and simulated using NeuroML.

To stress that **Python** is the suggested language for developing, simulating, and analysing NeuroML models, we will limit ourselves to the Python NeuroML tools only: *libNeuroML* and *pyNeuroML*. You can learn more about the different NeuroML tools in their *specific sections*.

You do not need to install any software on your computers to run the example code included in this chapter. Each example is followed by a *Jupyter notebook* for you to experiment with. These can be run using the “launch” button in the top right hand corner. Please proceed to the first example section using the navigation button on the right below.

3.1 Simulating a regular spiking Izhikevich neuron

See also the [interactive version](#).

Note: this is a more detailed description of the first example which is available as an *interactive Jupyter notebook* on the next page.

In this section, we wish to simulate a single regular spiking Izhikevich neuron ([Izh07]) and record/visualise its membrane potential (as shown in the figure below):

This plot, saved as `example-single-izhikevich2007cell-sim-v.png`, is generated using the following Python NeuroML script:

```
#!/usr/bin/env python3
"""
Simulating a regular spiking Izhikevich neuron with NeuroML.

File: izhikevich-single-neuron.py
"""

from neuroml import NeuroMLDocument
from neuroml import Izhikevich2007Cell
from neuroml import Population
from neuroml import Network
from neuroml import PulseGenerator
from neuroml import ExplicitInput
import neuroml.writers as writers
from neuroml.utils import validate_neuroml2
```

(continues on next page)

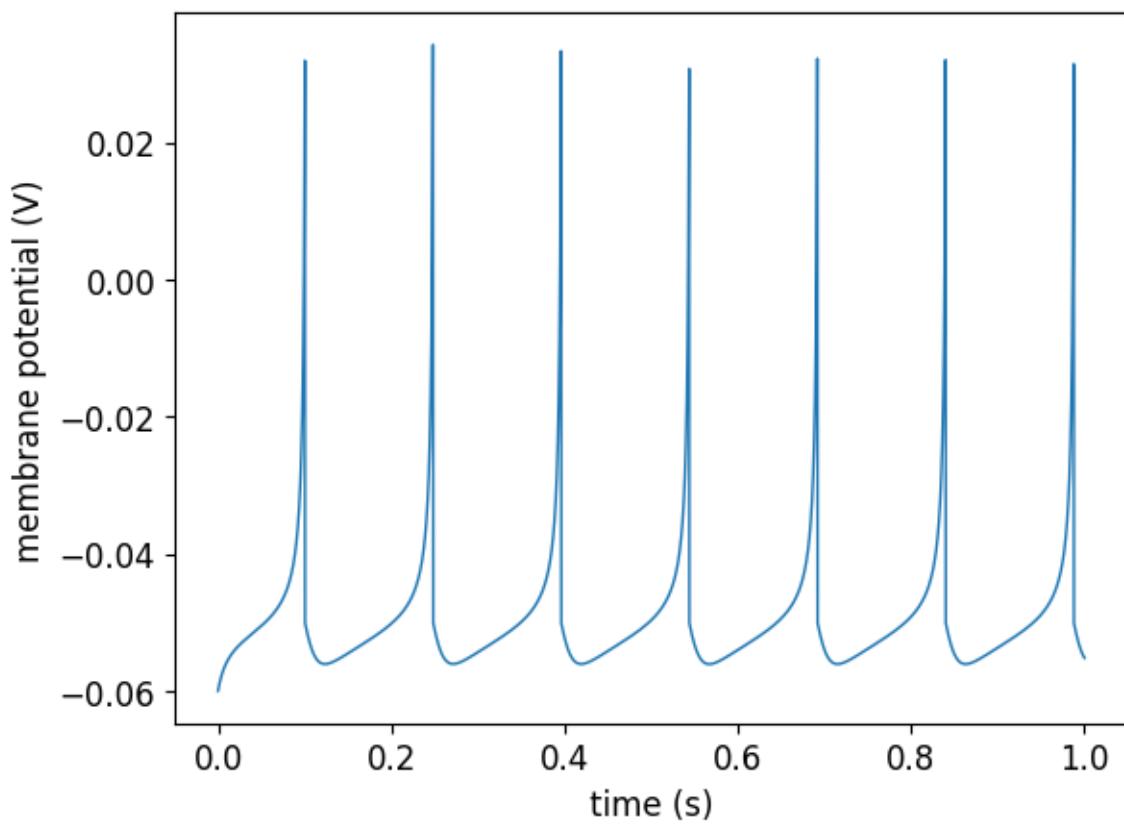


Fig. 3.1: Membrane potential of the simulated regular spiking Izhikevich neuron.

(continued from previous page)

```

from pyneuroml import pynml
from pyneuroml.lemns import LEMSSimulation
import numpy as np

# Create a new NeuroML model document
nml_doc = NeuroMLDocument(id="IzhSingleNeuron")

# Define the Izhikevich cell and add it to the model in the document
izh0 = Izhikevich2007Cell(
    id="izh2007RS0", v0="-60mV", C="100pF", k="0.7nS_per_mV", vr="-60mV",
    vt="-40mV", vpeak="35mV", a="0.03per_ms", b="-2nS", c="-50.0mV", d="100pA")
nml_doc.izhikevich2007_cells.append(izh0)

# Create a network and add it to the model
net = Network(id="IzhNet")
nml_doc.networks.append(net)

# Create a population of defined cells and add it to the model
size0 = 1
pop0 = Population(id="IzhPop0", component=izh0.id, size=size0)
net.populations.append(pop0)

# Define an external stimulus and add it to the model
pg = PulseGenerator(
    id="pulseGen_%i" % 0, delay="0ms", duration="1000ms",
    amplitude="0.07 nA"
)
nml_doc.pulse_generators.append(pg)
exp_input = ExplicitInput(target="%s[%i]" % (pop0.id, 0), input=pg.id)
net.explicit_inputs.append(exp_input)

# Write the NeuroML model to a file
nml_file = 'izhikevich2007_single_cell_network.nml'
writers.NeuroMLWriter.write(nml_doc, nml_file)
print("Written network file to: " + nml_file)

# Validate the NeuroML model against the NeuroML schema
validate_neuroml2(nml_file)

#####
## The NeuroML file has now been created and validated. The rest of the code
## involves writing a LEMS simulation file to run the model

# Create a simulation instance of the model
simulation_id = "example-single-izhikevich2007cell-sim"
simulation = LEMSSimulation(sim_id=simulation_id,
                            duration=1000, dt=0.1, simulation_seed=123)
simulation.assign_simulation_target(net.id)
simulation.include_neuroml2_file(nml_file)

# Define the output file to store simulation outputs
# we record the neuron's membrane potential
simulation.create_output_file(
    "output0", "%s.v.dat" % simulation_id
)

```

(continues on next page)

(continued from previous page)

```
simulation.add_column_to_output_file("output0", 'IzhPop0[0]', 'IzhPop0[0]/v')

# Save the simulation to a file
lems_simulation_file = simulation.save_to_file()

# Run the simulation using the jNeuroML simulator
pynml.run_lems_with_jneuroml(
    lems_simulation_file, max_memory="2G", nogui=True, plot=False
)

# Load the data from the file and plot the graph for the membrane potential
# using the pynml generate_plot utility function.
data_array = np.loadtxt("%s.v.dat" % simulation_id)
pynml.generate_plot(
    [data_array[:, 0]], [data_array[:, 1]],
    "Membrane potential", show_plot_already=False,
    save_figure_to="%s-v.png" % simulation_id,
    xaxis="time (s)", yaxis="membrane potential (V)"
)
```

3.1.1 Declaring the model in NeuroML

Python is the suggested programming language to use for working with NeuroML.

The Python NeuroML tools and libraries provide a convenient, easy to use interface to use NeuroML.

Let us step through the different sections of the Python script. To start writing a model in NeuroML, we first create a NeuroMLDocument. This document is the top level container for everything that the model should contain.

```
# Create a new NeuroML model document
nml_doc = NeuroMLDocument(id="IzhSingleNeuron")
```

Next, all entities that we want to use in the model must be defined. The *NeuroML specification* includes many standard entities, and it is possible to also define new entities that may not already be included in the NeuroML specification. We will look at the pre-defined entities, and how NeuroML may be extended later when we look at the *NeuroML standard* in detail. For now, we limit ourselves to defining a new `Izhikevich2007Cell` (definition of this [here](#)). The Izhikevich neuron model can take sets of parameters to show different types of spiking behaviour. Here, we define an instance of the general Izhikevich cell using parameters that exhibit regular spiking.

Units in NeuroML

NeuroML defines a *standard set of units* that can be used in models. Learn more about units and dimensions in NeuroML and LEMS [here](#).

Once defined, we add this to our NeuroMLDocument.

```
# Define the Izhikevich cell and add it to the model in the document
izh0 = Izhikevich2007Cell(
    id="izh2007RS0", v0="-60mV", C="100pF", k="0.7nS_per_mV", vr="-60mV",
    vt="-40mV", vpeak="35mV", a="0.03per_ms", b="-2nS", c="-50.0mV", d="100pA")
nml_doc.izhikevich2007_cells.append(izh0)
```

Now that the neuron has been defined, we declare a *network* with a *population* of these neurons to create a network. Here, our model includes one network which includes only one population, which in turn only consists of a single neuron. Once the network, its populations, and their neurons have been declared, we again them to our model:

```
# Create a network and add it to the model
net = Network(id="IzhNet")
nml_doc.networks.append(net)

# Create a population of defined cells and add it to the model
size0 = 1
pop0 = Population(id="IzhPop0", component=izh0.id, size=size0)
net.populations.append(pop0)
```

To record the membrane potential of the neuron, we must give it some external input that makes it spike. As with the neuron, we create and add a *pulse generator* to our network. We then connect it to our neuron, the target using an *explicit input*.

```
# Define an external stimulus and add it to the model
pg = PulseGenerator(
    id="pulseGen_%i" % 0, delay="0ms", duration="1000ms",
    amplitude="0.07 nA"
)
nml_doc.pulse_generators.append(pg)
exp_input = ExplicitInput(target="%s[%i]" % (pop0.id, 0), input=pg.id)
net.explicit_inputs.append(exp_input)
```

This completes our model. It includes a single network, with one population of one neuron that is driven by one pulse generator. At this point, we can save our model to a file and validate it to check if it conforms to the NeuroML schema (more on this *later*).

```
# Write the NeuroML model to a file
nml_file = 'izhikevich2007_single_cell_network.nml'
writers.NeuroMLWriter.write(nml_doc, nml_file)
print("Written network file to: " + nml_file)

# Validate the NeuroML model against the NeuroML schema
validate_neuroml2(nml_file)
```

The generated NeuroML model

We have now defined our model in NeuroML. Let us investigate the generated NeuroML file:

```
<neuroml xmlns="http://www.neuroml.org/schema/neuroml2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.neuroml.org/schema/neuroml2 https://raw.github.com/NeuroML/NeuroML2/development/Schemas/NeuroML2/NeuroML_v2.3.xsd" id="IzhSingleNeuron">
    <izhikevich2007Cell id="izh2007RS0" C="100pF" v0="-60mV" k="0.7nS_per_mV" vr="-60mV" vt="-40mV" vpeak="35mV" a="0.03per_ms" b="-2nS" c="-50.0mV" d="100pA"/>
        <pulseGenerator id="pulseGen_0" delay="0ms" duration="1000ms" amplitude="0.07 nA"/>
    <network id="IzhNet">
        <population id="IzhPop0" component="izh2007RS0" size="1"/>
        <explicitInput target="IzhPop0[0]" input="pulseGen_0"/>
```

(continues on next page)

(continued from previous page)

```
</network>
</neuroml>
```

NeuroML files are written in XML. So, they consist of tags and attributes and can be processed by general purpose XML tools. Each entity between chevrons is a *tag*: <..>, and each tag may have multiple *attributes* that are defined using the name=value format. For example <neuroml ..> is a tag, that contains the `id` attribute with value `NML2_SimpleIonChannel`.

XML Tutorial

For details on XML, have a look through [this tutorial](#).

Is this XML well-formed?

A NeuroML file needs to be both 1) well-formed, as it complies with the general rules of the XML language syntax, and 2) valid, i.e. contains the expected NeuroML specific tags/attributes.

Is the XML shown above well-formed? See for yourself. Copy the NeuroML file listed above and check it using an [online XML syntax checker](#).

Let us step through this file to understand the different constructs used in it. The first segment introduces the `neuroml` tag that includes information on the specification that this NeuroML file adheres to.

```
<neuroml xmlns="http://www.neuroml.org/schema/neuroml2" xmlns:xsi="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.neuroml.org/schema/neuroml2 https://raw.github.com/NeuroML/NeuroML2/development/Schemas/NeuroML2/NeuroML_v2.3.xsd" id="IzhSingleNeuron">
```

The first attribute, `xmns` defines the XML *namespace*. All the tags that are defined for use in NeuroML are defined for use in the NeuroML namespace. This prevents conflicts with other XML schemas that may use the same tags. Read more on XML namespaces [here](#).

The remaining lines in this snippet refer to the *XML Schema* that is defined for NeuroML. XML itself does not define any tags, so any tags can be used in a general XML document. Here is an example of a valid XML document, a simple HTML snippet:

```
<html>
<head>
<title>A title</title>
</head>
</html>
```

NeuroML, however, does not use these tags. It defines its own set of standard tags using an [XML Schema](#). In other words, the NeuroML XML schema defines the structure and contents of a valid NeuroML document. Various tools can then compare NeuroML documents to the NeuroML Schema to validate them.

Purpose of the NeuroML schema

The NeuroML Schema defines the structure and contents of a valid NeuroML document.

The `xmlns:xi` attribute documents that NeuroML has a defined XML Schema. The next attribute, `xsi:schemaLocation` tells us the locations of the NeuroML Schema. Here, two locations are provided:

- the Web URL: <http://www.neuroml.org/schema/neuroml2>,
- and the location of the Schema Definition file (an `xsd` file) relative to this example file in the GitHub repository.

We will look at the NeuroML schema in detail in later sections. All NeuroML files must include the `neuroml` tag, and the attributes related to the NeuroML Schema. The last attribute, `id` is the identification (or the name) of this particular NeuroML document.

The remaining part of the file is the *declaration* of the model and its dynamics:

```
<izhikevich2007Cell id="izh2007RS0" C="100pF" v0="-60mV" k="0.7nS_per_mV" vr="-
˓→60mV" vt="-40mV" vpeak="35mV" a="0.03per_ms" b="-2nS" c="-50.0mV" d="100pA"/>
<pulseGenerator id="pulseGen_0" delay="0ms" duration="1000ms" amplitude="0.07 nA"/
˓→>
<network id="IzhNet">
    <population id="IzhPop0" component="izh2007RS0" size="1"/>
    <explicitInput target="IzhPop0[0]" input="pulseGen_0"/>
</network>
```

The cell, is defined in the `izhikevich2007Cell` tag, which has a number of attributes (see [here](#) for more):

- `id`: the name that we want to give to this cell. To refer to it later, for example,
- `v0`: the initial membrane potential for the cell,
- `C`: the leak conductance,
- `k`: conductance per voltage,
- `vr`: the membrane potential after a spike,
- `vt`: the threshold membrane potential, to detect a spike,
- `vpeak`: the peak membrane potential,
- `a, b, c, and d`: are parameters of the Izhikevich neuron model.

Similarly, the `pulseGenerator` is also defined, and the `network` tag includes the `population` and `explicitInput`. We observe that even though we have declared the entities, and the values for parameters that govern them, we do not state what and how these parameters are used. This is because NeuroML is a [declarative language](#) that defines the structure of models. We do not need to define how the dynamics of the different parts of the model are implemented. As we will see further below, these are already defined in NeuroML.

NeuroML is a declarative language.

Users describe the various components of the model but do not need to worry about how they are implemented.

We have seen how an Izhikevich cell can be declared in NeuroML, with all its parameters. However, given that NeuroML develops a standard and defines what tags and attributes can be used, let us see how these are defined for the Izhikevich cell. The Izhikevich cell is defined in version 2 of the NeuroML schema [here](#):

```
<xss:complexType name="Izhikevich2007Cell">
    <xss:complexContent>
        <xss:extension base="BaseCellMembPotCap">
            <xss:attribute name="v0" type="Nml2Quantity_voltage" use="required"/>
            <xss:attribute name="k" type="Nml2Quantity_conductancePerVoltage" use=
˓→"required"/>
```

(continues on next page)

(continued from previous page)

```

<xs:attribute name="vr" type="Nml2Quantity_voltage" use="required"/>
<xs:attribute name="vt" type="Nml2Quantity_voltage" use="required"/>
<xs:attribute name="vpeak" type="Nml2Quantity_voltage" use="required"/>
<br/>
<xs:attribute name="a" type="Nml2Quantity_pertime" use="required"/>
<xs:attribute name="b" type="Nml2Quantity_conductance" use="required"/>
<br/>
<xs:attribute name="c" type="Nml2Quantity_voltage" use="required"/>
<xs:attribute name="d" type="Nml2Quantity_current" use="required"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>

```

The `xs:` prefix indicates that these are all part of an XML Schema. The Izhikevich cell and all its parameters are defined in the schema. As we saw before, parameters of the model are defined as attributes in NeuroML files. So, here in the schema, they are also defined as attributes of the `complexType` that the schema describes. The schema also specifies which of the parameters are necessary, and what their dimensions (units) are using the `use` and `type` properties.

This schema gives us all the information we need to describe an Izhikevich cell in NeuroML. Using the specification in the Schema, any number of Izhikevich cells can be defined in a NeuroML file with the necessary parameter sets to create networks of Izhikevich cells.

As is evident, XML files are excellent for storing structured data, but may not be easy to write by hand. However, NeuroML users *are not expected* to write in XML. They should use the Python tools as demonstrated here.

3.1.2 Simulating the model

Until now, we have just declared the model in NeuroML. We have not, however, included any information related to the simulation of this model, e.g. how long to run it for, what to save from the simulation etc.

With NeuroML v2, the information required to simulate the model is provided using a [LEMS Simulation file](#). We will not go into the details of LEMS just yet. We will limit ourselves to the bits necessary to simulate our Izhikevich neuron only.

The following lines of code instantiate a new simulation with certain simulation parameters: `duration`, `dt`, `simulation_seed`. Additionally, they also define what information is being recorded from the simulation. In this case, we create an output file, and then add a new column to record the membrane potential `v` from our one neuron in the one population in it. You can read more about recording from NeuroML simulations [here](#).

Finally, like we had saved our NeuroML model to a file, we also save our LEMS document to a file.

```

# Create a simulation instance of the model
simulation_id = "example-single-izhikevich2007cell-sim"
simulation = LEMSSimulation(sim_id=simulation_id,
                             duration=1000, dt=0.1, simulation_seed=123)
simulation.assign_simulation_target(net.id)
simulation.include_neuroml2_file(nml_file)

# Define the output file to store simulation outputs
# we record the neuron's membrane potential
simulation.create_output_file(
    "output0", "%s.v.dat" % simulation_id
)
simulation.add_column_to_output_file("output0", 'IzhPop0[0]', 'IzhPop0[0]/v')

```

(continues on next page)

(continued from previous page)

```
# Save the simulation to a file
lems_simulation_file = simulation.save_to_file()
```

The generated LEMS file is shown below:

```
<Lems>

<!--

This LEMS file has been automatically generated using PyNeuroML v0.7.0_
↳ (libNeuroML v0.4.0)

-->

<!-- Specify which component to run -->
<Target component="example-single-izhikevich2007cell-sim"/>

<!-- Include core NeuroML2 ComponentType definitions -->
<Include file="Cells.xml"/>
<Include file="Networks.xml"/>
<Include file="Simulation.xml"/>

<Include file="izhikevich2007_single_cell_network.nml"/>

<Simulation id="example-single-izhikevich2007cell-sim" length="1000ms" step="0.1ms"
↳ " target="IzhNet" seed="123"> <!-- Note seed: ensures same random numbers used
↳ every run -->

    <OutputFile id="output0" fileName="example-single-izhikevich2007cell-sim.v.dat"
↳ ">
        <OutputColumn id="IzhPop0[0]" quantity="IzhPop0[0]/v"/>
    </OutputFile>

</Simulation>

</Lems>
```

Similar to NeuroML, a *LEMS Simulation file* also has a well defined structure, i.e., a set of valid tags which define the contents of the LEMS file. We observe that whereas the NeuroML tags were related to the modelling parameters, the LEMS tags are related to simulation. We also note that our NeuroML model has been “included” in the LEMS file, so that all entities defined there are now known to the LEMS simulation also. Like NeuroML, *users are not expected to write the LEMS XML component by hand*. They should continue to use the NeuroML Python tools.

Finally, *pyNeuroML* also includes functions that allow you to run the simulation from the Python script itself:

```
# Run the simulation using the jNeuroML simulator
pynml.run_lems_with_jneuroml(
    lems_simulation_file, max_memory="2G", nogui=True, plot=False
)
```

Here, we are running our simulation using the *jNeuroML* simulator, which is bundled with *pyNeuroML*. Since NeuroML is a well defined standard, models defined in NeuroML can also be run using other *supported simulators*.

3.1.3 Plotting the recorded membrane potential

Once we have simulated our model and the data has been collected in the specified file, we can analyse the data. pyNeuroML also includes some helpful functions to quickly plot various recorded variables. The last few lines of code shows how the membrane potential plot at the top of the page is generated.

```
# Load the data from the file and plot the graph for the membrane potential
# using the pynml generate_plot utility function.
data_array = np.loadtxt("%s.v.dat" % simulation_id)
pynml.generate_plot(
    [data_array[:, 0]], [data_array[:, 1]],
    "Membrane potential", show_plot_already=False,
    save_figure_to="%s-v.png" % simulation_id,
    xaxis="time (s)", yaxis="membrane potential (V)"
)
```

The next section is an interactive Jupyter notebook where you can play with this example. Click the “launch” button in the top right hand corner to run the notebook in a configured service. *You do not need to install any software on your computer to run these notebooks.*

3.2 Interactive single Izhikevich neuron NeuroML example

To run this interactive Jupyter Notebook, please click on the rocket icon  in the top panel. For more information, please see [how to use this documentation](#). Please uncomment the line below if you use the Google Colab (it does not include these packages by default).

```
#%pip install pyneuroml neuromllite NEURON
```

```
from neuroml import NeuroMLDocument
from neuroml import Izhikevich2007Cell
from neuroml import Population
from neuroml import Network
from neuroml import PulseGenerator
from neuroml import ExplicitInput
import neuroml.writers as writers
from neuroml.utils import validate_neuroml2
from pyneuroml import pynml
from pyneuroml.lems import LEMSSimulation
import numpy as np
```

```
ModuleNotFoundError                                                 Traceback (most recent call last)
Input In [2], in <cell line: 1>()
----> 1 from neuroml import NeuroMLDocument
      2 from neuroml import Izhikevich2007Cell
      3 from neuroml import Population

ModuleNotFoundError: No module named 'neuroml'
```

3.2.1 Declaring the NeuroML model

Create a NeuroML document

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

Define the Izhikevich cell and add it to the model

```
izh0 = Izhikevich2007Cell(
    id="izh2007RS0", v0="-60mV", C="100pF", k="0.7nS_per_mV", vr="-60mV",
    vt="-40mV", vpeak="35mV", a="0.03per_ms", b="-2nS", c="-50.0mV", d="100pA")
nml_doc.izhikevich2007_cells.append(izh0)
```

Create a network and add it to the model

```
net = Network(id="IzhNet")
nml_doc.networks.append(net)
```

Create a population of defined cells and add it to the model

```
size0 = 1
pop0 = Population(id="IzhPop0", component=izh0.id, size=size0)
net.populations.append(pop0)
```

Define an external stimulus and add it to the model

```
pg = PulseGenerator(
    id="pulseGen_%i" % 0, delay="0ms", duration="1000ms",
    amplitude="0.07 nA"
)
nml_doc.pulse_generators.append(pg)
exp_input = ExplicitInput(target="%s[%i]" % (pop0.id, 0), input=pg.id)
net.explicit_inputs.append(exp_input)
```

Write the NeuroML model to a file

```
nml_file = 'izhikevich2007_single_cell_network.nml'
writers.NeuroMLWriter.write(nml_doc, nml_file)
print("Written network file to: " + nml_file)
```

Written network file to: izhikevich2007_single_cell_network.nml

Validate the NeuroML model

```
validate_neuroml2(nml_file)
```

```
Validating izhikevich2007_single_cell_network.nml against /Users/padraig/anaconda/
  ↵envs/py37/lib/python3.7/site-packages/libNeuroML-0.2.56-py3.7.egg/neuroml/nml/
  ↵NeuroML_v2.2.xsd
It's valid!
```

3.2.2 Simulating the model

Create a simulation instance of the model

```
simulation_id = "example-single-izhikevich2007cell-sim"
simulation = LEMSSimulation(sim_id=simulation_id,
                            duration=1000, dt=0.1, simulation_seed=123)
simulation.assign_simulation_target(net.id)
simulation.include_neuroml2_file(nml_file)
```

Define the output file to store simulation outputs

Here, we record the neuron's membrane potential to the specified data file.

```
simulation.create_output_file(
    "output0", "%s.v.dat" % simulation_id
)
simulation.add_column_to_output_file("output0", 'IzhPop0[0]', 'IzhPop0[0]/v')
```

Save the simulation to a file

```
lems_simulation_file = simulation.save_to_file()
```

```
pyNeuroML >>> Written LEMS Simulation example-single-izhikevich2007cell-sim to_
  ↵file: LEMS_example-single-izhikevich2007cell-sim.xml
```

3.2.3 Run the simulation using the jNeuroML simulator

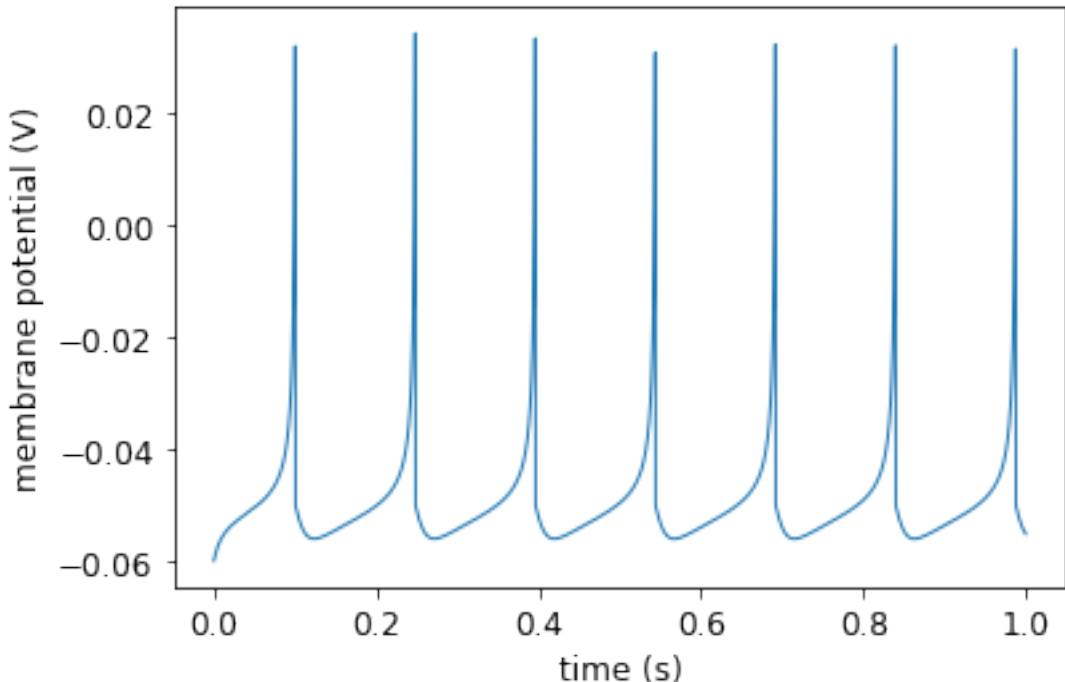
```
pynml.run_lems_with_jneuroml(
    lems_simulation_file, max_memory="2G", nogui=True, plot=False
)
```

```
True
```

3.2.4 Plot the recorded data

```
# Load the data from the file and plot the graph for the membrane potential
# using the pynml generate_plot utility function.
data_array = np.loadtxt("%s.v.dat" % simulation_id)
pynml.generate_plot(
    [data_array[:, 0], [data_array[:, 1]],
    "Membrane potential", show_plot_already=True,
    xaxis="time (s)", yaxis="membrane potential (V)"
)
```

pyNeuroML >>> Generating plot: Membrane potential



<AxesSubplot:xlabel='time (s)', ylabel='membrane potential (V)'>

3.3 A two population network of regular spiking Izhikevich neurons

Now that we have defined a cell, let us see how a network of these cells may be declared and simulated. We will create a small network of cells, simulate this network, and generate a plot of the spike times of the cells (a raster plot):

The Python script used to create the model, simulate it, and generate this plot is below:

```
#!/usr/bin/env python3
"""
Create a simple network with two populations.
"""

from neuroml import NeuroMLDocument
```

(continues on next page)

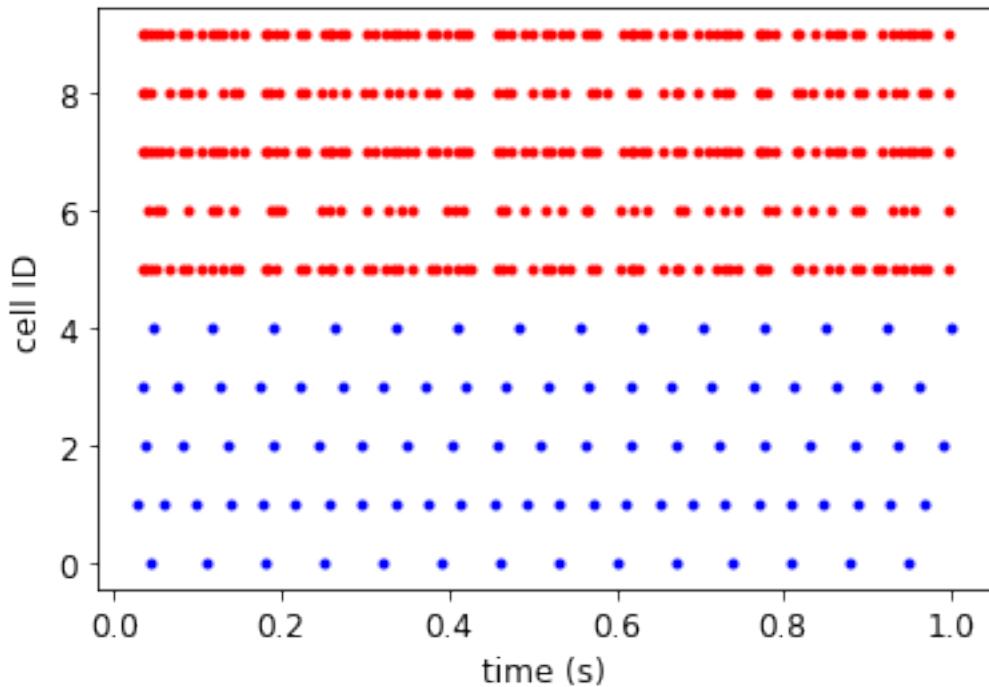


Fig. 3.2: Spike times of neurons in 2 populations recorded from the simulation.

(continued from previous page)

```

from neuroml import Izhikevich2007Cell
from neuroml import Network
from neuroml import ExpOneSynapse
from neuroml import Population
from neuroml import Projection
from neuroml import PulseGenerator
from neuroml import ExplicitInput
from neuroml import Connection
from neuroml import Property
import neuroml.writers as writers
import random
from pyneuroml import pynml
from pyneuroml.lemns import LEMSSimulation
import numpy as np

nml_doc = NeuroMLDocument(id="IzNet")

iz0 = Izhikevich2007Cell(
    id="iz2007RS0", v0="-60mV", C="100pF", k="0.7nS_per_mV", vr="-60mV",
    vt="-40mV", vpeak="35mV", a="0.03per_ms", b="-2nS", c="-50.0mV", d="100pA")
nml_doc.izhikevich2007_cells.append(iz0)

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

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

```

(continues on next page)

(continued from previous page)

```

size0 = 5
pop0 = Population(id="IzPop0", component=iz0.id, size=size0)
# Set optional color property. Note: used later when generating graphs etc.
pop0.properties.append(Property(tag='color', value='0 0 .8'))
net.populations.append(pop0)

size1 = 5
pop1 = Population(id="IzPop1", component=iz0.id, size=size1)
pop1.properties.append(Property(tag='color', value='.8 0 0'))
net.populations.append(pop1)

proj = Projection(id='proj', presynaptic_population=pop0.id,
                   postsynaptic_population=pop1.id, synapse=syn0.id)
net.projections.append(proj)

random.seed(123)
prob_connection = 0.8
count = 0
for pre in range(0, size0):
    pg = PulseGenerator(
        id="pg_%i" % pre, delay="0ms", duration="10000ms",
        amplitude="%f nA" % (0.1 + 0.1 * random.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):
    if random.random() <= prob_connection:
        syn = Connection(id=count,
                          pre_cell_id=".../%s[%i]" % (pop0.id, pre),
                          synapse=syn0.id,
                          post_cell_id=".../%s[%i]" % (pop1.id, post))
        proj.connections.append(syn)
        count += 1

nml_file = 'izhikevich2007_network.nml'
writers.NeuroMLWriter.write(nml_doc, nml_file)

print("Written network file to: " + nml_file)
pynml.validate_neuroml2(nml_file)

simulation_id = "example_izhikevich2007network_sim"
simulation = LEMSSimulation(sim_id=simulation_id,
                             duration=1000, dt=0.1, simulation_seed=123)
simulation.assign_simulation_target(net.id)
simulation.include_neuroml2_file(nml_file)

simulation.create_event_output_file(
    "pop0", "%s.0.spikes.dat" % simulation_id, format='ID_TIME'
)
for pre in range(0, size0):
    simulation.add_selection_to_event_output_file(
        "pop0", pre, 'IzPop0[{}].format(pre)', 'spike')

```

(continues on next page)

(continued from previous page)

```

simulation.create_event_output_file(
    "pop1", "%s.1.spikes.dat" % simulation_id, format='ID_TIME'
)
for pre in range(0, size1):
    simulation.add_selection_to_event_output_file(
        "pop1", pre, 'IzPop1[{}]'.format(pre), 'spike')

lems_simulation_file = simulation.save_to_file()

pynml.run_lems_with_jneuroml_neuron(
    lems_simulation_file, max_memory="2G", nogui=True, plot=False
)

# Load the data from the file and plot the spike times
# using the pynml generate_plot utility function.
data_array_0 = np.loadtxt("%s.0.spikes.dat" % simulation_id)
data_array_1 = np.loadtxt("%s.1.spikes.dat" % simulation_id)
times_0 = data_array_0[:,1]
times_1 = data_array_1[:,1]
ids_0 = data_array_0[:,0]
ids_1 = [id+size0 for id in data_array_1[:,0]]
pynml.generate_plot(
    [times_0,times_1], [ids_0,ids_1],
    "Spike times", show_plot_already=False,
    save_figure_to="%s-spikes.png" % simulation_id,
    xaxis="time (s)", yaxis="cell ID",
    colors=['b','r'],
    linewidths=[0,0], markers=[.,.],
)

```

As with the previous example, we will step through this script to see how the various components of the network are declared in NeuroML before running the simulation and generating the plot.

3.3.1 Declaring the model in NeuroML

To declare the complete network model, we must again first declare its core entities:

```

nml_doc = NeuroMLDocument(id="IzNet")

iz0 = Izhikevich2007Cell(
    id="iz2007RS0", v0="-60mV", C="100pF", k="0.7nS_per_mV", vr="-60mV",
    vt="-40mV", vpeak="35mV", a="0.03per_ms", b="-2nS", c="-50.0mV", d="100pA")
nml_doc.izhikevich2007_cells.append(iz0)

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

```

Here, we create a new document, declare the *Izhikevich neuron*, and also declare the synapse that we are going to use to connect one population of neurons to the other. Here we intend to use the *ExpOne Synapse*, where the conductance of the synapse increases instantaneously by a constant value `gbase` on receiving a spike, and then decays exponentially with a decay constant `tauDecay`.

We can now declare our *network* with 2 *populations* of these cells. Note: setting a color as a *property* is optional, but is used in some other tools like generating graphs below.

```

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

size0 = 5
pop0 = Population(id="IzPop0", component=iz0.id, size=size0)
# Set optional color property. Note: used later when generating graphs etc.
pop0.properties.append(Property(tag='color', value='0 0 .8'))
net.populations.append(pop0)

size1 = 5
pop1 = Population(id="IzPop1", component=iz0.id, size=size1)
pop1.properties.append(Property(tag='color', value='.8 0 0'))

```

Next, we create *projections* between the two populations based on some probability of connection. To do this, we iterate over each post-synaptic neuron for each pre-synaptic neuron and draw a random number between 0 and 1. If the drawn number is less than the required probability of connection, the connection is created.

While we are iterating over all our pre-synaptic cells here, we also add external inputs to them using *ExplicitInputs* (this could have been done in a different loop, but it is convenient to also do this here).

```

proj = Projection(id='proj', presynaptic_population=pop0.id,
                  postsynaptic_population=pop1.id, synapse=syn0.id)
net.projections.append(proj)

random.seed(123)
prob_connection = 0.8
count = 0
for pre in range(0, size0):
    pg = PulseGenerator(
        id="pg_%i" % pre, delay="0ms", duration="10000ms",
        amplitude="%f nA" % (0.1 + 0.1 * random.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):
    if random.random() <= prob_connection:
        syn = Connection(id=count,
                          pre_cell_id=".../%s[%i]" % (pop0.id, pre),
                          synapse=syn0.id,
                          post_cell_id=".../%s[%i]" % (pop1.id, post))
        proj.connections.append(syn)
        count += 1

```

We can now save and validate our model, as before:

```

nml_file = 'izhikevich2007_network.nml'
writers.NeuroMLWriter.write(nml_doc, nml_file)

print("Written network file to: " + nml_file)
pynml.validate_neuroml2(nml_file)

```

The generated NeuroML model

Let us take a look at the generated NeuroML model

```

<neuroml xmlns="http://www.neuroml.org/schema/neuroml2" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.neuroml.org/schema/neuroml2 https://raw.github.com/NeuroML/NeuroML2/development/Schemas/NeuroML2/NeuroML_v2.3.xsd" id="IzNet">
    <expOneSynapse id="syn0" gbase="65nS" erev="0mV" tauDecay="3ms"/>
    <izhikevich2007Cell id="iz2007RS0" C="100pF" v0="-60mV" k="0.7nS_per_mV" vr="-60mV" vt="-40mV" vpeak="35mV" a="0.03per_ms" b="-2ns" c="-50.0mV" d="100pA"/>
    <pulseGenerator id="pg_0" delay="0ms" duration="10000ms" amplitude="0.105236 nA"/>
    <pulseGenerator id="pg_1" delay="0ms" duration="10000ms" amplitude="0.153620 nA"/>
    <pulseGenerator id="pg_2" delay="0ms" duration="10000ms" amplitude="0.124516 nA"/>
    <pulseGenerator id="pg_3" delay="0ms" duration="10000ms" amplitude="0.131546 nA"/>
    <pulseGenerator id="pg_4" delay="0ms" duration="10000ms" amplitude="0.102124 nA"/>
    <network id="IzNet">
        <population id="IzPop0" component="iz2007RS0" size="5"/>
        <population id="IzPop1" component="iz2007RS0" size="5"/>
        <projection id="proj" presynapticPopulation="IzPop0" postsynapticPopulation="IzPop1" synapse="syn0">
            <connection id="0" preCellId="..../IzPop0[0]" postCellId="..../IzPop1[0]"/>
            <connection id="1" preCellId="..../IzPop0[0]" postCellId="..../IzPop1[1]"/>
            <connection id="2" preCellId="..../IzPop0[0]" postCellId="..../IzPop1[2]"/>
            <connection id="3" preCellId="..../IzPop0[0]" postCellId="..../IzPop1[4]"/>
            <connection id="4" preCellId="..../IzPop0[1]" postCellId="..../IzPop1[0]"/>
            <connection id="5" preCellId="..../IzPop0[1]" postCellId="..../IzPop1[2]"/>
            <connection id="6" preCellId="..../IzPop0[1]" postCellId="..../IzPop1[3]"/>
            <connection id="7" preCellId="..../IzPop0[1]" postCellId="..../IzPop1[4]"/>
            <connection id="8" preCellId="..../IzPop0[2]" postCellId="..../IzPop1[0]"/>
            <connection id="9" preCellId="..../IzPop0[2]" postCellId="..../IzPop1[1]"/>
            <connection id="10" preCellId="..../IzPop0[2]" postCellId="..../IzPop1[2]"/>
            <connection id="11" preCellId="..../IzPop0[2]" postCellId="..../IzPop1[3]"/>
            <connection id="12" preCellId="..../IzPop0[2]" postCellId="..../IzPop1[4]"/>
            <connection id="13" preCellId="..../IzPop0[3]" postCellId="..../IzPop1[0]"/>
            <connection id="14" preCellId="..../IzPop0[3]" postCellId="..../IzPop1[2]"/>
            <connection id="15" preCellId="..../IzPop0[3]" postCellId="..../IzPop1[3]"/>
            <connection id="16" preCellId="..../IzPop0[3]" postCellId="..../IzPop1[4]"/>
            <connection id="17" preCellId="..../IzPop0[4]" postCellId="..../IzPop1[1]"/>
            <connection id="18" preCellId="..../IzPop0[4]" postCellId="..../IzPop1[2]"/>
            <connection id="19" preCellId="..../IzPop0[4]" postCellId="..../IzPop1[4]"/>
        </projection>
        <explicitInput target="IzPop0[0]" input="pg_0"/>
        <explicitInput target="IzPop0[1]" input="pg_1"/>
        <explicitInput target="IzPop0[2]" input="pg_2"/>
        <explicitInput target="IzPop0[3]" input="pg_3"/>
        <explicitInput target="IzPop0[4]" input="pg_4"/>
    </network>
</neuroml>
```

It is easy to see how the model is clearly declared in the NeuroML file. Observe how entities are referenced in NeuroML depending on their location in the document architecture. Here, *population* and *projection* are at the same level. The synaptic connections using the *connection* tag are at the next level. So, in the *connection* tags, populations are to be referred to as *.. /* which indicates the previous level. The *explicitinput* tag is at the same level as the *population* and *projection* tags, so we do *not* need to use *.. /* here to reference them.

Another point worth noting here is that because we've defined a population of the same components by specifying a size rather than by individually adding components to it, we can refer to the entities of the population using the common *[..]*

index operator.

The advantage of such a declarative format is that we can also easily get information on our model from the NeuroML file. `pyNeuroML` includes the helper `pynml-summary` script:

```
$ pynml-summary izhikevich2007_network.nml
*****
* NeuroMLDocument: IzNet
*
* ExpOneSynapse: ['syn0']
* Izhikevich2007Cell: ['iz2007RS0']
* PulseGenerator: ['pulseGen_0', 'pulseGen_1', 'pulseGen_2', 'pulseGen_3',
  ↵'pulseGen_4']
*
* Network: IzNet
*
* 10 cells in 2 populations
* Population: IzPop0 with 5 components of type iz2007RS0
* Population: IzPop1 with 5 components of type iz2007RS0
*
* 20 connections in 1 projections
* Projection: proj from IzPop0 to IzPop1, synapse: syn0
* 20 connections: [(Connection 0: 0 -> 0), ...]
*
* 0 inputs in 0 input lists
*
* 5 explicit inputs (outside of input lists)
* Explicit Input of type pulseGen_0 to IzPop0(cell 0), destination: unspecified
* Explicit Input of type pulseGen_1 to IzPop0(cell 1), destination: unspecified
* Explicit Input of type pulseGen_2 to IzPop0(cell 2), destination: unspecified
* Explicit Input of type pulseGen_3 to IzPop0(cell 3), destination: unspecified
* Explicit Input of type pulseGen_4 to IzPop0(cell 4), destination: unspecified
*
*****
```

We can also generate a graphical summary of our model using `pynml` from `pyNeuroML`:

```
$ pynml izhikevich2007_network.nml -graph 3
```

This generates the following model summary diagram:

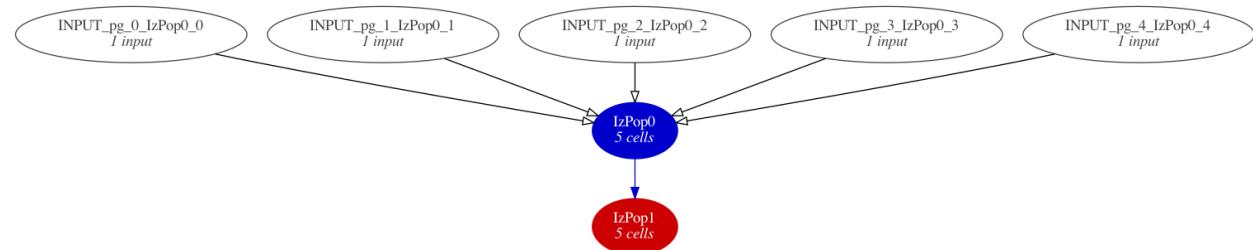


Fig. 3.3: A summary graph of the model generated using `pynml` and the dot tool.

Other options for `pynml` produce other views, e.g individual connections:

```
$ pynml izhikevich2007_network.nml -graph -1
```

In our very simple network here, neurons do not have morphologies and are not distributed in space. In later examples,

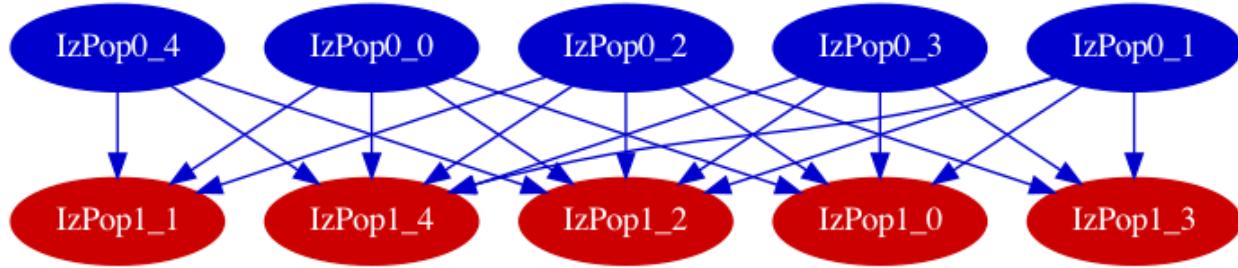


Fig. 3.4: Model summary graph showing individual connections between cells in the populations.

however, we will also see how summary figures of the network that show the morphologies, locations of different layers and neurons, and so on can also be generated using the NeuroML tools.

3.3.2 Simulating the model

Now that we have our model set up, we can proceed to simulating it. We create our simulation, and setup the information we want to record from it.

```
simulation_id = "example_izhikevich2007network_sim"
simulation = LEMSSimulation(sim_id=simulation_id,
                             duration=1000, dt=0.1, simulation_seed=123)
simulation.assign_simulation_target(net.id)
simulation.include_neuroml2_file(nml_file)

simulation.create_event_output_file(
    "pop0", "%s.0.spikes.dat" % simulation_id, format='ID_TIME'
)
for pre in range(0, size0):
    simulation.add_selection_to_event_output_file(
        "pop0", pre, 'IzPop0[{}]'.format(pre), 'spike')

simulation.create_event_output_file(
    "pop1", "%s.1.spikes.dat" % simulation_id, format='ID_TIME'
)
for pre in range(0, size1):
    simulation.add_selection_to_event_output_file(
        "pop1", pre, 'IzPop1[{}]'.format(pre), 'spike')

lems_simulation_file = simulation.save_to_file()
```

The generated LEMS file is here:

```
<Lems>

<!--

This LEMS file has been automatically generated using PyNeuroML v0.7.0
→(libNeuroML v0.4.0)

-->

<!-- Specify which component to run -->
```

(continues on next page)

(continued from previous page)

```

<Target component="example_izhikevich2007network_sim"/>

<!-- Include core NeuroML2 ComponentType definitions -->
<Include file="Cells.xml"/>
<Include file="Networks.xml"/>
<Include file="Simulation.xml"/>

<Include file="izhikevich2007_network.nml"/>

<Simulation id="example_izhikevich2007network_sim" length="1000ms" step="0.1ms"
  target="IzNet" seed="123"> <!-- Note seed: ensures same random numbers used every
  run -->

  <EventOutputFile id="pop0" fileName="example_izhikevich2007network_sim.0.
  spikes.dat" format="ID_TIME">
    <EventSelection id="0" select="IzPop0[0]" eventPort="spike"/>
    <EventSelection id="1" select="IzPop0[1]" eventPort="spike"/>
    <EventSelection id="2" select="IzPop0[2]" eventPort="spike"/>
    <EventSelection id="3" select="IzPop0[3]" eventPort="spike"/>
    <EventSelection id="4" select="IzPop0[4]" eventPort="spike"/>
  </EventOutputFile>

  <EventOutputFile id="pop1" fileName="example_izhikevich2007network_sim.1.
  spikes.dat" format="ID_TIME">
    <EventSelection id="0" select="IzPop1[0]" eventPort="spike"/>
    <EventSelection id="1" select="IzPop1[1]" eventPort="spike"/>
    <EventSelection id="2" select="IzPop1[2]" eventPort="spike"/>
    <EventSelection id="3" select="IzPop1[3]" eventPort="spike"/>
    <EventSelection id="4" select="IzPop1[4]" eventPort="spike"/>
  </EventOutputFile>

</Simulation>

</Lems>

```

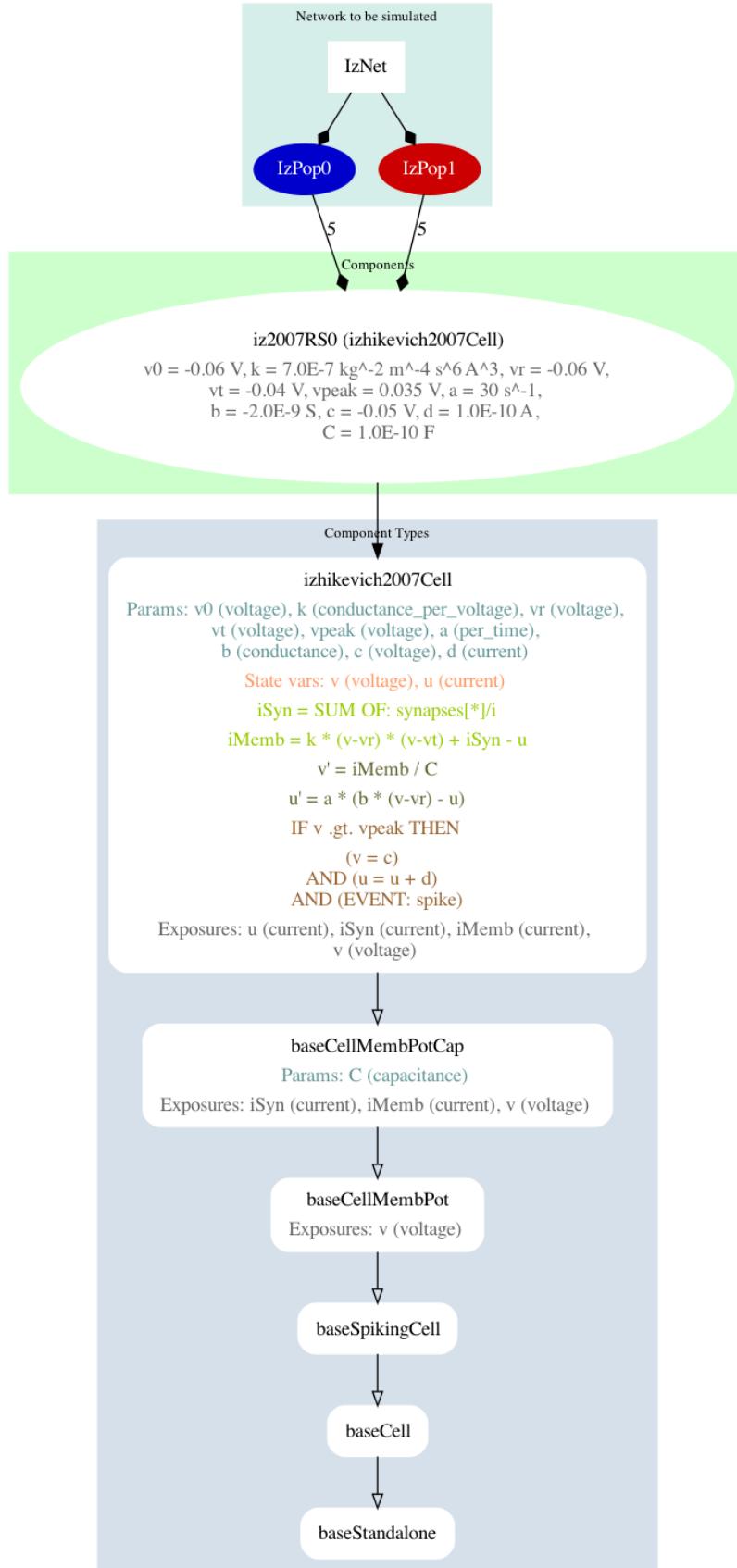
Where we had generated a graphical summary of the model before, we can now also generate graphical summaries of the simulation using pynml and the `-lems-graph` option. This dives deeper into the LEMS definition of the cells, showing more of the underlying dynamics of the components:

```
$ pynml LEMS_example_izhikevich2007network_sim.xml -lems-graph
```

Here is the generated summary graph:

It shows a top-down breakdown of the simulation: from the network, to the populations, to the cell types, leading up to the components that these cells are made of (more on Components later). Let us add the necessary code to run our simulation, this time using the well known NEURON simulator:

```
pynml.run_lems_with_jneuroml_neuron(
  lems_simulation_file, max_memory="2G", nogui=True, plot=False
)
```



3.3.3 Plotting recorded spike times

To analyse the outputs from the simulation, we can again plot the information we recorded. In the previous example, we had recorded and plotted the membrane potentials from our cell. Here, we have recorded the spike times. So let us plot them to generate our figure:

```
# Load the data from the file and plot the spike times
# using the pynml generate_plot utility function.
data_array_0 = np.loadtxt("%s.0.spikes.dat" % simulation_id)
data_array_1 = np.loadtxt("%s.1.spikes.dat" % simulation_id)
times_0 = data_array_0[:,1]
times_1 = data_array_1[:,1]
ids_0 = data_array_0[:,0]
ids_1 = [id+size0 for id in data_array_1[:,0]]
pynml.generate_plot(
    [times_0,times_1], [ids_0,ids_1],
    "Spike times", show_plot_already=False,
    save_figure_to="%s-spikes.png" % simulation_id,
    xaxis="time (s)", yaxis="cell ID",
    colors=['b','r'],
    linewidths=[1,1], markers=[1,1],
)
```

Observe how we are using the same `generate_plot` utility function as before: it is general enough to plot different recorded quantities. Under the hood, it passes this information to Python's Matplotlib library. This produces the rasterplot shown at the top of the page.

This concludes our second example. Here, we have seen how to create, simulate, and record from a simple two population network of single compartment point neurons. The next section is an interactive notebook that you can use to play with this example. After that we will move on to the next example: a neuron model using Hodgkin Huxley style ion channels.

3.4 Interactive two population network example

To run this interactive Jupyter Notebook, please click on the rocket icon  in the top panel. For more information, please see [how to use this documentation](#). Please uncomment the line below if you use the Google Colab. (It does not include these packages by default).

```
#%pip install pyneuroml neuromllite NEURON
```

```
#!/usr/bin/env python3
"""
Create a simple network with two populations.
"""

from neuroml import NeuroMLDocument
from neuroml import Izhikevich2007Cell
from neuroml import Network
from neuroml import ExpOneSynapse
from neuroml import Population
from neuroml import Projection
from neuroml import PulseGenerator
from neuroml import ExplicitInput
from neuroml import Connection
```

(continues on next page)

(continued from previous page)

```
from neuroml import Property
import neuroml.writers as writers
import random
from pyneuroml import pymml
from pyneuroml.lems import LEMSSimulation
import numpy as np
```

```
ModuleNotFoundError                                     Traceback (most recent call last)
Input In [2], in <cell line: 6>()
      1 #!/usr/bin/env python3
      2 """
      3 Create a simple network with two populations.
      4 """
----> 6 from neuroml import NeuroMLDocument
      7 from neuroml import Izhikevich2007Cell
      8 from neuroml import Network

ModuleNotFoundError: No module named 'neuroml'
```

3.4.1 Declaring the NeuroML model

Create a NeuroML document

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

Declare the Izhikevich cell and add it to the model document

```
iz0 = Izhikevich2007Cell(
    id="iz2007RS0", v0="-60mV", C="100pF", k="0.7nS_per_mV", vr="-60mV",
    vt="-40mV", vpeak="35mV", a="0.03per_ms", b="-2nS", c="-50.0mV", d="100pA")
nml_doc.izhikevich2007_cells.append(iz0)
```

Declare the Synapse and add it to the model document

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

Declare a Network and add it to the model document

```
net = Network(id="IzNet")
nml_doc.networks.append(net)
```

Create two populations

```
size0 = 5
pop0 = Population(id="IzPop0", component=iz0.id, size=size0)
net.populations.append(pop0)

size1 = 5
pop1 = Population(id="IzPop1", component=iz0.id, size=size1)
net.populations.append(pop1)
```

Declare projections

```
proj = Projection(id='proj', presynaptic_population=pop0.id,
                  postsynaptic_population=pop1.id, synapse=syn0.id)
net.projections.append(proj)
```

Add the projections between populations and the external inputs

```
random.seed(123)
prob_connection = 0.8
count = 0
for pre in range(0, size0):
    pg = PulseGenerator(
        id="pg_%i" % pre, delay="0ms", duration="10000ms",
        amplitude="%f nA" % (0.1 + 0.1 * random.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):
        if random.random() <= prob_connection:
            syn = Connection(id=count,
                              pre_cell_id=".../%s[%i]" % (pop0.id, pre),
                              synapse=syn0.id,
                              post_cell_id=".../%s[%i]" % (pop1.id, post))
```

(continues on next page)

(continued from previous page)

```
proj.connections.append(syn)
count += 1
```

Write the NeuroML model to a NeuroML file and validate it

```
nml_file = 'izhikevich2007_network.nml'
writers.NeuroMLWriter.write(nml_doc, nml_file)

print("Written network file to: " + nml_file)
pynml.validate_neuroml2(nml_file)
```

```
Written network file to: izhikevich2007_network.nml
pyNeuroML >>> Running jnml on izhikevich2007_network.nml with pre args: [-
    -validate], post args: [], in dir: ., verbose: True, report: True, exit on fail:-
    -False
pyNeuroML >>> Executing: (java -Xmx400M -jar "/home/padraig/anaconda2/envs/py37/
    -lib/python3.7/site-packages/pyNeuroML-0.5.13-py3.7.egg/pyneuroml/lib/jNeuroML-0.
    -11.0-jar-with-dependencies.jar" -validate "izhikevich2007_network.nml" ) in_
    -directory: .
pyNeuroML >>> Command completed. Output:
pyNeuroML >>>     jNeuroML >>     jNeuroML v0.11.0
pyNeuroML >>>     jNeuroML >>     Validating: /home/padraig/git/Documentation/source/
    -Userdocs/NML2_examples/izhikevich2007_network.nml
pyNeuroML >>>     jNeuroML >>     Valid against schema and all tests
pyNeuroML >>>     jNeuroML >>     No warnings
pyNeuroML >>>     jNeuroML >>
pyNeuroML >>>     jNeuroML >>     Validated 1 files: All valid and no warnings
pyNeuroML >>>     jNeuroML >>
pyNeuroML >>>     jNeuroML >>
pyNeuroML >>>     Successfully ran the following command using pyNeuroML v0.5.13:
pyNeuroML >>>         java -Xmx400M -jar "/home/padraig/anaconda2/envs/py37/lib/
    -python3.7/site-packages/pyNeuroML-0.5.13-py3.7.egg/pyneuroml/lib/jNeuroML-0.11.0-
    -jar-with-dependencies.jar" -validate "izhikevich2007_network.nml"
pyNeuroML >>>     Output:
pyNeuroML >>>
pyNeuroML >>>     jNeuroML v0.11.0
pyNeuroML >>>     Validating: /home/padraig/git/Documentation/source/Userdocs/NML2_-
    -examples/izhikevich2007_network.nml
pyNeuroML >>>     Valid against schema and all tests
pyNeuroML >>>     No warnings
pyNeuroML >>>
pyNeuroML >>>     Validated 1 files: All valid and no warnings
pyNeuroML >>>
pyNeuroML >>>
```

```
True
```

3.4.2 Simulating the model

Create a simulation instance of the model

```
simulation_id = "example_izhikevich2007network_sim"
simulation = LEMSSimulation(sim_id=simulation_id,
                             duration=1000, dt=0.1, simulation_seed=123)
simulation.assign_simulation_target(net.id)
simulation.include_neuroml2_file(nml_file)
```

Define the output file to store spikes

```
simulation.create_event_output_file(
    "pop0", "%s.0.spikes.dat" % simulation_id, format='ID_TIME'
)
for pre in range(0, size0):
    simulation.add_selection_to_event_output_file(
        "pop0", pre, 'IzPop0[{}].format(pre), 'spike')

simulation.create_event_output_file(
    "pop1", "%s.1.spikes.dat" % simulation_id, format='ID_TIME'
)
for pre in range(0, size1):
    simulation.add_selection_to_event_output_file(
        "pop1", pre, 'IzPop1[{}].format(pre), 'spike')
```

```
/home/padraig/anaconda2/envs/py37/lib/python3.7/site-packages/ipykernel/ipkern
el.py:287: DeprecationWarning: `should_run_async` will not call `transform_cell`_
automatically in the future. Please pass the result to `transformed_cell`_
argument and any exception that happen during the transform in `preprocessin
g_exc_tuple` in IPython 7.17 and above.
and should_run_async(code)
```

3.4.3 Save the simulation to a file

```
lems_simulation_file = simulation.save_to_file()
```

```
pyNeuroML >>> Written LEMS Simulation example_izhikevich2007network_sim to file:_
LEMS_example_izhikevich2007network_sim.xml
```

3.4.4 Run the simulation using jNeuroML

```
pynml.run_lems_with_jneuroml(  
    lems_simulation_file, max_memory="2G", nogui=True, plot=False  
)
```

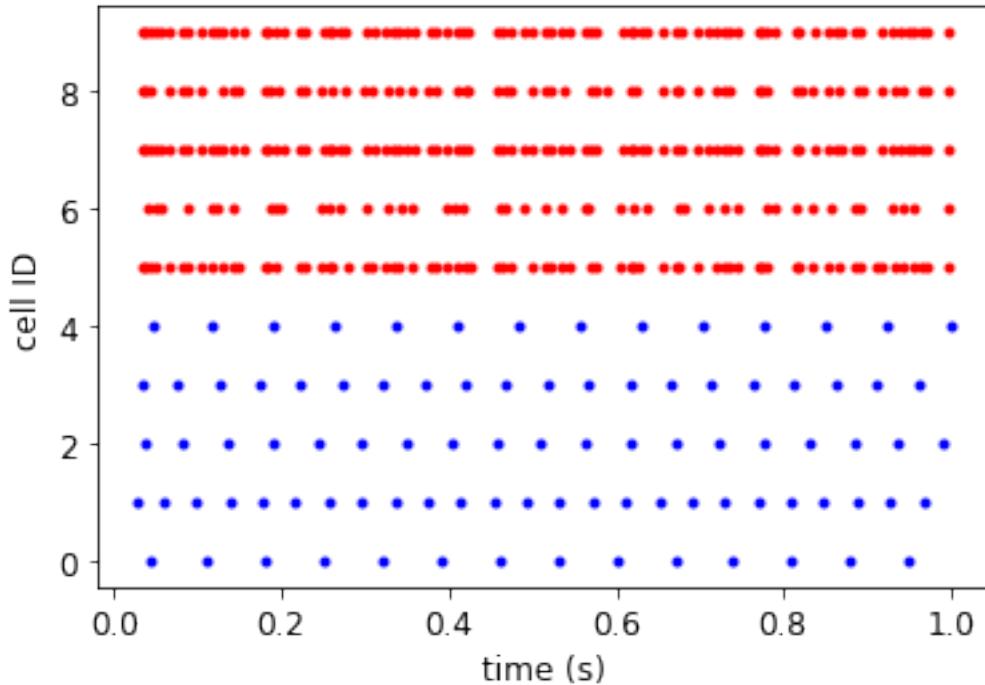
```
True
```

3.4.5 Plot the recorded data

```
# Load the data from the file and plot the spike times  
# using the pynml generate_plot utility function.  
data_array_0 = np.loadtxt("%s.0.spikes.dat" % simulation_id)  
data_array_1 = np.loadtxt("%s.1.spikes.dat" % simulation_id)  
times_0 = data_array_0[:,1]  
times_1 = data_array_1[:,1]  
ids_0 = data_array_0[:,0]  
ids_1 = [id+size0 for id in data_array_1[:,0]]  
pynml.generate_plot(  
    [times_0,times_1], [ids_0,ids_1],  
    "Spike times", show_plot_already=False,  
    save_figure_to="%s-spikes.png" % simulation_id,  
    xaxis="time (s)", yaxis="cell ID",  
    colors=['b','r'],  
    linewidths=[0,0], markers=[.,.],  
)
```

```
pyNeuroML >>> Generating plot: Spike times  
pyNeuroML >>> Saved image to example_izhikevich2007network_sim-spikes.png of plot:  
↳ Spike times
```

```
<AxesSubplot:xlabel='time (s)', ylabel='cell ID'>
```



3.5 Simulating a single compartment Hodgkin-Huxley neuron

In this section we will model and simulate a Hodgkin-Huxley (HH) neuron ([HH52]). A Hodgkin-Huxley neuron includes Sodium (Na), Potassium (K), and leak ion channels. For further information on this neuron model, please see [here](#).

This plot, saved as `HH_single_compartment_example_sim-v.png` is generated using the following Python NeuroML script:

```
#!/usr/bin/env python3
"""
Create a network with a single HH cell, and simulate it.

File: hh-single-compartment.py

Copyright 2021 NeuroML contributors
Author: Ankur Sinha <sanjay DOT ankur AT gmail DOT com>
"""

import math
from neuroml import NeuroMLDocument
from neuroml import Cell
from neuroml import IonChannelHH
from neuroml import GateHHRates
from neuroml import BiophysicalProperties
from neuroml import MembraneProperties
from neuroml import ChannelDensity
from neuroml import HHRate
from neuroml import SpikeThresh
from neuroml import SpecificCapacitance
from neuroml import InitMembPotential
```

(continues on next page)

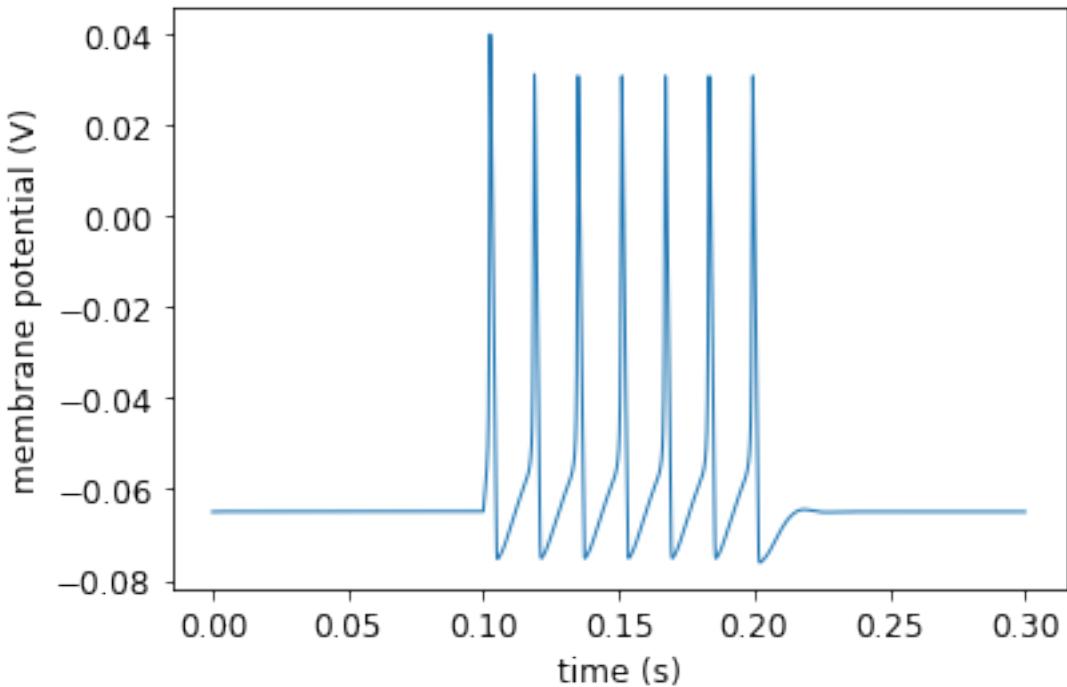


Fig. 3.6: Membrane potential of the simulated Hodgkin-Huxley neuron.

(continued from previous page)

```

from neuroml import IntracellularProperties
from neuroml import IncludeType
from neuroml import Resistivity
from neuroml import Morphology, Segment, Point3DWithDiam
from neuroml import Network, Population
from neuroml import PulseGenerator, ExplicitInput
import numpy as np
from pyneuroml import pynml
from pyneuroml.lemn import LEMSSimulation

def main():
    """Main function

    Include the NeuroML model into a LEMS simulation file, run it, plot some
    data.
    """
    # Simulation bits
    sim_id = "HH_single_compartment_example_sim"
    simulation = LEMSSimulation(sim_id=sim_id, duration=300, dt=0.01, simulation_
    seed=123)
    # Include the NeuroML model file
    simulation.include_neuroml2_file(create_network())
    # Assign target for the simulation
    simulation.assign_simulation_target("single_hh_cell_network")

    # Recording information from the simulation
    simulation.create_output_file(id="output0", file_name=sim_id + ".dat")

```

(continues on next page)

(continued from previous page)

```

simulation.add_column_to_output_file("output0", column_id="pop0[0]/v", quantity=
→"pop0[0]/v")
    simulation.add_column_to_output_file("output0", column_id="pop0[0]/iChannels", ←
→quantity="pop0[0]/iChannels")
    simulation.add_column_to_output_file("output0", column_id="pop0[0]/na/iDensity", ←
→quantity="pop0[0]/hh_b_prop/membraneProperties/na_channels/iDensity/")
    simulation.add_column_to_output_file("output0", column_id="pop0[0]/k/iDensity", ←
→quantity="pop0[0]/hh_b_prop/membraneProperties/k_channels/iDensity/")

    # Save LEMS simulation to file
    sim_file = simulation.save_to_file()

    # Run the simulation using the default jNeuroML simulator
    pynml.run_lems_with_jneuroml(sim_file, max_memory="2G", nogui=True, plot=False)
    # Plot the data
    plot_data(sim_id)

def plot_data(sim_id):
    """Plot the sim data.

    Load the data from the file and plot the graph for the membrane potential
    using the pynml generate_plot utility function.

    :sim_id: ID of simulation

    """
    data_array = np.loadtxt(sim_id + ".dat")
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 1]], "Membrane potential", ←
→show_plot_already=False, save_figure_to=sim_id + "-v.png", xaxis="time (s)", yaxis=
→"membrane potential (V)")
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 2]], "channel current", ←
→show_plot_already=False, save_figure_to=sim_id + "-i.png", xaxis="time (s)", yaxis=
→"channel current (A)")
    pynml.generate_plot([data_array[:, 0], data_array[:, 0]], [data_array[:, 3], data_
→array[:, 4]], "current density", labels=["Na", "K"], show_plot_already=False, save_
→figure_to=sim_id + "-iden.png", xaxis="time (s)", yaxis="current density (A_per_m2)
→")

def create_na_channel():
    """Create the Na channel.

    This will create the Na channel and save it to a file.
    It will also validate this file.

    returns: name of the created file
    """
    na_channel = IonChannelHH(id="na_channel", notes="Sodium channel for HH cell", ←
→conductance="10pS", species="na")
    gate_m = GateHHRates(id="na_m", instances="3", notes="m gate for na channel")

    m_forward_rate = HHRate(type="HHExpLinearRate", rate="1per_ms", midpoint="-40mV", ←
→scale="10mV")
    m_reverse_rate = HHRate(type="HHExpRate", rate="4per_ms", midpoint="-65mV", scale=
→"-18mV")

```

(continues on next page)

(continued from previous page)

```

gate_m.forward_rate = m_forward_rate
gate_m.reverse_rate = m_reverse_rate
na_channel.gate_hh_rates.append(gate_m)

gate_h = GateHHRates(id="na_h", instances="1", notes="h gate for na channel")
h_forward_rate = HHRate(type="HHExpRate", rate="0.07per_ms", midpoint="-65mV",  

    ↪scale="-20mV")
h_reverse_rate = HHRate(type="HHsigmoidRate", rate="1per_ms", midpoint="-35mV",  

    ↪scale="10mV")
gate_h.forward_rate = h_forward_rate
gate_h.reverse_rate = h_reverse_rate
na_channel.gate_hh_rates.append(gate_h)

na_channel_doc = NeuroMLDocument(id="na_channel", notes="Na channel for HH neuron  

    ↪")
na_channel_fn = "HH_example_na_channel.nml"
na_channel_doc.ion_channel_hhs.append(na_channel)

pynml.write_neuroml2_file(nml2_doc=na_channel_doc, nml2_file_name=na_channel_fn,  

    ↪validate=True)

return na_channel_fn

def create_k_channel():
    """Create the K channel

    This will create the K channel and save it to a file.
    It will also validate this file.

    :returns: name of the K channel file
    """
    k_channel = IonChannelHH(id="k_channel", notes="Potassium channel for HH cell",  

        ↪conductance="10pS", species="k")
    gate_n = GateHHRates(id="k_n", instances="4", notes="n gate for k channel")
    n_forward_rate = HHRate(type="HHExpLinearRate", rate="0.1per_ms", midpoint="-55mV  

        ↪", scale="10mV")
    n_reverse_rate = HHRate(type="HHExpRate", rate="0.125per_ms", midpoint="-65mV",  

        ↪scale="-80mV")
    gate_n.forward_rate = n_forward_rate
    gate_n.reverse_rate = n_reverse_rate
    k_channel.gate_hh_rates.append(gate_n)

    k_channel_doc = NeuroMLDocument(id="k_channel", notes="k channel for HH neuron")
    k_channel_fn = "HH_example_k_channel.nml"
    k_channel_doc.ion_channel_hhs.append(k_channel)

    pynml.write_neuroml2_file(nml2_doc=k_channel_doc, nml2_file_name=k_channel_fn,  

        ↪validate=True)

    return k_channel_fn

def create_leak_channel():
    """Create a leak channel

```

(continues on next page)

(continued from previous page)

```

This will create the leak channel and save it to a file.
It will also validate this file.

:returns: name of leak channel nml file
"""
leak_channel = IonChannelHH(id="leak_channel", conductance="10pS", notes="Leak_
conductance")
leak_channel_doc = NeuroMLDocument(id="leak_channel", notes="leak channel for HH_
neuron")
leak_channel_fn = "HH_example_leak_channel.nml"
leak_channel_doc.ion_channel_hhs.append(leak_channel)

pynml.write_neuroml2_file(nml2_doc=leak_channel_doc, nml2_file_name=leak_channel_
fn, validate=True)

return leak_channel_fn

def create_cell():
    """Create the cell.

    :returns: name of the cell nml file
    """
    # Create the nml file and add the ion channels
    hh_cell_doc = NeuroMLDocument(id="cell", notes="HH cell")
    hh_cell_fn = "HH_example_cell.nml"
    hh_cell_doc.includes.append(IncludeType(href=create_na_channel()))
    hh_cell_doc.includes.append(IncludeType(href=create_k_channel()))
    hh_cell_doc.includes.append(IncludeType(href=create_leak_channel()))

    # Define a cell
    hh_cell = Cell(id="hh_cell", notes="A single compartment HH cell")

    # Define its biophysical properties
    bio_prop = BiophysicalProperties(id="hh_b_prop")
    # notes="Biophysical properties for HH cell"

    # Membrane properties are a type of biophysical properties
    mem_prop = MembraneProperties()
    # Add membrane properties to the biophysical properties
    bio_prop.membrane_properties = mem_prop

    # Append to cell
    hh_cell.biophysical_properties = bio_prop

    # Channel density for Na channel
    na_channel_density = ChannelDensity(id="na_channels", cond_density="120.0 mS_per_
    cm2", erev="50.0 mV", ion="na", ion_channel="na_channel")
    mem_prop.channel_densities.append(na_channel_density)

    # Channel density for k channel
    k_channel_density = ChannelDensity(id="k_channels", cond_density="360 S_per_m2",_
    erev="-77mV", ion="k", ion_channel="k_channel")
    mem_prop.channel_densities.append(k_channel_density)

    # Leak channel

```

(continues on next page)

(continued from previous page)

```

        leak_channel_density = ChannelDensity(id="leak_channels", cond_density="3.0 S_per_
        ↪m2", erev="-54.3mV", ion="non_specific", ion_channel="leak_channel")
        mem_prop.channel_densities.append(leak_channel_density)

    # Other membrane properties
    mem_prop.spike_threshes.append(SpikeThresh(value="-20mV"))
    mem_prop.specific_capacitances.append(SpecificCapacitance(value="1.0 uF_per_cm2"))
    mem_prop.init_memb_potentials.append(InitMembPotential(value="-65mV"))

    intra_prop = IntracellularProperties()
    intra_prop.resistivities.append(Resistivity(value="0.03 kohm_cm"))

    # Add to biological properties
    bio_prop.intracellular_properties = intra_prop

    # Morphology
    morph = Morphology(id="hh_cell_morph")
    # notes="Simple morphology for the HH cell"
    seg = Segment(id="0", name="soma", notes="Soma segment")
    # We want a diameter such that area is 1000 micro meter^2
    # surface area of a sphere is 4pi r^2 = 4pi diam^2
    diam = math.sqrt(1000 / math.pi)
    proximal = distal = Point3DWithDiam(x="0", y="0", z="0", diameter=str(diam))
    seg.proximal = proximal
    seg.distal = distal
    morph.segments.append(seg)
    hh_cell.morphology = morph

    hh_cell_doc.cells.append(hh_cell)
    pynml.write_neuroml2_file(nml2_doc=hh_cell_doc, nml2_file_name=hh_cell_fn,_
    ↪validate=True)
    return hh_cell_fn

def create_network():
    """Create the network

    :returns: name of network nml file
    """
    net_doc = NeuroMLDocument(id="network",
                               notes="HH cell network")
    net_doc_fn = "HH_example_net.nml"
    net_doc.includes.append(IncludeType(href=create_cell()))
    # Create a population: convenient to create many cells of the same type
    pop = Population(id="pop0", notes="A population for our cell", component="hh_cell"
    ↪", size=1)
    # Input
    pulsegen = PulseGenerator(id="pg", notes="Simple pulse generator", delay="100ms",_
    ↪duration="100ms", amplitude="0.08nA")

    exp_input = ExplicitInput(target="pop0[0]", input="pg")

    net = Network(id="single_hh_cell_network", note="A network with a single_
    ↪population")
    net_doc.pulse_generators.append(pulsegen)
    net.explicit_inputs.append(exp_input)

```

(continues on next page)

(continued from previous page)

```

net.populations.append(pop)
net_doc.networks.append(net)

pynml.write_neuroml2_file(nml2_doc=net_doc, nml2_file_name=net_doc_fn, ↴
validate=True)
return net_doc_fn

if __name__ == "__main__":
    main()

```

3.5.1 Declaring the model in NeuroML

Similar to previous examples, we will first declare the model, visualise it, and then simulate it. The HH neuron model is more complex than the [Izhikevich neuron model](#) we have seen so far. For example, it includes voltage-gated ion channels. We will first implement these ion channels in NeuroML, then add them to a cell. We will then create a network of one cell which will stimulate with external input to record the membrane potential.

As you can also see in the script, since this is a slightly more complex model, we have modularised our code into different functions that carry out different tasks. Let us now step through the script in a bottom-up fashion. We start with the ion channels and build the network simulation.

Declaring ion channels

Note: you might not need to define your ion channels in Python every time...

In this example, all parts of the model, including the ion channels, are defined from scratch in Python and then NeuroML files in XML are generated and saved. For many modelling projects however, ion channel XML files will be reused from other models, and can just be included in the cells that use them with: <include href="my_channel.nml"/>. See [here](#) for tips on where to find ion channel models in NeuroML.

Let us look at the definition of the Sodium (Na) channel in NeuroML:

```

def create_na_channel():
    """Create the Na channel.

    This will create the Na channel and save it to a file.
    It will also validate this file.

    returns: name of the created file
    """
    na_channel = IonChannelHH(id="na_channel", notes="Sodium channel for HH cell", ↴
conductance="10pS", species="na")
    gate_m = GateHHRates(id="na_m", instances="3", notes="m gate for na channel")

    m_forward_rate = HHRate(type="HHExpLinearRate", rate="1per_ms", midpoint="-40mV", ↴
scale="10mV")
    m_reverse_rate = HHRate(type="HHExpRate", rate="4per_ms", midpoint="-65mV", scale= ↴
"-18mV")
    gate_m.forward_rate = m_forward_rate
    gate_m.reverse_rate = m_reverse_rate

```

(continues on next page)

(continued from previous page)

```

na_channel.gate_hh_rates.append(gate_m)

gate_h = GateHHRates(id="na_h", instances="1", notes="h gate for na channel")
h_forward_rate = HHRate(type="HHExpRate", rate="0.07per_ms", midpoint="-65mV", ↴
scale="-20mV")
h_reverse_rate = HHRate(type="HHSigmoidRate", rate="1per_ms", midpoint="-35mV", ↴
scale="10mV")
gate_h.forward_rate = h_forward_rate
gate_h.reverse_rate = h_reverse_rate
na_channel.gate_hh_rates.append(gate_h)

na_channel_doc = NeuroMLDocument(id="na_channel", notes="Na channel for HH neuron"
")
na_channel_fn = "HH_example_na_channel.nml"
na_channel_doc.ion_channel_hhs.append(na_channel)

pynml.write_neuroml2_file(nml2_doc=na_channel_doc, nml2_file_name=na_channel_fn, ↴
validate=True)

return na_channel_fn

```

Here, we define the two gates, m and h, with their forward and reverse rates and add them to the channel. Next, we create a NeuroML document and save this channel (only this channel that we've just defined) to a NeuroML file and validate it. So we now have our Na channel defined in a separate NeuroML file that can be used in multiple models and shared:

```

<neuroml xmlns="http://www.neuroml.org/schema/neuroml2" xmlns:xs="http://www.w3.org/
2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ↴
xsi:schemaLocation="http://www.neuroml.org/schema/neuroml2 https://raw.github.com/
NeuroML/NeuroML2/development/Schemas/NeuroML2/NeuroML_v2.3.xsd" id="na_channel">
    <notes>Na channel for HH neuron</notes>
    <ionChannelHH id="na_channel" species="na" conductance="10pS">
        <notes>Sodium channel for HH cell</notes>
        <gateHHRates id="na_m" instances="3">
            <notes>m gate for na channel</notes>
            <forwardRate type="HHExpLinearRate" rate="1per_ms" midpoint="-40mV" scale=
"10mV"/>
            <reverseRate type="HHExpRate" rate="4per_ms" midpoint="-65mV" scale="-18mV
"/>
        </gateHHRates>
        <gateHHRates id="na_h" instances="1">
            <notes>h gate for na channel</notes>
            <forwardRate type="HHExpRate" rate="0.07per_ms" midpoint="-65mV" scale="-
20mV"/>
            <reverseRate type="HHSigmoidRate" rate="1per_ms" midpoint="-35mV" scale=
"10mV"/>
        </gateHHRates>
    </ionChannelHH>
</neuroml>

```

The various rate equations (*HHExpLinearRate*, *HHExpRate*, *HHSigmoidRate*) that can be used in the gate (here *gateHHRates*, but other forms such as *gateHHtauInf* and *gateHHInstantaneous* can be used) are defined in the NeuroML schema.

Also note that since we'll want to *include* this file in other NeuroML files, we make the function return the name of the file. This is an implementation detail, and there are other ways of doing this too. We could have hard-coded this in all our functions or defined it as a global variable in the script for example. If we were using object-oriented programming, we could have created a class and stored this information as a class or object variable.

The K and leak channels are defined in a similar way:

```
def create_k_channel():
    """Create the K channel

    This will create the K channel and save it to a file.
    It will also validate this file.

    :returns: name of the K channel file
    """
    k_channel = IonChannelHH(id="k_channel", notes="Potassium channel for HH cell",
    conductance="10pS", species="k")
    gate_n = GateHHRates(id="k_n", instances="4", notes="n gate for k channel")
    n_forward_rate = HHRate(type="HHExpLinearRate", rate="0.1per_ms", midpoint="-55mV",
    scale="10mV")
    n_reverse_rate = HHRate(type="HHExpRate", rate="0.125per_ms", midpoint="-65mV",
    scale="-80mV")
    gate_n.forward_rate = n_forward_rate
    gate_n.reverse_rate = n_reverse_rate
    k_channel.gate_hh_rates.append(gate_n)

    k_channel_doc = NeuroMLDocument(id="k_channel", notes="k channel for HH neuron")
    k_channel_fn = "HH_example_k_channel.nml"
    k_channel_doc.ion_channel_hhs.append(k_channel)

    pynml.write_neuroml2_file(nml2_doc=k_channel_doc, nml2_file_name=k_channel_fn,
    validate=True)

    return k_channel_fn


def create_leak_channel():
    """Create a leak channel

    This will create the leak channel and save it to a file.
    It will also validate this file.

    :returns: name of leak channel nml file
    """
    leak_channel = IonChannelHH(id="leak_channel", conductance="10pS", notes="Leak",
    conductance="")
    leak_channel_doc = NeuroMLDocument(id="leak_channel", notes="leak channel for HH",
    neuron="")
    leak_channel_fn = "HH_example_leak_channel.nml"
    leak_channel_doc.ion_channel_hhs.append(leak_channel)

    pynml.write_neuroml2_file(nml2_doc=leak_channel_doc, nml2_file_name=leak_channel_fn,
    validate=True)

    return leak_channel_fn
```

They are also saved in their own NeuroML files, which have also been validated. The file for the K channel:

```
<neuroml xmlns="http://www.neuroml.org/schema/neuroml2" xmlns:xs="http://www.w3.org/
    2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.neuroml.org/schema/neuroml2 https://raw.github.com/
    NeuroML/NeuroML2/development/Schemas/NeuroML2/NeuroML_v2.3.xsd" id="k_channel">
```

(continues on next page)

(continued from previous page)

```
<notes>k channel for HH neuron</notes>
<ionChannelHH id="k_channel" species="k" conductance="10pS">
    <notes>Potassium channel for HH cell</notes>
    <gateHHrates id="k_n" instances="4">
        <notes>n gate for k channel</notes>
        <forwardRate type="HHExpLinearRate" rate="0.1per_ms" midpoint="-55mV" scale="10mV"/>
        <reverseRate type="HHExpRate" rate="0.125per_ms" midpoint="-65mV" scale="-80mV"/>
    </gateHHrates>
</ionChannelHH>
</neuroml>
```

For the leak channel:

```
<neuroml xmlns="http://www.neuroml.org/schema/neuroml2" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.neuroml.org/schema/neuroml2 https://raw.github.com/NeuroML/NeuroML2/development/Schemas/NeuroML2/NeuroML_v2.3.xsd" id="leak_channel">
    <notes>leak channel for HH neuron</notes>
    <ionChannelHH id="leak_channel" conductance="10pS">
        <notes>Leak conductance</notes>
    </ionChannelHH>
</neuroml>
```

Declaring the cell

Now that we have declared our ion channels, we can start constructing our *cell* in a different function.

```
def create_cell():
    """Create the cell.

    :returns: name of the cell nml file
    """
    # Create the nml file and add the ion channels
    hh_cell_doc = NeuroMLDocument(id="cell", notes="HH cell")
    hh_cell_fn = "HH_example_cell.nml"
    hh_cell_doc.includes.append(IncludeType(href=create_na_channel()))
    hh_cell_doc.includes.append(IncludeType(href=create_k_channel()))
    hh_cell_doc.includes.append(IncludeType(href=create_leak_channel()))

    # Define a cell
    hh_cell = Cell(id="hh_cell", notes="A single compartment HH cell")

    # Define its biophysical properties
    bio_prop = BiophysicalProperties(id="hh_b_prop")
    # notes="Biophysical properties for HH cell"

    # Membrane properties are a type of biophysical properties
    mem_prop = MembraneProperties()
    # Add membrane properties to the biophysical properties
    bio_prop.membrane_properties = mem_prop

    # Append to cell
    hh_cell.biophysical_properties = bio_prop
```

(continues on next page)

(continued from previous page)

```

hh_cell.biophysical_properties = bio_prop

# Channel density for Na channel
na_channel_density = ChannelDensity(id="na_channels", cond_density="120.0 mS_per_
cm2", erev="50.0 mV", ion="na", ion_channel="na_channel")
mem_prop.channel_densities.append(na_channel_density)

# Channel density for k channel
k_channel_density = ChannelDensity(id="k_channels", cond_density="360 S_per_m2",_
erev="-77mV", ion="k", ion_channel="k_channel")
mem_prop.channel_densities.append(k_channel_density)

# Leak channel
leak_channel_density = ChannelDensity(id="leak_channels", cond_density="3.0 S_per_
m2", erev="-54.3mV", ion="non_specific", ion_channel="leak_channel")
mem_prop.channel_densities.append(leak_channel_density)

# Other membrane properties
mem_prop.spike_threshes.append(SpikeThresh(value="-20mV"))
mem_prop.specific_capacitances.append(SpecificCapacitance(value="1.0 uF_per_cm2"))
mem_prop.init_memb_potentials.append(InitMembPotential(value="-65mV"))

intra_prop = IntracellularProperties()
intra_prop.resistivities.append(Resistivity(value="0.03 kohm_cm"))

# Add to biological properties
bio_prop.intracellular_properties = intra_prop

# Morphology
morph = Morphology(id="hh_cell_morph")
# notes="Simple morphology for the HH cell"
seg = Segment(id="0", name="soma", notes="Soma segment")
# We want a diameter such that area is 1000 micro meter^2
# surface area of a sphere is 4pi r^2 = 4pi diam^2
diam = math.sqrt(1000 / math.pi)
proximal = distal = Point3DWithDiam(x="0", y="0", z="0", diameter=str(diam))
seg.proximal = proximal
seg.distal = distal
morph.segments.append(seg)
hh_cell.morphology = morph

hh_cell_doc.cells.append(hh_cell)
pynml.write_neuroml2_file(nml2_doc=hh_cell_doc, nml2_file_name=hh_cell_fn,_
validate=True)
return hh_cell_fn

```

Let us walk through this function:

```

hh_cell_doc = NeuroMLDocument(id="cell", notes="HH cell")
hh_cell_fn = "HH_example_cell.nml"
hh_cell_doc.includes.append(IncludeType(href=create_na_channel()))
hh_cell_doc.includes.append(IncludeType(href=create_k_channel()))
hh_cell_doc.includes.append(IncludeType(href=create_leak_channel()))

# Define a cell

```

(continues on next page)

(continued from previous page)

```
hh_cell = Cell(id="hh_cell", notes="A single compartment HH cell")
```

First, before we do anything else, we create a new NeuroML document that we will use to save this cell. Now, since the ion-channels were created in other files, we need to make this document aware of their declarations. To do this, we *include* the other files into this one using the `IncludeType` construct. Each document we want to include gets appended to the list of `includes` for the document.

Now we can proceed to building our cell using the *Cell NeuroML component type*. As the schema document shows, a `Cell` component has two children: *morphology*, and *biophysical properties*. Let us first look at setting up the biophysical properties:

```
# Define its biophysical properties
bio_prop = BiophysicalProperties(id="hh_b_prop")
# notes="Biophysical properties for HH cell"

# Membrane properties are a type of biophysical properties
mem_prop = MembraneProperties()
# Add membrane properties to the biophysical properties
bio_prop.membrane_properties = mem_prop

# Append to cell
hh_cell.biophysical_properties = bio_prop

# Channel density for Na channel
na_channel_density = ChannelDensity(id="na_channels", cond_density="120.0 mS_per_
cm2", erev="50.0 mV", ion="na", ion_channel="na_channel")
mem_prop.channel_densities.append(na_channel_density)

# Channel density for k channel
k_channel_density = ChannelDensity(id="k_channels", cond_density="360 S_per_m2",_
erev="-77mV", ion="k", ion_channel="k_channel")
mem_prop.channel_densities.append(k_channel_density)

# Leak channel
leak_channel_density = ChannelDensity(id="leak_channels", cond_density="3.0 S_per_-
m2", erev="-54.3mV", ion="non_specific", ion_channel="leak_channel")
mem_prop.channel_densities.append(leak_channel_density)

# Other membrane properties
mem_prop.spike_threshes.append(SpikeThresh(value="-20mV"))
mem_prop.specific_capacitances.append(SpecificCapacitance(value="1.0 uF_per_cm2"))
mem_prop.init_memb_potentials.append(InitMembPotential(value="-65mV"))

intra_prop = IntracellularProperties()
intra_prop.resistivities.append(Resistivity(value="0.03 kohm_cm"))

# Add to biological properties
bio_prop.intracellular_properties = intra_prop
```

Biophysical properties are themselves split into two:

- the *membrane properties*
- the *intracellular properties*

Let us look at membrane properties first. The *schema* shows that membrane properties has two *child* elements:

- *initMembPotential*

- *spikeThresh*

and three *children* elements:

- *specificCapacitances*
- *populations*
- *channelDensities*

Child elements vs Children elements

When an element specifies a **Child** subelement, it will only have one of these present (it could have zero). **Children** explicitly says that there can be zero, one or many subelements.

So, we start with the ion-channels which are distributed along the membrane with some density. We create new *ChannelDensity* objects for each of our defined ion-channels (Na, K, leak) and append these to the list of channel densities in the membrane properties. For example, for the Na channels:

```
# Channel density for Na channel
na_channel_density = ChannelDensity(id="na_channels", cond_density="120.0 mS_per_
cm2", erev="50.0 mV", ion="na", ion_channel="na_channel")
mem_prop.channel_densities.append(na_channel_density)
```

and similarly for the K and leak channels. Next, we add the other child and children elements: the *specificCapacitance*, the *spikeThreshold*, the *initMembPotential*. This completes the membrane properties. Similarly we add the intracellular properties next: *Resistivity*.

Next, we add the *Morphology* related information for our cell. Here, we are only creating a single compartment cell with only one segment. We will look into multi-compartment cells with more segments in later examples:

```
# Morphology
morph = Morphology(id="hh_cell_morph")
# notes="Simple morphology for the HH cell"
seg = Segment(id="0", name="soma", notes="Soma segment")
# We want a diameter such that area is 1000 micro meter^2
# surface area of a sphere is 4pi r^2 = 4pi diam^2
diam = math.sqrt(1000 / math.pi)
proximal = distal = Point3DWithDiam(x="0", y="0", z="0", diameter=str(diam))
seg.proximal = proximal
seg.distal = distal
morph.segments.append(seg)
hh_cell.morphology = morph
```

A *segment* has *proximal* and *distal* child elements which describe the extent of the segment. These are described using a *Point3DWithDiam* object.

This completes our cell. We add it to our NeuroML document, and save (and validate) it. The resulting NeuroML file is:

```
<neuroml xmlns="http://www.neuroml.org/schema/neuroml2" xmlns:xs="http://www.w3.org/
2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"_
xsi:schemaLocation="http://www.neuroml.org/schema/neuroml2 https://raw.github.com/
NeuroML/NeuroML2/development/Schemas/NeuroML2/NeuroML_v2.3.xsd" id="cell">
<notes>HH cell</notes>
<include href="HH_example_na_channel.nml"/>
<include href="HH_example_k_channel.nml"/>
<include href="HH_example_leak_channel.nml"/>
```

(continues on next page)

(continued from previous page)

```

<cell id="hh_cell">
    <notes>A single compartment HH cell</notes>
    <morphology id="hh_cell_morph">
        <segment id="0" name="soma">
            <proximal x="0.0" y="0.0" z="0.0" diameter="17.841241161527712"/>
            <distal x="0.0" y="0.0" z="0.0" diameter="17.841241161527712"/>
        </segment>
    </morphology>
    <biophysicalProperties id="hh_b_prop">
        <membraneProperties>
            <channelDensity id="na_channels" ionChannel="na_channel" condDensity=
            ↵ "120.0 mS_per_cm2" erev="50.0 mV" ion="na"/>
            <channelDensity id="k_channels" ionChannel="k_channel" condDensity=
            ↵ "360 S_per_m2" erev="-77mV" ion="k"/>
            <channelDensity id="leak_channels" ionChannel="leak_channel"_
            ↵ condDensity="3.0 S_per_m2" erev="-54.3mV" ion="non_specific"/>
            <spikeThresh value="-20mV"/>
            <specificCapacitance value="1.0 uF_per_cm2"/>
            <initMembPotential value="-65mV"/>
        </membraneProperties>
        <intracellularProperties>
            <resistivity value="0.03 kohm_cm"/>
        </intracellularProperties>
    </biophysicalProperties>
</cell>
</neuroml>

```

We now have our cell defined in a separate NeuroML file, that can be re-used and shared.

Declaring the network

We now use our cell in a network. A *network in NeuroML* has multiple children elements: *populations*, *projections*, *inputLists* and so on. Here we are going to only create a network with one cell, and an *explicit input* to the cell:

```

def create_network():
    """Create the network

    :returns: name of network nml file
    """
    net_doc = NeuroMLDocument(id="network",
                               notes="HH cell network")
    net_doc_fn = "HH_example_net.nml"
    net_doc.includes.append(IncludeType(href=create_cell()))
    # Create a population: convenient to create many cells of the same type
    pop = Population(id="pop0", notes="A population for our cell", component="hh_cell",
                     size=1)
    # Input
    pulsegen = PulseGenerator(id="pg", notes="Simple pulse generator", delay="100ms",
                               duration="100ms", amplitude="0.08nA")

    exp_input = ExplicitInput(target="pop0[0]", input="pg")

    net = Network(id="single_hh_cell_network", note="A network with a single_
    ↵ population")
    net_doc.pulse_generators.append(pulsegen)

```

(continues on next page)

(continued from previous page)

```

net.explicit_inputs.append(exp_input)
net.populations.append(pop)
net_doc.networks.append(net)

pynml.write_neuroml2_file(nml2_doc=net_doc, nml2_file_name=net_doc_fn, validate=True)
return net_doc_fn

```

We start in the same way, by creating a new NeuroML document and including our cell file into it. We then create a *population* comprising of a single cell. We create a *pulse generator* as an *explicit input*, which targets our population. Note that as the schema documentation for `ExplicitInput` notes, any current source (any component that *extends basePointCurrent*) can be used as an `ExplicitInput`.

We add all of these to the *network* and save (and validate) our network file. The NeuroML file generated is below:

```

<neuroml xmlns="http://www.neuroml.org/schema/neuroml2" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.neuroml.org/schema/neuroml2 https://raw.github.com/NeuroML/NeuroML2/development/Schemas/NeuroML2/NeuroML_v2.3.xsd" id="network">
    <notes>HH cell network</notes>
    <include href="HH_example_cell.nml"/>
    <pulseGenerator id="pg" delay="100ms" duration="100ms" amplitude="0.08nA">
        <notes>Simple pulse generator</notes>
    </pulseGenerator>
    <network id="single_hh_cell_network">
        <population id="pop0" component="hh_cell" size="1">
            <notes>A population for our cell</notes>
        </population>
        <explicitInput target="pop0[0]" input="pg"/>
    </network>
</neuroml>

```

The generated NeuroML model

Before we look at simulating the model, we can inspect our model to check for correctness. All our NeuroML files were validated when they were created already, so we do not need to run this step again. However, if required, this can be easily done:

```
pynml -validate HH_*nml
```

Next, we can visualise our model using the information noted in the *visualising NeuroML models* page (including the `-v` verbose option for more information on the cell):

```

pynml-summary HH_example_net.nml -v
*****
* NeuroMLDocument: network
*
* IonChannelHH: ['k_channel', 'leak_channel', 'na_channel']
* PulseGenerator: ['pg']
*
* Cell: hh_cell
*     <Segment 0 / soma>

```

(continues on next page)

(continued from previous page)

```

*      Parent segment: None (root segment)
*      (0.0, 0.0, 0.0), diam 17.841241161527712um -> (0.0, 0.0, 0.0), diam 17.
*      ↳ 841241161527712um; seg length: 0.0 um
*      Surface area: 1000.0 um2, volume: 2973.5401935879518 um3
*      Total length of 1 segment: 0.0 um; total area: 1000.0 um2
*
*      Channel density: na_channels on all; conductance of 120.0 mS_per_cm2
*      ↳ through ion chan na_channel with ion na, erev: 50.0 mV
*      Channel is on <Segment/0/soma>, total conductance: 1200.0 S_per_m2 x 1e-09 m2
*      ↳ = 1.2000000000000002e-06 S (1200000.0000000002 pS)
*      Channel density: k_channels on all; conductance of 360 S_per_m2 through
*      ↳ ion chan k_channel with ion k, erev: -77mV
*      Channel is on <Segment/0/soma>, total conductance: 360.0 S_per_m2 x 1e-09 m2
*      ↳ = 3.600000000000005e-07 S (360000.0000000006 pS)
*      Channel density: leak_channels on all; conductance of 3.0 S_per_m2 through
*      ↳ ion chan leak_channel with ion non_specific, erev: -54.3mV
*      Channel is on <Segment/0/soma>, total conductance: 3.0 S_per_m2 x 1e-09 m2 =
*      ↳ 3.000000000000004e-09 S (3000.000000000005 pS)
*
*      Specific capacitance on all: 1.0 uF_per_cm2
*      Capacitance of <Segment/0/soma>, total capacitance: 0.01 F_per_m2 x 1e-09 m2 =
*      ↳ 1.000000000000001e-11 F (10.00000000000002 pF)
*
*      Network: single_hh_cell_network
*
*      1 cells in 1 populations
*      Population: pop0 with 1 components of type hh_cell
*
*      0 connections in 0 projections
*
*      0 inputs in 0 input lists
*
*      1 explicit inputs (outside of input lists)
*      Explicit Input of type pg to pop0(cell 0), destination: unspecified
*
*****

```

Since our model is a single compartment model with only one cell, it doesn't have any 3D structure to visualise. We can check the connectivity graph of the model:

```
pynml -graph 10 HH_example_net.nml
```

which will give us this figure:

Analysing channels

Finally, we can analyse the ion channels that we've declared using the pynml-channelanalysis utility:

```
pynml-channelanalysis HH_example_k_channel.nml
```

This generates graphs to show the behaviour of the channel:

Similarly, we can get these for the Na channel also:

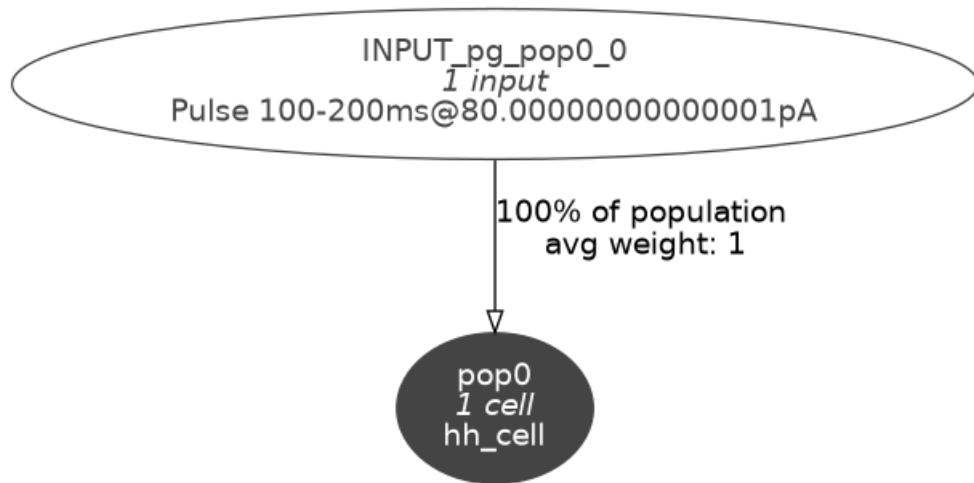


Fig. 3.7: Level 10 network graph generated by pynml

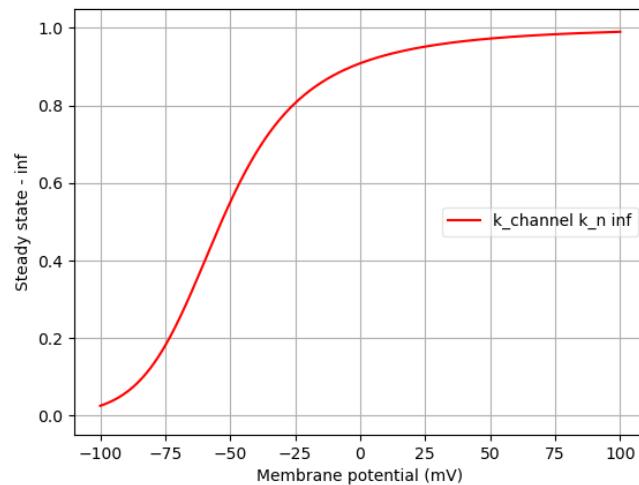


Fig. 3.8: Steady state behaviour of the K ion channel.

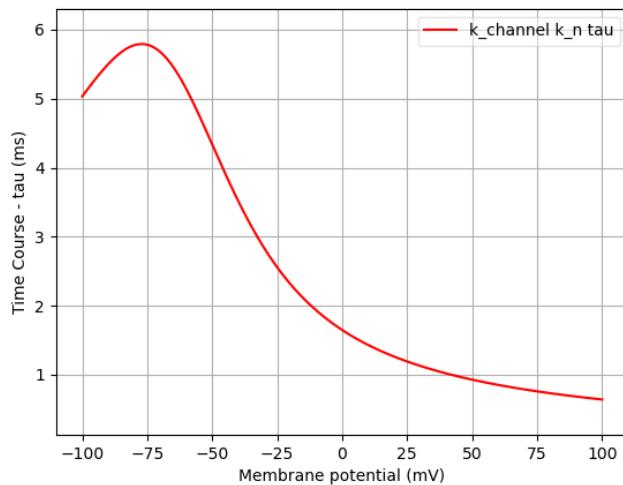


Fig. 3.9: Time course of the K ion channel.

```
pynml-channelanalysis HH_example_na_channel.nml
```

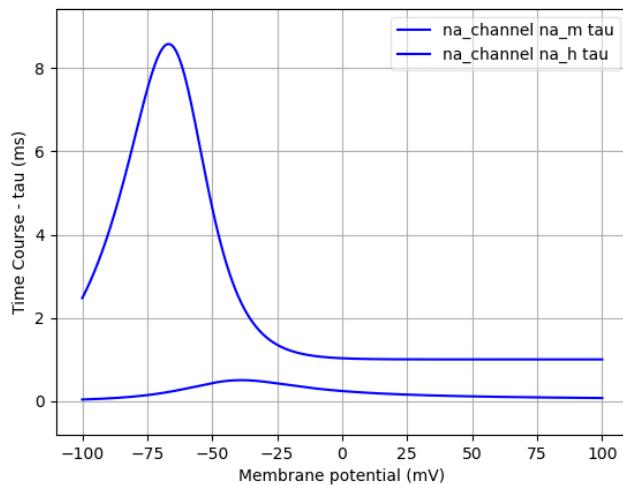


Fig. 3.10: Steady state behaviour of the Na ion channel.

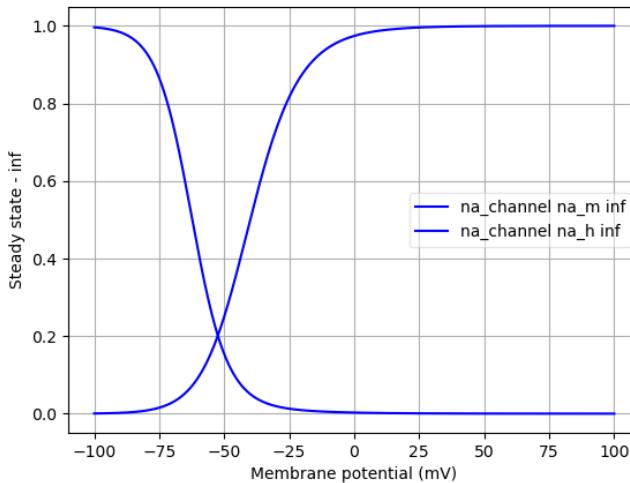


Fig. 3.11: Time course of the Na ion channel.

3.5.2 Simulating the model

Now that we have declared and inspected our network model and all its components, we can proceed to simulate it. We do this in the `main` function:

```
def main():
    """Main function

    Include the NeuroML model into a LEMS simulation file, run it, plot some
    data.
    """
    # Simulation bits
    sim_id = "HH_single_compartment_example_sim"
    simulation = LEMSSimulation(sim_id=sim_id, duration=300, dt=0.01, simulation_
    ↪seed=123)
    # Include the NeuroML model file
    simulation.include_neuroml2_file(create_network())
    # Assign target for the simulation
    simulation.assign_simulation_target("single_hh_cell_network")

    # Recording information from the simulation
    simulation.create_output_file(id="output0", file_name=sim_id + ".dat")
    simulation.add_column_to_output_file("output0", column_id="pop0[0]/v", quantity=
    ↪"pop0[0]/v")
    simulation.add_column_to_output_file("output0", column_id="pop0[0]/iChannels", ↪
    ↪quantity="pop0[0]/iChannels")
    simulation.add_column_to_output_file("output0", column_id="pop0[0]/na/iDensity", ↪
    ↪quantity="pop0[0]/hh_b_prop/membraneProperties/na_channels/iDensity/")
    simulation.add_column_to_output_file("output0", column_id="pop0[0]/k/iDensity", ↪
    ↪quantity="pop0[0]/hh_b_prop/membraneProperties/k_channels/iDensity/")

    # Save LEMS simulation to file
    sim_file = simulation.save_to_file()
```

(continues on next page)

(continued from previous page)

```
# Run the simulation using the default jNeuroML simulator
pynml.run_lems_with_jneuroml(sim_file, max_memory="2G", nogui=True, plot=False)
# Plot the data
plot_data(sim_id)
```

Here we first create a `LEMSSimulation` instance and include our network NeuroML file in it. We must inform LEMS what the target of the simulation is. In our case, it's the id of our network, `single_hh_cell_network`:

```
sim_id = "HH_single_compartment_example_sim"
simulation = LEMSSimulation(sim_id=sim_id, duration=300, dt=0.01, simulation_
↪seed=123)
# Include the NeuroML model file
simulation.include_neuroml2_file(create_network())
# Assign target for the simulation
simulation.assign_simulation_target("single_hh_cell_network")
```

We also want to record some information, so we create an output file first with an `id` of `output0`:

```
simulation.create_output_file(id="output0", file_name=sim_id + ".dat")
simulation.add_column_to_output_file("output0", column_id="pop0[0]/v", quantity=
↪"pop0[0]/v")
```

Now, we can record any quantity that is exposed by NeuroML (any `exposure`). For example, we add a column for the membrane potential `v` of the `cell` which would be the *0th* (and only) cell in our population `pop0`: `pop0[0]/v`. We can also record the current in the channels: `pop[0]/iChannels` We can also record the `current density` `iDensity` for the channels, so we also record these.

```
simulation.add_column_to_output_file("output0", column_id="pop0[0]/v", quantity=
↪"pop0[0]/v")
simulation.add_column_to_output_file("output0", column_id="pop0[0]/iChannels",_
↪quantity="pop0[0]/iChannels")
simulation.add_column_to_output_file("output0", column_id="pop0[0]/na/iDensity",_
↪quantity="pop0[0]/hh_b_prop/membraneProperties/na_channels/iDensity/")
simulation.add_column_to_output_file("output0", column_id="pop0[0]/k/iDensity",_
↪quantity="pop0[0]/hh_b_prop/membraneProperties/k_channels/iDensity/")
```

We then save the LEMS simulation file, run our simulation with the default `jNeuroML` simulator.

3.5.3 Plotting the recorded variables

To plot the variables that we recorded, we read the data and use the `generate_plot` utility function:

```
def plot_data(sim_id):
    """Plot the sim data.

    Load the data from the file and plot the graph for the membrane potential
    using the pynml generate_plot utility function.

    :sim_id: ID of simulation

    """
    data_array = np.loadtxt(sim_id + ".dat")
```

(continues on next page)

(continued from previous page)

```

pynml.generate_plot([data_array[:, 0]], [data_array[:, 1]], "Membrane potential",  

↳ show_plot_already=False, save_figure_to=sim_id + "-v.png", xaxis="time (s)", yaxis=  

↳ "membrane potential (V)")  

pynml.generate_plot([data_array[:, 0]], [data_array[:, 2]], "channel current",  

↳ show_plot_already=False, save_figure_to=sim_id + "-i.png", xaxis="time (s)", yaxis=  

↳ "channel current (A)")  

pynml.generate_plot([data_array[:, 0], data_array[:, 0]], [data_array[:, 3], data_  

↳ array[:, 4]], "current density", labels=["Na", "K"], show_plot_already=False, save_  

↳ figure_to=sim_id + "-iden.png", xaxis="time (s)", yaxis="current density (A_per_m2)  

↳ ")

```

This generates the following graphs:

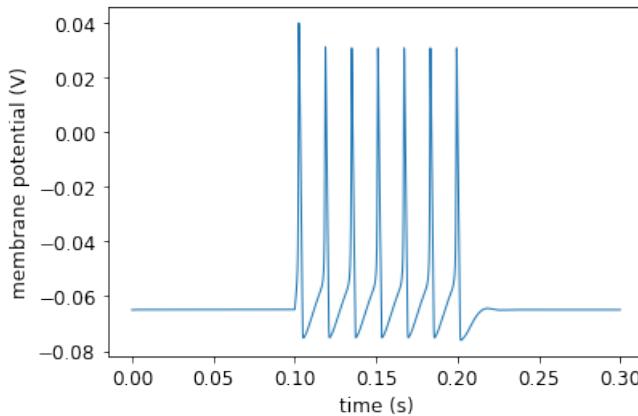


Fig. 3.12: Membrane potential

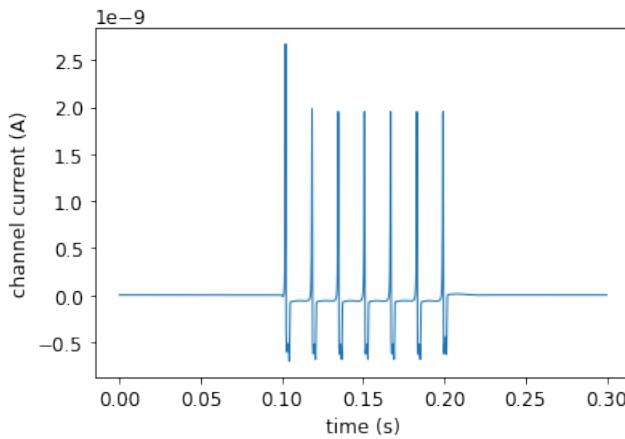


Fig. 3.13: Channel current.

This concludes our third example. Here we have seen how to create, simulate, record, and visualise a single compartment Hodgkin-Huxley neuron. In the next section, you will find an interactive notebook where you can play with this example.

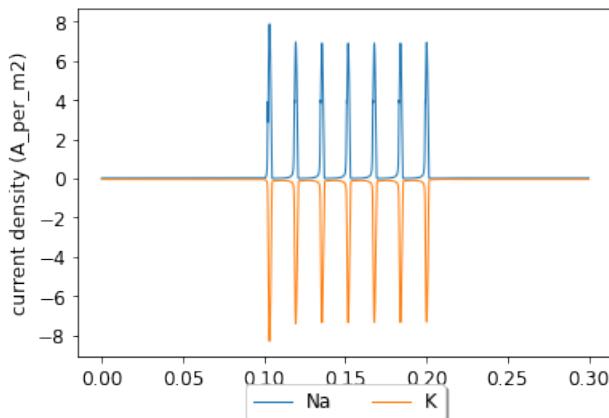


Fig. 3.14: Channel current densities

3.6 Interactive single compartment HH example

To run this interactive Jupyter Notebook, please click on the rocket icon in the top panel. For more information, please see [how to use this documentation](#). Please uncomment the line below if you use the Google Colab. (It does not include these packages by default).

```
#%pip install pyneuroml neuromllite NEURON
```

```
import math
from neuroml import NeuroMLDocument
from neuroml import Cell
from neuroml import IonChannelHH
from neuroml import GateHHRates
from neuroml import BiophysicalProperties
from neuroml import MembraneProperties
from neuroml import ChannelDensity
from neuroml import HHRate
from neuroml import SpikeThresh
from neuroml import SpecificCapacitance
from neuroml import InitMembPotential
from neuroml import IntracellularProperties
from neuroml import IncludeType
from neuroml import Resistivity
from neuroml import Morphology, Segment, Point3DWithDiam
from neuroml import Network, Population
from neuroml import PulseGenerator, ExplicitInput
import numpy as np
from pyneuroml import pynml
from pyneuroml.lems import LEMSSimulation
```

```
ModuleNotFoundError                                     Traceback (most recent call last)
Input In [2], in <cell line: 2>()
      1 import math
----> 2 from neuroml import NeuroMLDocument
      3 from neuroml import Cell
```

(continues on next page)

(continued from previous page)

```
4 from neuroml import IonChannelHH

ModuleNotFoundError: No module named 'neuroml'
```

3.6.1 Declare the model

Create ion channels

```
def create_na_channel():
    """Create the Na channel.

    This will create the Na channel and save it to a file.
    It will also validate this file.

    returns: name of the created file
    """
    na_channel = IonChannelHH(id="na_channel", notes="Sodium channel for HH cell",
    conductance="10pS", species="na")
    gate_m = GateHHRates(id="na_m", instances="3", notes="m gate for na channel")

    m_forward_rate = HHRate(type="HHExpLinearRate", rate="1per_ms", midpoint="-40mV",
    scale="10mV")
    m_reverse_rate = HHRate(type="HHExpRate", rate="4per_ms", midpoint="-65mV", scale=
    "-18mV")
    gate_m.forward_rate = m_forward_rate
    gate_m.reverse_rate = m_reverse_rate
    na_channel.gate_hh_rates.append(gate_m)

    gate_h = GateHHRates(id="na_h", instances="1", notes="h gate for na channel")
    h_forward_rate = HHRate(type="HHExpRate", rate="0.07per_ms", midpoint="-65mV",
    scale="-20mV")
    h_reverse_rate = HHRate(type="HHSigmoidRate", rate="1per_ms", midpoint="-35mV",
    scale="10mV")
    gate_h.forward_rate = h_forward_rate
    gate_h.reverse_rate = h_reverse_rate
    na_channel.gate_hh_rates.append(gate_h)

    na_channel_doc = NeuroMLDocument(id="na_channel", notes="Na channel for HH neuron
    ")
    na_channel_fn = "HH_example_na_channel.nml"
    na_channel_doc.ion_channel_hhs.append(na_channel)

    pynml.write_neuroml2_file(nml2_doc=na_channel_doc, nml2_file_name=na_channel_fn,
    validate=True)

    return na_channel_fn
```

```
def create_k_channel():
    """Create the K channel

    This will create the K channel and save it to a file.
    It will also validate this file.
```

(continues on next page)

(continued from previous page)

```

:returns: name of the K channel file
"""
k_channel = IonChannelHH(id="k_channel", notes="Potassium channel for HH cell",_
conductance="10pS", species="k")
gate_n = GateHHRates(id="k_n", instances="4", notes="n gate for k channel")
n_forward_rate = HHRate(type="HHExpLinearRate", rate="0.1per_ms", midpoint="-55mV"
", scale="10mV")
n_reverse_rate = HHRate(type="HHExpRate", rate="0.125per_ms", midpoint="-65mV",_
scale="-80mV")
gate_n.forward_rate = n_forward_rate
gate_n.reverse_rate = n_reverse_rate
k_channel.gate_hh_rates.append(gate_n)

k_channel_doc = NeuroMLDocument(id="k_channel", notes="k channel for HH neuron")
k_channel_fn = "HH_example_k_channel.nml"
k_channel_doc.ion_channel_hhs.append(k_channel)

pynml.write_neuroml2_file(nml2_doc=k_channel_doc, nml2_file_name=k_channel_fn,_
validate=True)

return k_channel_fn

```

```

def create_leak_channel():
    """Create a leak channel

    This will create the leak channel and save it to a file.
    It will also validate this file.

:returns: name of leak channel nml file
"""
leak_channel = IonChannelHH(id="leak_channel", conductance="10pS", notes="Leak_"
conductance")
leak_channel_doc = NeuroMLDocument(id="leak_channel", notes="leak channel for HH_"
neuron")
leak_channel_fn = "HH_example_leak_channel.nml"
leak_channel_doc.ion_channel_hhs.append(leak_channel)

pynml.write_neuroml2_file(nml2_doc=leak_channel_doc, nml2_file_name=leak_channel_
fn, validate=True)

return leak_channel_fn

```

Create cell

```

def create_cell():
    """Create the cell.

:returns: name of the cell nml file
"""
# Create the nml file and add the ion channels
hh_cell_doc = NeuroMLDocument(id="cell", notes="HH cell")
hh_cell_fn = "HH_example_cell.nml"
hh_cell_doc.includes.append(IncludeType(href=create_na_channel()))

```

(continues on next page)

(continued from previous page)

```

hh_cell_doc.includes.append(IncludeType(href=create_k_channel()))
hh_cell_doc.includes.append(IncludeType(href=create_leak_channel()))

# Define a cell
hh_cell = Cell(id="hh_cell", notes="A single compartment HH cell")

# Define its biophysical properties
bio_prop = BiophysicalProperties(id="hh_b_prop")
# notes="Biophysical properties for HH cell"

# Membrane properties are a type of biophysical properties
mem_prop = MembraneProperties()
# Add membrane properties to the biophysical properties
bio_prop.membrane_properties = mem_prop

# Append to cell
hh_cell.biophysical_properties = bio_prop

# Channel density for Na channel
na_channel_density = ChannelDensity(id="na_channels", cond_density="120.0 mS_per_
˓cm2", erev="50.0 mV", ion="na", ion_channel="na_channel")
mem_prop.channel_densities.append(na_channel_density)

# Channel density for k channel
k_channel_density = ChannelDensity(id="k_channels", cond_density="360 S_per_m2",_
˓erev="-77mV", ion="k", ion_channel="k_channel")
mem_prop.channel_densities.append(k_channel_density)

# Leak channel
leak_channel_density = ChannelDensity(id="leak_channels", cond_density="3.0 S_per_-
˓m2", erev="-54.3mV", ion="non_specific", ion_channel="leak_channel")
mem_prop.channel_densities.append(leak_channel_density)

# Other membrane properties
mem_prop.spike_threshes.append(SpikeThresh(value="-20mV"))
mem_prop.specific_capacitances.append(SpecificCapacitance(value="1.0 uF_per_cm2"))
mem_prop.init_memb_potentials.append(InitMembPotential(value="-65mV"))

intra_prop = IntracellularProperties()
intra_prop.resistivities.append(Resistivity(value="0.03 kohm_cm"))

# Add to biological properties
bio_prop.intracellular_properties = intra_prop

# Morphology
morph = Morphology(id="hh_cell_morph")
# notes="Simple morphology for the HH cell"
seg = Segment(id="0", name="soma", notes="Soma segment")
# We want a diameter such that area is 1000 micro meter^2
# surface area of a sphere is 4pi r^2 = 4pi diam^2
diam = math.sqrt(1000 / math.pi)
proximal = distal = Point3DWithDiam(x="0", y="0", z="0", diameter=str(diam))
seg.proximal = proximal
seg.distal = distal
morph.segments.append(seg)
hh_cell.morphology = morph

```

(continues on next page)

(continued from previous page)

```

hh_cell_doc.cells.append(hh_cell)
pynml.write_neuroml2_file(nml2_doc=hh_cell_doc, nml2_file_name=hh_cell_fn,_
→validate=True)
return hh_cell_fn

```

Create a network

```

def create_network():
    """Create the network

    :returns: name of network nml file
    """
    net_doc = NeuroMLDocument(id="network",
                               notes="HH cell network")
    net_doc_fn = "HH_example_net.nml"
    net_doc.includes.append(IncludeType(href=create_cell()))
    # Create a population: convenient to create many cells of the same type
    pop = Population(id="pop0", notes="A population for our cell", component="hh_cell"
→", size=1)
    # Input
    pulsegen = PulseGenerator(id="pg", notes="Simple pulse generator", delay="100ms",_
→duration="100ms", amplitude="0.08nA")

    exp_input = ExplicitInput(target="pop0[0]", input="pg")

    net = Network(id="single_hh_cell_network", note="A network with a single_
→population")
    net_doc.pulse_generators.append(pulsegen)
    net.explicit_inputs.append(exp_input)
    net.populations.append(pop)
    net_doc.networks.append(net)

    pynml.write_neuroml2_file(nml2_doc=net_doc, nml2_file_name=net_doc_fn,_
→validate=True)
    return net_doc_fn

```

3.6.2 Plot the data we record

```

def plot_data(sim_id):
    """Plot the sim data.

    Load the data from the file and plot the graph for the membrane potential
    using the pynml generate_plot utility function.

    :sim_id: ID of simulation
    """
    data_array = np.loadtxt(sim_id + ".dat")
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 1]], "Membrane potential",_
→show_plot_already=False, save_figure_to=sim_id + "-v.png", xaxis="time (s)", yaxis=
→"membrane potential (V)")

```

(continues on next page)

(continued from previous page)

```
pynml.generate_plot([data_array[:, 0]], [data_array[:, 2]], "channel current",  
↳ show_plot_already=False, save_figure_to=sim_id + "-i.png", xaxis="time (s)", yaxis=  
↳ "channel current (A)")  
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 3], data_  
↳ array[:, 4]], "current density", labels=["Na", "K"], show_plot_already=False, save_  
↳ figure_to=sim_id + "-iden.png", xaxis="time (s)", yaxis="current density (A_per_m2)  
↳ ")
```

3.6.3 Create and run the simulation

Create the simulation, run it, record data, and plot the recorded information.

```
def main():  
    """Main function  
  
    Include the NeuroML model into a LEMS simulation file, run it, plot some  
    data.  
    """  
    # Simulation bits  
    sim_id = "HH_single_compartment_example_sim"  
    simulation = LEMSSimulation(sim_id=sim_id, duration=300, dt=0.01, simulation_=  
↳ seed=123)  
        # Include the NeuroML model file  
        simulation.include_neuroml2_file(create_network())  
        # Assign target for the simulation  
        simulation.assign_simulation_target("single_hh_cell_network")  
  
        # Recording information from the simulation  
        simulation.create_output_file(id="output0", file_name=sim_id + ".dat")  
        simulation.add_column_to_output_file("output0", column_id="pop0[0]/v", quantity=  
↳ "pop0[0]/v")  
            simulation.add_column_to_output_file("output0", column_id="pop0[0]/iChannels",  
↳ quantity="pop0[0]/iChannels")  
                simulation.add_column_to_output_file("output0", column_id="pop0[0]/na/iDensity",  
↳ quantity="pop0[0]/hh_b_prop/membraneProperties/na_channels/iDensity/")  
                    simulation.add_column_to_output_file("output0", column_id="pop0[0]/k/iDensity",  
↳ quantity="pop0[0]/hh_b_prop/membraneProperties/k_channels/iDensity/")  
  
        # Save LEMS simulation to file  
        sim_file = simulation.save_to_file()  
  
        # Run the simulation using the default jNeuroML simulator  
        pynml.run_lems_with_jneuroml(sim_file, max_memory="2G", nogui=True, plot=False)  
        # Plot the data  
        plot_data(sim_id)
```

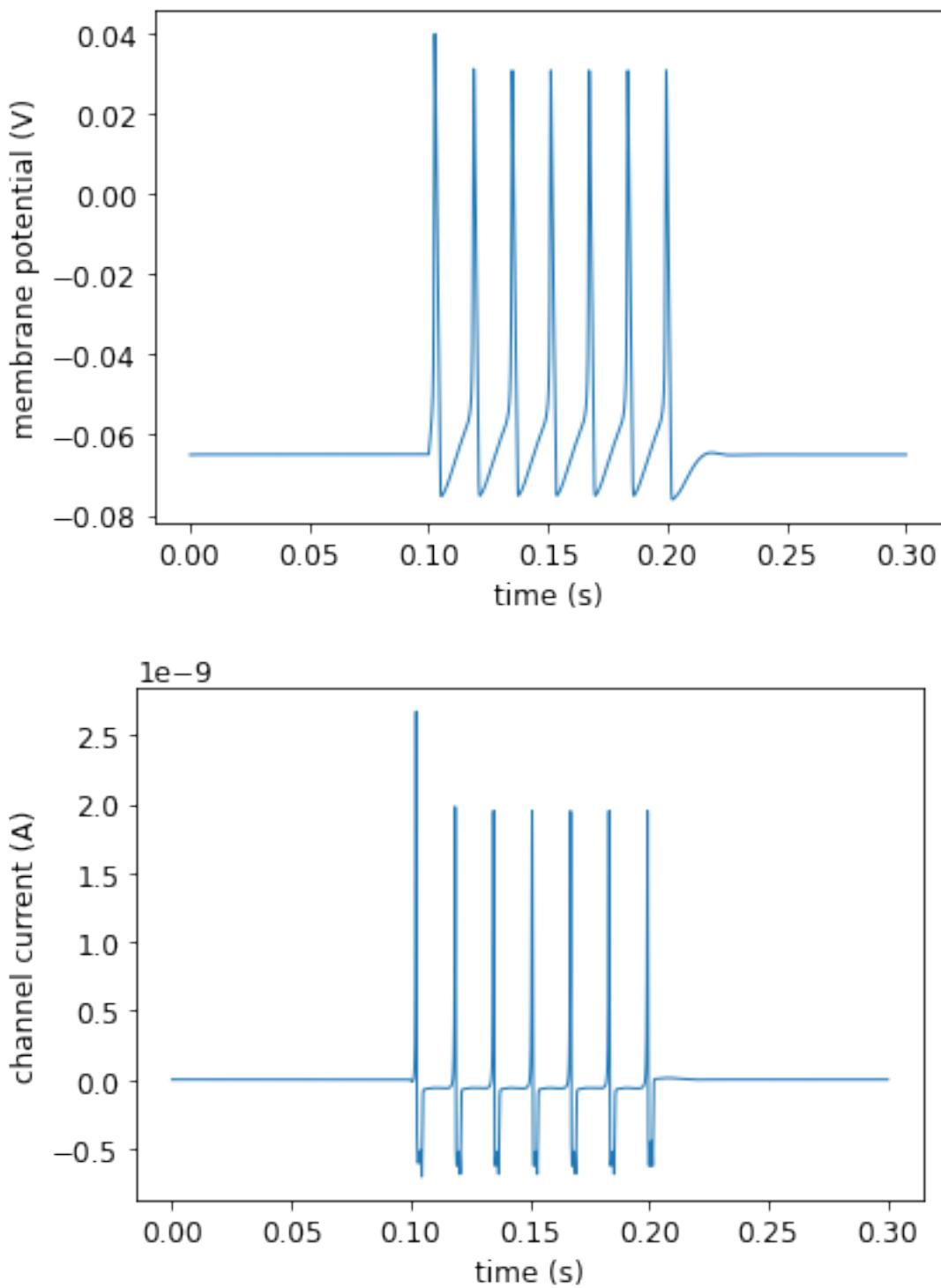
```
if __name__ == "__main__":  
    main()
```

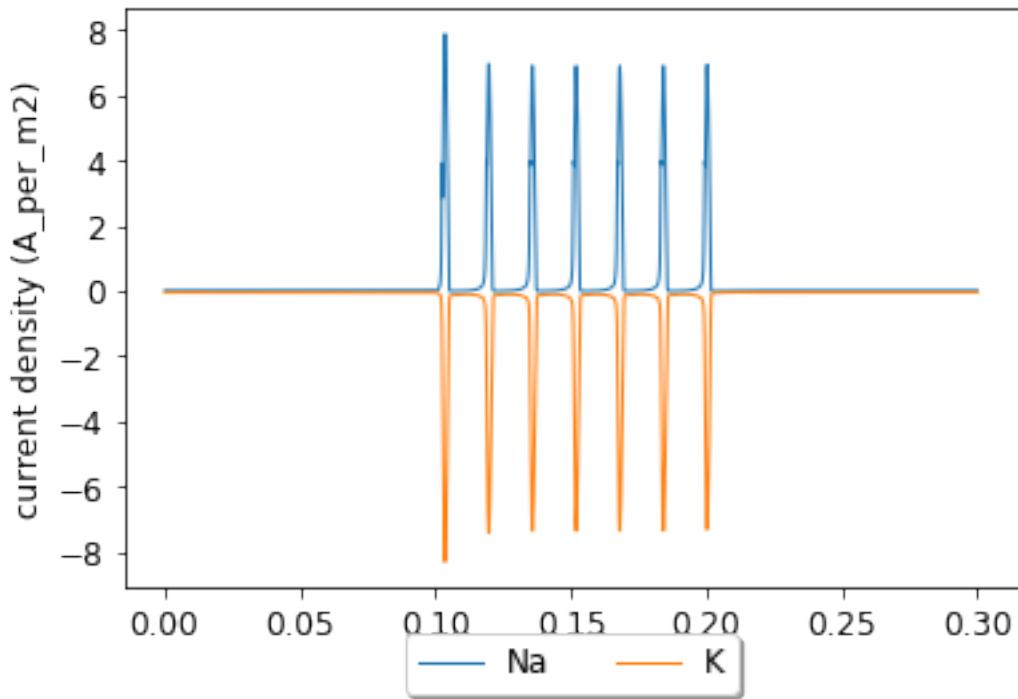
```
pyNeuroML >>> Written LEMS Simulation HH_single_compartment_example_sim to file:  
↳ LEMS_HH_single_compartment_example_sim.xml  
pyNeuroML >>> Generating plot: Membrane potential
```

```
/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1688:  
  ↳MatplotlibDeprecationWarning:  
The set_window_title function was deprecated in Matplotlib 3.4 and will be removed  
  ↳two minor releases later. Use manager.set_window_title or GUI-specific methods  
  ↳instead.  
    fig.canvas.set_window_title(title)  
/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1727: UserWarning: marker is  
  ↳redundantly defined by the 'marker' keyword argument and the fmt string "o" (->  
  ↳marker='o'). The keyword argument will take precedence.  
    plt.plot(xvalues[i], yvalues[i], 'o', marker=marker, markersize=markersize,  
  ↳linestyle=linestyle, linewidth=linewidth, label=label)  
/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1688:  
  ↳MatplotlibDeprecationWarning:  
The set_window_title function was deprecated in Matplotlib 3.4 and will be removed  
  ↳two minor releases later. Use manager.set_window_title or GUI-specific methods  
  ↳instead.  
    fig.canvas.set_window_title(title)  
/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1727: UserWarning: marker is  
  ↳redundantly defined by the 'marker' keyword argument and the fmt string "o" (->  
  ↳marker='o'). The keyword argument will take precedence.  
    plt.plot(xvalues[i], yvalues[i], 'o', marker=marker, markersize=markersize,  
  ↳linestyle=linestyle, linewidth=linewidth, label=label)
```

```
pyNeuroML >>> Saved image to HH_single_compartment_example_sim-v.png of plot:  
  ↳Membrane potential  
pyNeuroML >>> Generating plot: channel current  
pyNeuroML >>> Saved image to HH_single_compartment_example_sim-i.png of plot:  
  ↳channel current  
pyNeuroML >>> Generating plot: current density  
pyNeuroML >>> Saved image to HH_single_compartment_example_sim-iden.png of plot:  
  ↳current density
```

```
/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1688:  
  ↳MatplotlibDeprecationWarning:  
The set_window_title function was deprecated in Matplotlib 3.4 and will be removed  
  ↳two minor releases later. Use manager.set_window_title or GUI-specific methods  
  ↳instead.  
    fig.canvas.set_window_title(title)  
/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1727: UserWarning: marker is  
  ↳redundantly defined by the 'marker' keyword argument and the fmt string "o" (->  
  ↳marker='o'). The keyword argument will take precedence.  
    plt.plot(xvalues[i], yvalues[i], 'o', marker=marker, markersize=markersize,  
  ↳linestyle=linestyle, linewidth=linewidth, label=label)  
/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1727: UserWarning: marker is  
  ↳redundantly defined by the 'marker' keyword argument and the fmt string "o" (->  
  ↳marker='o'). The keyword argument will take precedence.  
    plt.plot(xvalues[i], yvalues[i], 'o', marker=marker, markersize=markersize,  
  ↳linestyle=linestyle, linewidth=linewidth, label=label)
```





3.7 Simulating a multi compartment OLM neuron

In this section we will model and simulate a multi-compartment Oriens-lacunosum moleculare (OLM) interneuron cell from the rodent hippocampal CA1 network model developed by Bezaire et al. ([BRB+16]). The complete network model can be seen [here on GitHub](#), and [here on Open Source Brain](#).

This plot, saved as `olm_example_sim_seg0_soma0-v.png` is generated using the following Python NeuroML script:

```
#!/usr/bin/env python3
"""
Multi-compartmental OLM cell example

File: olm-example.py

Copyright 2021 NeuroML contributors
Authors: Padraig Gleeson, Ankur Sinha
"""

from neuroml import (NeuroMLDocument, IncludeType, Population, PulseGenerator,
                     ExplicitInput, Network, SegmentGroup, Member, Property, Include, Instance, Location)
from CellBuilder import (create_cell, add_segment, add_channel_density, set_init_membrane_potential,
                         set_resistivity, set_specific_capacitance, get_seg_group_by_id)
from pyneuroml import pynml
from pyneuroml.lemn import LEMSSimulation
import numpy as np

def main():
    """Main function
```

(continues on next page)

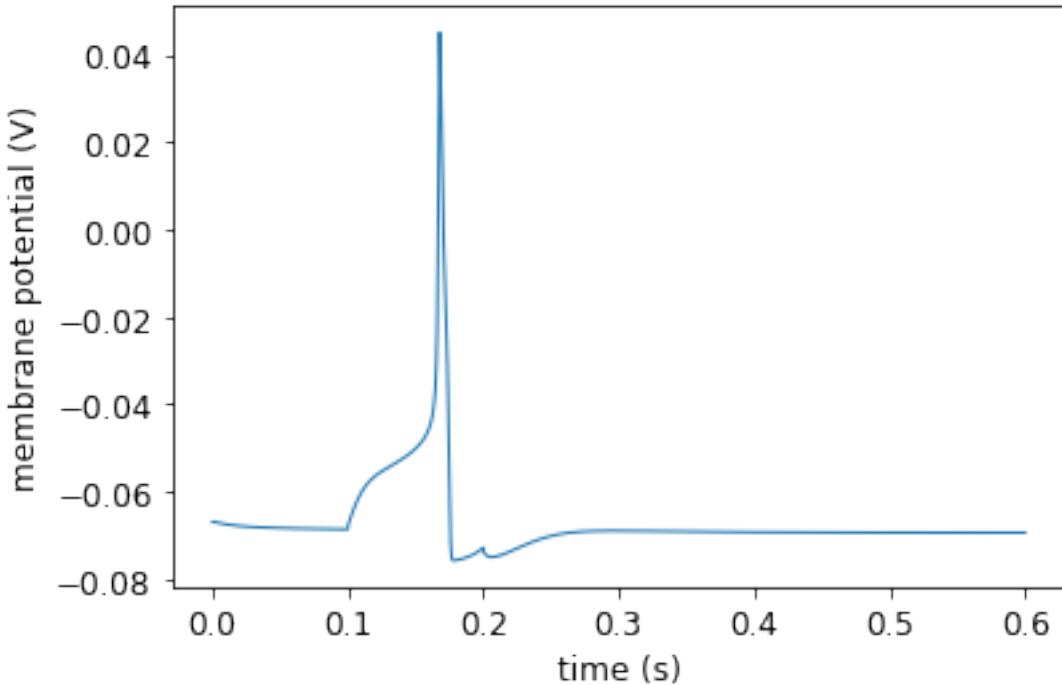


Fig. 3.15: Membrane potential of the simulated OLM cell at the soma.

(continued from previous page)

```

Include the NeuroML model into a LEMS simulation file, run it, plot some
data.
"""
# Simulation bits
sim_id = "olm_example_sim"
simulation = LEMSSimulation(sim_id=sim_id, duration=600, dt=0.01, simulation_
seed=123)
# Include the NeuroML model file
simulation.include_neuroml2_file(create_olm_network())
# Assign target for the simulation
simulation.assign_simulation_target("single_olm_cell_network")

# Recording information from the simulation
simulation.create_output_file(id="output0", file_name=sim_id + ".dat")
simulation.add_column_to_output_file("output0", column_id="pop0_0_v", quantity=
"pop0[0]/v")
simulation.add_column_to_output_file("output0",
                                    column_id="pop0_0_v_Seg0_soma_0",
                                    quantity="pop0/0/olm/0/v")
simulation.add_column_to_output_file("output0",
                                    column_id="pop0_0_v_Seg1_soma_0",
                                    quantity="pop0/0/olm/1/v")
simulation.add_column_to_output_file("output0",
                                    column_id="pop0_0_v_Seg0_axon_0",
                                    quantity="pop0/0/olm/2/v")
simulation.add_column_to_output_file("output0",
                                    column_id="pop0_0_v_Seg1_axon_0",
                                    quantity="pop0/0/olm/3/v")

```

(continues on next page)

(continued from previous page)

```

        quantity="pop0/0/olm/3/v")
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg0_dend_0",
                                      quantity="pop0/0/olm/4/v")
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg1_dend_0",
                                      quantity="pop0/0/olm/6/v")
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg0_dend_1",
                                      quantity="pop0/0/olm/5/v")
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg1_dend_1",
                                      quantity="pop0/0/olm/7/v")

# Save LEMS simulation to file
sim_file = simulation.save_to_file()

# Run the simulation using the NEURON simulator
pynml.run_lems_with_jneuroml_neuron(sim_file, max_memory="2G", nogui=True,
                                      plot=False, skip_run=False)

# Plot the data
plot_data(sim_id)

def plot_data(sim_id):
    """Plot the sim data.

    Load the data from the file and plot the graph for the membrane potential
    using the pynml generate_plot utility function.

    :sim_id: ID of simulation

    """
    data_array = np.loadtxt(sim_id + ".dat")
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 1]], "Membrane potential"
                        +(soma seg 0)", show_plot_already=False, save_figure_to=sim_id + "_seg0_soma0-v.png",
                        xaxis="time (s)", yaxis="membrane potential (V)")
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 2]], "Membrane potential"
                        +(soma seg 1)", show_plot_already=False, save_figure_to=sim_id + "_seg1_soma0-v.png",
                        xaxis="time (s)", yaxis="membrane potential (V)")
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 3]], "Membrane potential"
                        +(axon seg 0)", show_plot_already=False, save_figure_to=sim_id + "_seg0_axon0-v.png",
                        xaxis="time (s)", yaxis="membrane potential (V)")
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 4]], "Membrane potential"
                        +(axon seg 1)", show_plot_already=False, save_figure_to=sim_id + "_seg1_axon0-v.png",
                        xaxis="time (s)", yaxis="membrane potential (V)")

def create_olm_network():
    """Create the network

    :returns: name of network nml file
    """
    net_doc = NeuroMLDocument(id="network",
                               notes="OLM cell network")
    net_doc_fn = "olm_example_net.nml"
    net_doc.includes.append(IncludeType(href=create_olm_cell()))

```

(continues on next page)

(continued from previous page)

```

# Create a population: convenient to create many cells of the same type
pop = Population(id="pop0", notes="A population for our cell",
                  component="olm", size=1, type="populationList")
pop.instances.append(Instance(id=1, location=Location(0., 0., 0.)))
# Input
pulsegen = PulseGenerator(id="pg_olm", notes="Simple pulse generator", delay=
    "100ms", duration="100ms", amplitude="0.08nA")

exp_input = ExplicitInput(target="pop0[0]", input="pg_olm")

net = Network(id="single_olm_cell_network", note="A network with a single_
population")
net_doc.pulse_generators.append(pulsegen)
net.explicit_inputs.append(exp_input)
net.populations.append(pop)
net_doc.networks.append(net)

pynml.write_neuroml2_file(nml2_doc=net_doc, nml2_file_name=net_doc_fn,_
validate=True)
return net_doc_fn

def create_olm_cell():
    """Create the complete cell.

    :returns: cell object
    """
    nml_cell_doc = NeuroMLDocument(id="olm_cell")
    cell = create_cell("olm")
    nml_cell_file = cell.id + ".cell.nml"

    # Add two soma segments
    diam = 10.0
    soma_0 = add_segment(cell,
                          prox=[0.0, 0.0, 0.0, diam],
                          dist=[0.0, 10., 0.0, diam],
                          name="Seg0_soma_0",
                          group="soma_0")

    soma_1 = add_segment(cell,
                          prox=None,
                          dist=[0.0, 10. + 10., 0.0, diam],
                          name="Seg1_soma_0",
                          parent=soma_0,
                          group="soma_0")

    # Add axon segments
    diam = 1.5
    axon_0 = add_segment(cell,
                          prox=[0.0, 0.0, 0.0, diam],
                          dist=[0.0, -75, 0.0, diam],
                          name="Seg0_axon_0",
                          parent=soma_0,
                          fraction_along=0.0,
                          group="axon_0")
    axon_1 = add_segment(cell,

```

(continues on next page)

(continued from previous page)

```

prox=None,
dist=[0.0, -150, 0.0, diam],
name="Seg1_axon_0",
parent=axon_0,
group="axon_0")

# Add 2 dendrite segments

diam = 3.0
dend_0_0 = add_segment(cell,
                       prox=[0.0, 20, 0.0, diam],
                       dist=[100, 120, 0.0, diam],
                       name="Seg0_dend_0",
                       parent=soma_1,
                       fraction_along=1,
                       group="dend_0")

dend_1_0 = add_segment(cell,
                       prox=None,
                       dist=[177, 197, 0.0, diam],
                       name="Seg1_dend_0",
                       parent=dend_0_0,
                       fraction_along=1,
                       group="dend_0")

dend_0_1 = add_segment(cell,
                       prox=[0.0, 20, 0.0, diam],
                       dist=[-100, 120, 0.0, diam],
                       name="Seg0_dend_1",
                       parent=soma_1,
                       fraction_along=1,
                       group="dend_1")
dend_1_1 = add_segment(cell,
                       prox=None,
                       dist=[-177, 197, 0.0, diam],
                       name="Seg1_dend_1",
                       parent=dend_0_1,
                       fraction_along=1,
                       group="dend_1")

# XXX: For segment groups to be correctly mapped to sections in NEURON,
# they must include the correct neurolex ID
for section_name in ["soma_0", "axon_0", "dend_0", "dend_1"]:
    section_group = get_seg_group_by_id(section_name, cell)
    section_group.neuro_lex_id = 'sao864921383'

den_seg_group = get_seg_group_by_id("dendrite_group", cell)
den_seg_group.includes.append(Include(segment_groups="dend_0"))
den_seg_group.includes.append(Include(segment_groups="dend_1"))
den_seg_group.properties.append(Property(tag="color", value="0.8 0 0"))

ax_seg_group = get_seg_group_by_id("axon_group", cell)
ax_seg_group.includes.append(Include(segment_groups="axon_0"))
ax_seg_group.properties.append(Property(tag="color", value="0 0.8 0"))

soma_seg_group = get_seg_group_by_id("soma_group", cell)

```

(continues on next page)

(continued from previous page)

```
soma_seg_group.includes.append(Include(segment_groups="soma_0"))

soma_seg_group.properties.append(Property(tag="color", value="0 0 0.8"))

# Other cell properties
set_init_memb_potential(cell, "-67mV")
set_resistivity(cell, "0.15 kohm_cm")
set_specific_capacitance(cell, "1.3 uF_per_cm2")

# channels
# leak
add_channel_density(cell, nml_cell_doc,
                    cd_id="leak_all",
                    cond_density="0.01 mS_per_cm2",
                    ion_channel="leak_chan",
                    ion_chan_def_file="olm-example/leak_chan.channel.nml",
                    erev="-67mV",
                    ion="non_specific")

# HCNolm_soma
add_channel_density(cell, nml_cell_doc,
                    cd_id="HCNolm_soma",
                    cond_density="0.5 mS_per_cm2",
                    ion_channel="HCNolm",
                    ion_chan_def_file="olm-example/HCNolm.channel.nml",
                    erev="-32.9mV",
                    ion="h",
                    group="soma_group")

# Kdrfast_soma
add_channel_density(cell, nml_cell_doc,
                    cd_id="Kdrfast_soma",
                    cond_density="73.37 mS_per_cm2",
                    ion_channel="Kdrfast",
                    ion_chan_def_file="olm-example/Kdrfast.channel.nml",
                    erev="-77mV",
                    ion="k",
                    group="soma_group")

# Kdrfast_dendrite
add_channel_density(cell, nml_cell_doc,
                    cd_id="Kdrfast_dendrite",
                    cond_density="105.8 mS_per_cm2",
                    ion_channel="Kdrfast",
                    ion_chan_def_file="olm-example/Kdrfast.channel.nml",
                    erev="-77mV",
                    ion="k",
                    group="dendrite_group")

# Kdrfast_axon
add_channel_density(cell, nml_cell_doc,
                    cd_id="Kdrfast_axon",
                    cond_density="117.392 mS_per_cm2",
                    ion_channel="Kdrfast",
                    ion_chan_def_file="olm-example/Kdrfast.channel.nml",
                    erev="-77mV",
                    ion="k",
                    group="axon_group")

# KvAolm_soma
add_channel_density(cell, nml_cell_doc,
```

(continues on next page)

(continued from previous page)

```

        cd_id="KvAolm_soma",
        cond_density="4.95 mS_per_cm2",
        ion_channel="KvAolm",
        ion_chan_def_file="olm-example/KvAolm.channel.nml",
        erev="-77mV",
        ion="k",
        group="soma_group")

# KvAolm_dendrite
add_channel_density(cell, nml_cell_doc,
                    cd_id="KvAolm_dendrite",
                    cond_density="2.8 mS_per_cm2",
                    ion_channel="KvAolm",
                    ion_chan_def_file="olm-example/KvAolm.channel.nml",
                    erev="-77mV",
                    ion="k",
                    group="dendrite_group")

# Nav_soma
add_channel_density(cell, nml_cell_doc,
                    cd_id="Nav_soma",
                    cond_density="10.7 mS_per_cm2",
                    ion_channel="Nav",
                    ion_chan_def_file="olm-example/Nav.channel.nml",
                    erev="50mV",
                    ion="na",
                    group="soma_group")

# Nav_dendrite
add_channel_density(cell, nml_cell_doc,
                    cd_id="Nav_dendrite",
                    cond_density="23.4 mS_per_cm2",
                    ion_channel="Nav",
                    ion_chan_def_file="olm-example/Nav.channel.nml",
                    erev="50mV",
                    ion="na",
                    group="dendrite_group")

# Nav_axon
add_channel_density(cell, nml_cell_doc,
                    cd_id="Nav_axon",
                    cond_density="17.12 mS_per_cm2",
                    ion_channel="Nav",
                    ion_chan_def_file="olm-example/Nav.channel.nml",
                    erev="50mV",
                    ion="na",
                    group="axon_group")

nml_cell_doc.cells.append(cell)
pynml.write_neuroml2_file(nml_cell_doc, nml_cell_file, True, True)
return nml_cell_file

if __name__ == "__main__":
    main()

```

As we will see, we repeat the same operations in the script while adding segments and ion-channels to our model, so we also write some helper functions to make it easier for ourselves:

```

#!/usr/bin/env python3
"""
Utility functions to help build cells in NeuroML

File: CellBuilder.py

Copyright 2021 NeuroML contributors
Author: Ankur Sinha <sanjay DOT ankur AT gmail DOT com>
"""

from typing import List
from neuroml import (Cell, Morphology, MembraneProperties, IntracellularProperties,
                     BiophysicalProperties, Segment, SegmentGroup, Point3DWithDiam, SegmentParent,
                     Member, InitMembPotential, Resistivity, SpecificCapacitance, NeuroMLDocument,
                     IncludeType, ChannelDensity) # type: ignore # noqa
from pyneuroml.pynml import print_function # type: ignore

neuro_lex_ids = {
    'axon': "GO:0030424",
    'dend': "GO:0030425",
    'soma': "GO:0043025",
}

def create_cell(cell_id):
    """Create a NeuroML Cell.

    Initialises the cell with these properties assigning IDs where applicable:
    - Morphology: "morphology"
    - BiophysicalProperties: "biophys"
    - MembraneProperties
    - IntracellularProperties
    - SegmentGroups: "all", "soma_group", "dendrite_group", "axon_group" which
      can be used to include all, soma, dendrite, and axon segments
      respectively.

    Note that since this cell does not currently include a segment in its
    morphology, it is *not* a valid NeuroML construct. Use the `add_segment`-
    function to add segments. `add_segment` will also populate the default
    segment groups this creates.

    :param cell_id: id of the cell
    :type cell_id: str
    :returns: created cell object of type neuroml.Cell

    """
    cell = Cell(id=cell_id)
    cell.morphology = Morphology(id='morphology')
    membrane_properties = MembraneProperties()
    intracellular_properties = IntracellularProperties()

    cell.biophysical_properties = BiophysicalProperties(
        id="biophys", intracellular_properties=intracellular_properties,
        membrane_properties=membrane_properties)

```

(continues on next page)

(continued from previous page)

```

seg_group_all = SegmentGroup(id='all')
seg_group_soma = SegmentGroup(id='soma_group',
                               neuro_lex_id=neuro_lex_ids["soma"],
                               notes="Default soma segment group for the cell")
seg_group_axon = SegmentGroup(id='axon_group',
                               neuro_lex_id=neuro_lex_ids["axon"],
                               notes="Default axon segment group for the cell")
seg_group_dend = SegmentGroup(id='dendrite_group',
                               neuro_lex_id=neuro_lex_ids["dend"],
                               notes="Default dendrite segment group for the cell")
cell.morphology.segment_groups.append(seg_group_all)
cell.morphology.segment_groups.append(seg_group_soma)
cell.morphology.segment_groups.append(seg_group_axon)
cell.morphology.segment_groups.append(seg_group_dend)

return cell


def add_segment(cell, prox, dist, name=None, parent=None, fraction_along=1.0, ↴
group=None):
    # type: (Cell, List[float], List[float], str, SegmentParent, float, SegmentGroup) ↴
    """Add a segment to the cell.

    Convention: use axon_, soma_, dend_ prefixes for axon, soma, and dendrite
    segments respectively. This will allow this function to add the correct
    neurolex IDs to the group.

    The created segment is also added to the default segment groups that were
    created by the `create_cell` function: "all", "dendrite_group",
    "soma_group", "axon_group" if the convention is followed.

    :param cell: cell to add segment to
    :type cell: Cell
    :param prox: proximal segment information
    :type prox: list with 4 float entries: [x, y, z, diameter]
    :param dist: dist segment information
    :type dist: list with 4 float entries: [x, y, z, diameter]
    :param name: name of segment
    :type name: str
    :param parent: parent segment
    :type parent: SegmentParent
    :param fraction_along: where the new segment is connected to the parent (0:→
    distal point, 1: proximal point)
    :type fraction_along: float
    :param group: segment group to add the segment to
    :type group: SegmentGroup
    :returns: the created segment

    """
    try:
        if prox:
            p = Point3DWithDiam(x=prox[0], y=prox[1], z=prox[2], diameter=prox[3])
        else:
            p = None
    except:
        raise Exception("Proximal information must be provided")
    if parent:
        if parent not in cell.morphology.segment_groups:
            raise Exception("Parent segment not found in morphology")
        parent.add_segment(p, fraction_along)
    else:
        if group:
            if group not in cell.morphology.segment_groups:
                raise Exception("Segment group not found in morphology")
            group.add_segment(p, fraction_along)
        else:
            seg_group_all.add_segment(p, fraction_along)
    return p

```

(continues on next page)

(continued from previous page)

```

except IndexError as e:
    print_function("{}: prox must be a list of 4 elements".format(e))
try:
    d = Point3DWithDiam(x=dist[0], y=dist[1], z=dist[2], diameter=dist[3])
except IndexError as e:
    print_function("{}: dist must be a list of 4 elements".format(e))

segid = len(cell.morphology.segments)

sp = SegmentParent(segments=parent.id, fraction_along=fraction_along) if parent_
else None
segment = Segment(id=segid, proximal=p, distal=d, parent=sp)

if name:
    segment.name = name

if group:
    seg_group = None
    seg_group = get_seg_group_by_id(group, cell)
    seg_group_all = get_seg_group_by_id("all", cell)
    seg_group_default = None
    neuro_lex_id = None

    if "axon_" in group:
        neuro_lex_id = neuro_lex_ids["axon"] # See http://amigo.geneontology.org/
        ↪amigo/term/GO:0030424
        seg_group_default = get_seg_group_by_id("axon_group", cell)
    if "soma_" in group:
        neuro_lex_id = neuro_lex_ids["soma"]
        seg_group_default = get_seg_group_by_id("soma_group", cell)
    if "dend_" in group:
        neuro_lex_id = neuro_lex_ids["dend"]
        seg_group_default = get_seg_group_by_id("dendrite_group", cell)

    if seg_group is None:
        seg_group = SegmentGroup(id=group, neuro_lex_id=neuro_lex_id)
        cell.morphology.segment_groups.append(seg_group)

    seg_group.members.append(Member(segments=segment.id))
    # Ideally, these higher level segment groups should just include other
    # segment groups using Include, which would result in smaller NML
    # files. However, because these default segment groups are defined
    # first, they are printed in the NML file before the new segments and their
    # groups. The jnml validator does not like this.
    # TODO: clarify if the order of definition is important, or if the jnml
    # validator needs to be updated to manage this use case.
    if seg_group_default:
        seg_group_default.members.append(Member(segments=segment.id))

seg_group_all.members.append(Member(segments=segment.id))

cell.morphology.segments.append(segment)
return segment

def set_init_memb_potential(cell, v, group="all"):
```

(continues on next page)

(continued from previous page)

```

# type: (Cell, str, str) -> None
"""Set the initial membrane potential of the cell.

:param cell: cell to modify
:type cell: Cell
:param v: value to set for membrane potential with units
:type v: str
:param group: id of segment group to modify
:type group: str
"""
cell.biophysical_properties.membrane_properties.init_memb_potentials = \
    [InitMembPotential(value=v, segment_groups=group)]


def set_resistivity(cell, resistivity, group="all"):
    # type: (Cell, str, str) -> None
    """Set the resistivity of the cell

    :param cell: cell to modify
    :param resistivity: value resistivity to set with units
    :type resistivity: str
    :param group: segment group to modify
    :type group: str
    """
    cell.biophysical_properties.intracellular_properties.resistivities = \
        [Resistivity(value=resistivity, segment_groups=group)]


def set_specific_capacitance(cell, spec_cap, group="all"):
    # type: (Cell, str, str) -> None
    """Set the specific capacitance for the cell.

    :param cell: cell to set specific capacitance for
    :type cell: Cell
    :param spec_cap: value of specific capacitance with units
    :type spec_cap: str
    :param group: segment group to modify
    :type group: str
    """
    cell.biophysical_properties.membrane_properties.specIFIC_capacitances.\
        append(SpecificCapacitance(value=spec_cap, segment_groups=group))


def add_channel_density(cell, nml_cell_doc, cd_id, cond_density, ion_channel, ion_
    ↪chan_def_file="", erev="0.0 mV", ion="non_specific", group="all"):
    # type: (Cell, NeuroMLDocument, str, str, str, str, str, str, str) -> None
    """Add channel density.

    :param cell: cell to be modified
    :type cell: Cell
    :param nml_cell_doc: cell NeuroML document to which channel density is to be added
    :type nml_cell_doc: NeuroMLDocument
    :param cd_id: id for channel density
    :type cd_id: str
    :param cond_density: value of conductance density with units
    :type cond_density: str

```

(continues on next page)

(continued from previous page)

```

:param ion_channel: name of ion channel
:type ion_channel: str
:param ion_chan_def_file: path to NeuroML2 file defining the ion channel, if
empty, it assumes the channel is defined in the same file
:type ion_chan_def_file: str
:param erev: value of reversal potential with units
:type erev: str
:param ion: name of ion
:type ion: str
:param group: segment groups to add to
:type group: str
"""

cd = ChannelDensity(id=cd_id,
                     segment_groups=group,
                     ion=ion,
                     ion_channel=ion_channel,
                     erev=erev,
                     cond_density=cond_density)

cell.biophysical_properties.membrane_properties.channel_densities.append(cd)

if len(ion_chan_def_file) > 0:
    if IncludeType(ion_chan_def_file) not in nml_cell_doc.includes:
        nml_cell_doc.includes.append(IncludeType(ion_chan_def_file))

```



```

def get_seg_group_by_id(sg_id, cell):
    # type (str, Cell) -> SegmentGroup
    """Return the SegmentGroup object for the specified segment group id.

    :param sg_id: id of segment group to find
    :type sg_id: str
    :param cell: cell to look for segment group in
    :type cell: Cell
    :returns: SegmentGroup object of specified ID or None if not found

    """
    if not sg_id or not cell:
        print_function("Please specify both a segment group id and a Cell")
        return None

    for sg in cell.morphology.segment_groups:
        if sg.id == sg_id:
            return sg

    return None

```



```

def get_seg_group_by_id_substring(sg_id_substring, cell):
    # type (str, Cell) -> list
    """Return segment groups that include the specified substring.

    :param sg_id_substring: substring to match
    :type sg_id_substring: str
    :param cell: cell to look for segment group in
    :type cell: Cell

```

(continues on next page)

(continued from previous page)

```
:returns: list of SegmentGroups whose IDs include the given substring

"""

sg_group_list = []
if not sg_id_substring or not cell:
    print_function("Please specify both a segment group id and a Cell")
    return None
for sg in cell.morphology.segment_groups:
    if sg_id_substring in sg.id:
        sg_group_list.append(sg)
return sg_group_list
```

These helper functions will be included in the Python NeuroML libraries in their next release. Currently, we *import* them into our Python script at the top:

```
from CellBuilder import (create_cell, add_segment, add_channel_density, set_init_memb_
                           _potential, set_resistivity, set_specific_capacitance, get_seg_group_by_id)
```

3.7.1 Declaring the model in NeuroML

Similar to previous examples, we will first declare the model, visualise it, and then simulate it. The OLM model is slightly more complex than the HH neuron model we had worked with in the [previous tutorial](#) since it includes multiple compartments. However, where we had declared the ion-channels ourselves in the previous example, here will will not do so. We will *include* channels that have been pre-defined in NeuroML to demonstrate how components defined in NeuroML can be easily re-used in models.

We will follow the same method as before. We will first define the cell, create a network with one instance of the cell, and then simulate it to record and plot the membrane potential from different segments.

Declaring the cell

To keep our Python script modularised, we start constructing our *cell* in a separate function.

```
def create_olm_cell():
    """Create the complete cell.

    :returns: cell object
    """
    nml_cell_doc = NeuroMLDocument(id="olm_cell")
    cell = create_cell("olm")
    nml_cell_file = cell.id + ".cell.nml"

    # Add two soma segments
    diam = 10.0
    soma_0 = add_segment(cell,
                         prox=[0.0, 0.0, 0.0, diam],
                         dist=[0.0, 10., 0.0, diam],
                         name="Seg0_soma_0",
                         group="soma_0")

    soma_1 = add_segment(cell,
                         prox=None,
```

(continues on next page)

(continued from previous page)

```

        dist=[0.0, 10. + 10., 0.0, diam],
        name="Seg1_soma_0",
        parent=soma_0,
        group="soma_0")

# Add axon segments
diam = 1.5
axon_0 = add_segment(cell,
                      prox=[0.0, 0.0, 0.0, diam],
                      dist=[0.0, -75, 0.0, diam],
                      name="Seg0_axon_0",
                      parent=soma_0,
                      fraction_along=0.0,
                      group="axon_0")
axon_1 = add_segment(cell,
                      prox=None,
                      dist=[0.0, -150, 0.0, diam],
                      name="Seg1_axon_0",
                      parent=axon_0,
                      group="axon_0")

# Add 2 dendrite segments

diam = 3.0
dend_0_0 = add_segment(cell,
                       prox=[0.0, 20, 0.0, diam],
                       dist=[100, 120, 0.0, diam],
                       name="Seg0_dend_0",
                       parent=soma_1,
                       fraction_along=1,
                       group="dend_0")

dend_1_0 = add_segment(cell,
                       prox=None,
                       dist=[177, 197, 0.0, diam],
                       name="Seg1_dend_0",
                       parent=dend_0_0,
                       fraction_along=1,
                       group="dend_0")

dend_0_1 = add_segment(cell,
                       prox=[0.0, 20, 0.0, diam],
                       dist=[-100, 120, 0.0, diam],
                       name="Seg0_dend_1",
                       parent=soma_1,
                       fraction_along=1,
                       group="dend_1")
dend_1_1 = add_segment(cell,
                       prox=None,
                       dist=[-177, 197, 0.0, diam],
                       name="Seg1_dend_1",
                       parent=dend_0_1,
                       fraction_along=1,
                       group="dend_1")

# XXX: For segment groups to be correctly mapped to sections in NEURON,

```

(continues on next page)

(continued from previous page)

```

# they must include the correct neurolex ID
for section_name in ["soma_0", "axon_0", "dend_0", "dend_1"]:
    section_group = get_seg_group_by_id(section_name, cell)
    section_group.neuro_lex_id = 'sao864921383'

den_seg_group = get_seg_group_by_id("dendrite_group", cell)
den_seg_group.includes.append(Include(segment_groups="dend_0"))
den_seg_group.includes.append(Include(segment_groups="dend_1"))
den_seg_group.properties.append(Property(tag="color", value="0.8 0 0"))

ax_seg_group = get_seg_group_by_id("axon_group", cell)
ax_seg_group.includes.append(Include(segment_groups="axon_0"))
ax_seg_group.properties.append(Property(tag="color", value="0 0.8 0"))

soma_seg_group = get_seg_group_by_id("soma_group", cell)
soma_seg_group.includes.append(Include(segment_groups="soma_0"))

soma_seg_group.properties.append(Property(tag="color", value="0 0 0.8"))

# Other cell properties
set_init_memb_potential(cell, "-67mV")
set_resistivity(cell, "0.15 kohm_cm")
set_specific_capacitance(cell, "1.3 uF_per_cm2")

# channels
# leak
add_channel_density(cell, nml_cell_doc,
                    cd_id="leak_all",
                    cond_density="0.01 mS_per_cm2",
                    ion_channel="leak_chan",
                    ion_chan_def_file="olm-example/leak Chan.channel.nml",
                    erev="-67mV",
                    ion="non_specific")

# HCNolm_soma
add_channel_density(cell, nml_cell_doc,
                    cd_id="HCNolm_soma",
                    cond_density="0.5 mS_per_cm2",
                    ion_channel="HCNolm",
                    ion_chan_def_file="olm-example/HCNolm.channel.nml",
                    erev="-32.9mV",
                    ion="h",
                    group="soma_group")

# Kdrfast_soma
add_channel_density(cell, nml_cell_doc,
                    cd_id="Kdrfast_soma",
                    cond_density="73.37 mS_per_cm2",
                    ion_channel="Kdrfast",
                    ion_chan_def_file="olm-example/Kdrfast.channel.nml",
                    erev="-77mV",
                    ion="k",
                    group="soma_group")

# Kdrfast_dendrite
add_channel_density(cell, nml_cell_doc,
                    cd_id="Kdrfast_dendrite",
                    cond_density="105.8 mS_per_cm2",
                    ion_channel="Kdrfast",

```

(continues on next page)

(continued from previous page)

```

ion_chan_def_file="olm-example/Kdrfast.channel.nml",
erev="-77mV",
ion="k",
group="dendrite_group")

# Kdrfast_axon
add_channel_density(cell, nml_cell_doc,
cd_id="Kdrfast_axon",
cond_density="117.392 mS_per_cm2",
ion_channel="Kdrfast",
ion_chan_def_file="olm-example/Kdrfast.channel.nml",
erev="-77mV",
ion="k",
group="axon_group")

# KvAolm_soma
add_channel_density(cell, nml_cell_doc,
cd_id="KvAolm_soma",
cond_density="4.95 mS_per_cm2",
ion_channel="KvAolm",
ion_chan_def_file="olm-example/KvAolm.channel.nml",
erev="-77mV",
ion="k",
group="soma_group")

# KvAolm_dendrite
add_channel_density(cell, nml_cell_doc,
cd_id="KvAolm_dendrite",
cond_density="2.8 mS_per_cm2",
ion_channel="KvAolm",
ion_chan_def_file="olm-example/KvAolm.channel.nml",
erev="-77mV",
ion="k",
group="dendrite_group")

# Nav_soma
add_channel_density(cell, nml_cell_doc,
cd_id="Nav_soma",
cond_density="10.7 mS_per_cm2",
ion_channel="Nav",
ion_chan_def_file="olm-example/Nav.channel.nml",
erev="50mV",
ion="na",
group="soma_group")

# Nav_dendrite
add_channel_density(cell, nml_cell_doc,
cd_id="Nav_dendrite",
cond_density="23.4 mS_per_cm2",
ion_channel="Nav",
ion_chan_def_file="olm-example/Nav.channel.nml",
erev="50mV",
ion="na",
group="dendrite_group")

# Nav_axon
add_channel_density(cell, nml_cell_doc,
cd_id="Nav_axon",
cond_density="17.12 mS_per_cm2",
ion_channel="Nav",
ion_chan_def_file="olm-example/Nav.channel.nml",
erev="50mV",

```

(continues on next page)

(continued from previous page)

```

        ion="na",
        group="axon_group")

nml_cell_doc.cells.append(cell)
pynml.write_neuroml2_file(nml_cell_doc, nml_cell_file, True, True)
return nml_cell_file

```

Let us walk through this function:

```

nml_cell_doc = NeuroMLDocument(id="oml_cell")
cell = create_cell("olm")
nml_cell_file = cell.id + ".cell.nml"

```

First, we create a new NeuroML document that we will use to save this cell. Then, we proceed to create a new cell using the *Cell NeuroML component type*, and define the name of the file we will use to store the cell in NeuroML. To create the cell, we use the `create_cell` utility function:

```

def create_cell(cell_id):
    cell = Cell(id=cell_id)
    cell.morphology = Morphology(id='morphology')
    membrane_properties = MembraneProperties()
    intracellular_properties = IntracellularProperties()

    cell.biophysical_properties = BiophysicalProperties(
        id="biophys", intracellular_properties=intracellular_properties,
        membrane_properties=membrane_properties)

    seg_group_all = SegmentGroup(id='all')
    seg_group_soma = SegmentGroup(id='soma_group',
                                  neuro_lex_id=neuro_lex_ids["soma"],
                                  notes="Default soma segment group for the cell")
    seg_group_axon = SegmentGroup(id='axon_group',
                                  neuro_lex_id=neuro_lex_ids["axon"],
                                  notes="Default axon segment group for the cell")
    seg_group_dend = SegmentGroup(id='dendrite_group',
                                  neuro_lex_id=neuro_lex_ids["dend"],
                                  notes="Default dendrite segment group for the cell")
    cell.morphology.segment_groups.append(seg_group_all)
    cell.morphology.segment_groups.append(seg_group_soma)
    cell.morphology.segment_groups.append(seg_group_axon)
    cell.morphology.segment_groups.append(seg_group_dend)

    return cell

```

We now know that a *Cell* component has two children: *morphology*, and *biophysical properties*. The function simply creates the new *Cell* component for us and initialises the *morphology* and *biophysical properties*. Additionally, it creates some default *segment groups* for us that we use to organise our segments in later.

We now have an empty cell. Since we are building a multi-compartmental cell, we now proceed to define the detailed morphology of the cell. We do this by adding *segments* and grouping them in to *segment groups* which are both children elements of the *morphology* of the cell. This is done using the `add_segment` utility function as required:

```
soma_0 = add_segment(cell,
                      prox=[0.0, 0.0, 0.0, diam],
                      dist=[0.0, 10., 0.0, diam],
                      name="Seg0_soma_0",
                      group="soma_0")
```

The utility function takes the dimensions of the segment—it's *proximal* and *distal* co-ordinates and the diameter to create a segment of the provided name. Additionally, since segments need to be contiguous, it adds the segment to a *parent* segment. Finally, it places the segment into the specified segment group and the default groups that we also have and adds the segment to the cell's morphology.

```
def add_segment(cell, prox, dist, name=None, parent=None, fraction_along=1.0,_
                group=None):
    try:
        if prox:
            p = Point3DWithDiam(x=prox[0], y=prox[1], z=prox[2], diameter=prox[3])
        else:
            p = None
    except IndexError as e:
        print_function("{}: prox must be a list of 4 elements".format(e))
    try:
        d = Point3DWithDiam(x=dist[0], y=dist[1], z=dist[2], diameter=dist[3])
    except IndexError as e:
        print_function("{}: dist must be a list of 4 elements".format(e))

    segid = len(cell.morphology.segments)

    sp = SegmentParent(segments=parent.id, fraction_along=fraction_along) if parent_
    else None
    segment = Segment(id=segid, proximal=p, distal=d, parent=sp)

    if name:
        segment.name = name

    if group:
        seg_group = None
        seg_group = get_seg_group_by_id(group, cell)
        seg_group_all = get_seg_group_by_id("all", cell)
        seg_group_default = None
        neuro_lex_id = None

        if "axon_" in group:
            neuro_lex_id = neuro_lex_ids["axon"] # See http://amigo.geneontology.org/
            amigo/term/GO:0030424
            seg_group_default = get_seg_group_by_id("axon_group", cell)
        if "soma_" in group:
            neuro_lex_id = neuro_lex_ids["soma"]
            seg_group_default = get_seg_group_by_id("soma_group", cell)
        if "dend_" in group:
            neuro_lex_id = neuro_lex_ids["dend"]
            seg_group_default = get_seg_group_by_id("dendrite_group", cell)

        if seg_group is None:
            seg_group = SegmentGroup(id=group, neuro_lex_id=neuro_lex_id)
            cell.morphology.segment_groups.append(seg_group)
```

(continues on next page)

(continued from previous page)

```

seg_group.members.append(Member(segments=segment.id))
if seg_group_default:
    seg_group_default.members.append(Member(segments=segment.id))

seg_group_all.members.append(Member(segments=segment.id))

cell.morphology.segments.append(segment)
return segment

```

We call the same function multiple times to add soma, dendritic, and axonal segments to our cell. Note how the segments connect to each other to form the contiguous cell morphology.

```

# Add two soma segments
diam = 10.0
soma_0 = add_segment(cell,
                      prox=[0.0, 0.0, 0.0, diam],
                      dist=[0.0, 10., 0.0, diam],
                      name="Seg0_soma_0",
                      group="soma_0")

soma_1 = add_segment(cell,
                      prox=None,
                      dist=[0.0, 10. + 10., 0.0, diam],
                      name="Seg1_soma_0",
                      parent=soma_0,
                      group="soma_0")

# Add axon segments
diam = 1.5
axon_0 = add_segment(cell,
                      prox=[0.0, 0.0, 0.0, diam],
                      dist=[0.0, -75, 0.0, diam],
                      name="Seg0_axon_0",
                      parent=soma_0,
                      fraction_along=0.0,
                      group="axon_0")
axon_1 = add_segment(cell,
                      prox=None,
                      dist=[0.0, -150, 0.0, diam],
                      name="Seg1_axon_0",
                      parent=axon_0,
                      group="axon_0")

# Add 2 dendrite segments

diam = 3.0
dend_0_0 = add_segment(cell,
                       prox=[0.0, 20, 0.0, diam],
                       dist=[100, 120, 0.0, diam],
                       name="Seg0_dend_0",
                       parent=soma_1,
                       fraction_along=1,
                       group="dend_0")

dend_1_0 = add_segment(cell,
                       prox=None,

```

(continues on next page)

(continued from previous page)

```

        dist=[177, 197, 0.0, diam],
        name="Seg1_dend_0",
        parent=dend_0_0,
        fraction_along=1,
        group="dend_0")

dend_0_1 = add_segment(cell,
                       prox=[0.0, 20, 0.0, diam],
                       dist=[-100, 120, 0.0, diam],
                       name="Seg0_dend_1",
                       parent=soma_1,
                       fraction_along=1,
                       group="dend_1")
dend_1_1 = add_segment(cell,
                       prox=None,
                       dist=[-177, 197, 0.0, diam],
                       name="Seg1_dend_1",
                       parent=dend_0_1,
                       fraction_along=1,
                       group="dend_1")

```

Next, we add extra information to our segments and organise them so that they can be correctly exported to the NEURON format for simulation later.

```

for section_name in ["soma_0", "axon_0", "dend_0", "dend_1"]:
    section_group = get_seg_group_by_id(section_name, cell)
    section_group.neuro_lex_id = 'sao864921383'

    den_seg_group = get_seg_group_by_id("dendrite_group", cell)
    den_seg_group.includes.append(Include(segment_groups="dend_0"))
    den_seg_group.includes.append(Include(segment_groups="dend_1"))
    den_seg_group.properties.append(Property(tag="color", value="0.8 0 0"))

    ax_seg_group = get_seg_group_by_id("axon_group", cell)
    ax_seg_group.includes.append(Include(segment_groups="axon_0"))
    ax_seg_group.properties.append(Property(tag="color", value="0 0.8 0"))

    soma_seg_group = get_seg_group_by_id("soma_group", cell)
    soma_seg_group.includes.append(Include(segment_groups="soma_0"))

    soma_seg_group.properties.append(Property(tag="color", value="0 0 0.8"))

```

We have now completed adding the morphological information to our cell. Next, we proceed to our *biophysical properties*, which are split into two:

- the *membrane properties*
- the *intracellular properties*

We also use a few simple helper functions defined in the `CellBuilder.py` module to add these to our cell:

```

# Other cell properties
set_init_memb_potential(cell, "-67mV")
set_resistivity(cell, "0.15 kohm_cm")
set_specific_capacitance(cell, "1.3 uF_per_cm2")

```

(continues on next page)

(continued from previous page)

```

# channels
# leak
add_channel_density(cell, nml_cell_doc,
                     cd_id="leak_all",
                     cond_density="0.01 mS_per_cm2",
                     ion_channel="leak_chan",
                     ion_chan_def_file="olm-example/leak_chan.channel.nml",
                     erev="-67mV",
                     ion="non_specific")

# HCNolm_soma
add_channel_density(cell, nml_cell_doc,
                     cd_id="HCNolm_soma",
                     cond_density="0.5 mS_per_cm2",
                     ion_channel="HCNolm",
                     ion_chan_def_file="olm-example/HCNolm.channel.nml",
                     erev="-32.9mV",
                     ion="h",
                     group="soma_group")

# Kdrfast_soma
add_channel_density(cell, nml_cell_doc,
                     cd_id="Kdrfast_soma",
                     cond_density="73.37 mS_per_cm2",
                     ion_channel="Kdrfast",
                     ion_chan_def_file="olm-example/Kdrfast.channel.nml",
                     erev="-77mV",
                     ion="k",
                     group="soma_group")

# Kdrfast_dendrite
add_channel_density(cell, nml_cell_doc,
                     cd_id="Kdrfast_dendrite",
                     cond_density="105.8 mS_per_cm2",
                     ion_channel="Kdrfast",
                     ion_chan_def_file="olm-example/Kdrfast.channel.nml",
                     erev="-77mV",
                     ion="k",
                     group="dendrite_group")

# Kdrfast_axon
add_channel_density(cell, nml_cell_doc,
                     cd_id="Kdrfast_axon",
                     cond_density="117.392 mS_per_cm2",
                     ion_channel="Kdrfast",
                     ion_chan_def_file="olm-example/Kdrfast.channel.nml",
                     erev="-77mV",
                     ion="k",
                     group="axon_group")

# KvAolm_soma
add_channel_density(cell, nml_cell_doc,
                     cd_id="KvAolm_soma",
                     cond_density="4.95 mS_per_cm2",
                     ion_channel="KvAolm",
                     ion_chan_def_file="olm-example/KvAolm.channel.nml",
                     erev="-77mV",
                     ion="k",
                     group="soma_group")

# KvAolm_dendrite
add_channel_density(cell, nml_cell_doc,

```

(continues on next page)

(continued from previous page)

```

        cd_id="KvAolm_dendrite",
        cond_density="2.8 mS_per_cm2",
        ion_channel="KvAolm",
        ion_chan_def_file="olm-example/KvAolm.channel.nml",
        erev="-77mV",
        ion="k",
        group="dendrite_group")

# Nav_soma
add_channel_density(cell, nml_cell_doc,
                     cd_id="Nav_soma",
                     cond_density="10.7 mS_per_cm2",
                     ion_channel="Nav",
                     ion_chan_def_file="olm-example/Nav.channel.nml",
                     erev="50mV",
                     ion="na",
                     group="soma_group")

# Nav_dendrite
add_channel_density(cell, nml_cell_doc,
                     cd_id="Nav_dendrite",
                     cond_density="23.4 mS_per_cm2",
                     ion_channel="Nav",
                     ion_chan_def_file="olm-example/Nav.channel.nml",
                     erev="50mV",
                     ion="na",
                     group="dendrite_group")

# Nav_axon
add_channel_density(cell, nml_cell_doc,
                     cd_id="Nav_axon",
                     cond_density="17.12 mS_per_cm2",
                     ion_channel="Nav",
                     ion_chan_def_file="olm-example/Nav.channel.nml",
                     erev="50mV",
                     ion="na",
                     group="axon_group")

nml_cell_doc.cells.append(cell)

```

This completes the definition of our cell. We write it to a NeuroML file, and validate it.

```
pynml.write_neuroml2_file(nml_cell_doc, nml_cell_file, True, True)
```

The resulting NeuroML file is:

```

<neuroml xmlns="http://www.neuroml.org/schema/neuroml2" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.neuroml.org/schema/neuroml2 https://raw.github.com/NeuroML/NeuroML2/development/Schemas/NeuroML2/NeuroML_v2.3.xsd" id="oml_cell">
    <include href="olm-example/leak_chan.channel.nml"/>
    <include href="olm-example/HCNolm.channel.nml"/>
    <include href="olm-example/Kdrfast.channel.nml"/>
    <include href="olm-example/KvAolm.channel.nml"/>
    <include href="olm-example/Nav.channel.nml"/>
    <cell id="olm">
        <morphology id="morphology">
            <segment id="0" name="Seg0_soma_0">

```

(continues on next page)

(continued from previous page)

```

<proximal x="0.0" y="0.0" z="0.0" diameter="10.0"/>
<distal x="0.0" y="10.0" z="0.0" diameter="10.0"/>
</segment>
<segment id="1" name="Seg1_soma_0">
    <parent segment="0"/>
    <distal x="0.0" y="20.0" z="0.0" diameter="10.0"/>
</segment>
<segment id="2" name="Seg0_axon_0">
    <parent segment="0" fractionAlong="0."/>
    <proximal x="0.0" y="0.0" z="0.0" diameter="1.5"/>
    <distal x="0.0" y="-75.0" z="0.0" diameter="1.5"/>
</segment>
<segment id="3" name="Seg1_axon_0">
    <parent segment="2"/>
    <distal x="0.0" y="-150.0" z="0.0" diameter="1.5"/>
</segment>
<segment id="4" name="Seg0_dend_0">
    <parent segment="1"/>
    <proximal x="0.0" y="20.0" z="0.0" diameter="3.0"/>
    <distal x="100.0" y="120.0" z="0.0" diameter="3.0"/>
</segment>
<segment id="5" name="Seg1_dend_0">
    <parent segment="4"/>
    <distal x="177.0" y="197.0" z="0.0" diameter="3.0"/>
</segment>
<segment id="6" name="Seg0_dend_1">
    <parent segment="1"/>
    <proximal x="0.0" y="20.0" z="0.0" diameter="3.0"/>
    <distal x="-100.0" y="120.0" z="0.0" diameter="3.0"/>
</segment>
<segment id="7" name="Seg1_dend_1">
    <parent segment="6"/>
    <distal x="-177.0" y="197.0" z="0.0" diameter="3.0"/>
</segment>
<segmentGroup id="all">
    <member segment="0"/>
    <member segment="1"/>
    <member segment="2"/>
    <member segment="3"/>
    <member segment="4"/>
    <member segment="5"/>
    <member segment="6"/>
    <member segment="7"/>
</segmentGroup>
<segmentGroup neuroLexId="GO:0043025" id="soma_group">
    <notes>Default soma segment group for the cell</notes>
    <property tag="color" value="0 0 0.8"/>
    <member segment="0"/>
    <member segment="1"/>
    <include segmentGroup="soma_0"/>
</segmentGroup>
<segmentGroup neuroLexId="GO:0030424" id="axon_group">
    <notes>Default axon segment group for the cell</notes>
    <property tag="color" value="0 0.8 0"/>
    <member segment="2"/>
    <member segment="3"/>

```

(continues on next page)

(continued from previous page)

```

<include segmentGroup="axon_0"/>
</segmentGroup>
<segmentGroup neuroLexId="GO:0030425" id="dendrite_group">
    <notes>Default dendrite segment group for the cell</notes>
    <property tag="color" value="0.8 0 0"/>
    <member segment="4"/>
    <member segment="5"/>
    <member segment="6"/>
    <member segment="7"/>
    <include segmentGroup="dend_0"/>
    <include segmentGroup="dend_1"/>
</segmentGroup>
<segmentGroup neuroLexId="sao864921383" id="soma_0">
    <member segment="0"/>
    <member segment="1"/>
</segmentGroup>
<segmentGroup neuroLexId="sao864921383" id="axon_0">
    <member segment="2"/>
    <member segment="3"/>
</segmentGroup>
<segmentGroup neuroLexId="sao864921383" id="dend_0">
    <member segment="4"/>
    <member segment="5"/>
</segmentGroup>
<segmentGroup neuroLexId="sao864921383" id="dend_1">
    <member segment="6"/>
    <member segment="7"/>
</segmentGroup>
</morphology>
<biophysicalProperties id="biophys">
    <membraneProperties>
        <channelDensity id="leak_all" ionChannel="leak_chan" condDensity="0.
        ↪01 mS_per_cm2" erev="-67mV" ion="non_specific"/>
        <channelDensity id="HCNolm_soma" ionChannel="HCNolm" condDensity="0.5
        ↪mS_per_cm2" erev="-32.9mV" segmentGroup="soma_group" ion="h"/>
        <channelDensity id="Kdrfast_soma" ionChannel="Kdrfast" condDensity=
        ↪"73.37 mS_per_cm2" erev="-77mV" segmentGroup="soma_group" ion="k"/>
        <channelDensity id="Kdrfast_dendrite" ionChannel="Kdrfast"_
        ↪condDensity="105.8 mS_per_cm2" erev="-77mV" segmentGroup="dendrite_group" ion="k"/>
        <channelDensity id="Kdrfast_axon" ionChannel="Kdrfast" condDensity=
        ↪"117.392 mS_per_cm2" erev="-77mV" segmentGroup="axon_group" ion="k"/>
        <channelDensity id="KvAolm_soma" ionChannel="KvAolm" condDensity="4.
        ↪95 mS_per_cm2" erev="-77mV" segmentGroup="soma_group" ion="k"/>
        <channelDensity id="KvAolm_dendrite" ionChannel="KvAolm" condDensity=
        ↪"2.8 mS_per_cm2" erev="-77mV" segmentGroup="dendrite_group" ion="k"/>
        <channelDensity id="Nav_soma" ionChannel="Nav" condDensity="10.7 mS_
        ↪per_cm2" erev="50mV" segmentGroup="soma_group" ion="na"/>
        <channelDensity id="Nav_dendrite" ionChannel="Nav" condDensity="23.4
        ↪mS_per_cm2" erev="50mV" segmentGroup="dendrite_group" ion="na"/>
        <channelDensity id="Nav_axon" ionChannel="Nav" condDensity="17.12 mS_
        ↪per_cm2" erev="50mV" segmentGroup="axon_group" ion="na"/>
        <specificCapacitance value="1.3 uF_per_cm2"/>
        <initMembPotential value="-67mV"/>
    </membraneProperties>
    <intracellularProperties>
        <resistivity value="0.15 kohm_cm"/>

```

(continues on next page)

(continued from previous page)

```
</intracellularProperties>
</biophysicalProperties>
</cell>
</neuroml>
```

We can now already inspect our cell using the NeuroML tools:

```
pynml -png olm.cell.png
...
pyNeuroML >>> Writing to: /home/asinha/Documents/02_Code/00_mine/2020-OSB/NeuroML-
↪Documentation/source/Userdocs/NML2_examples/olm.cell.png
```

This gives us a figure of the morphology of our cell:

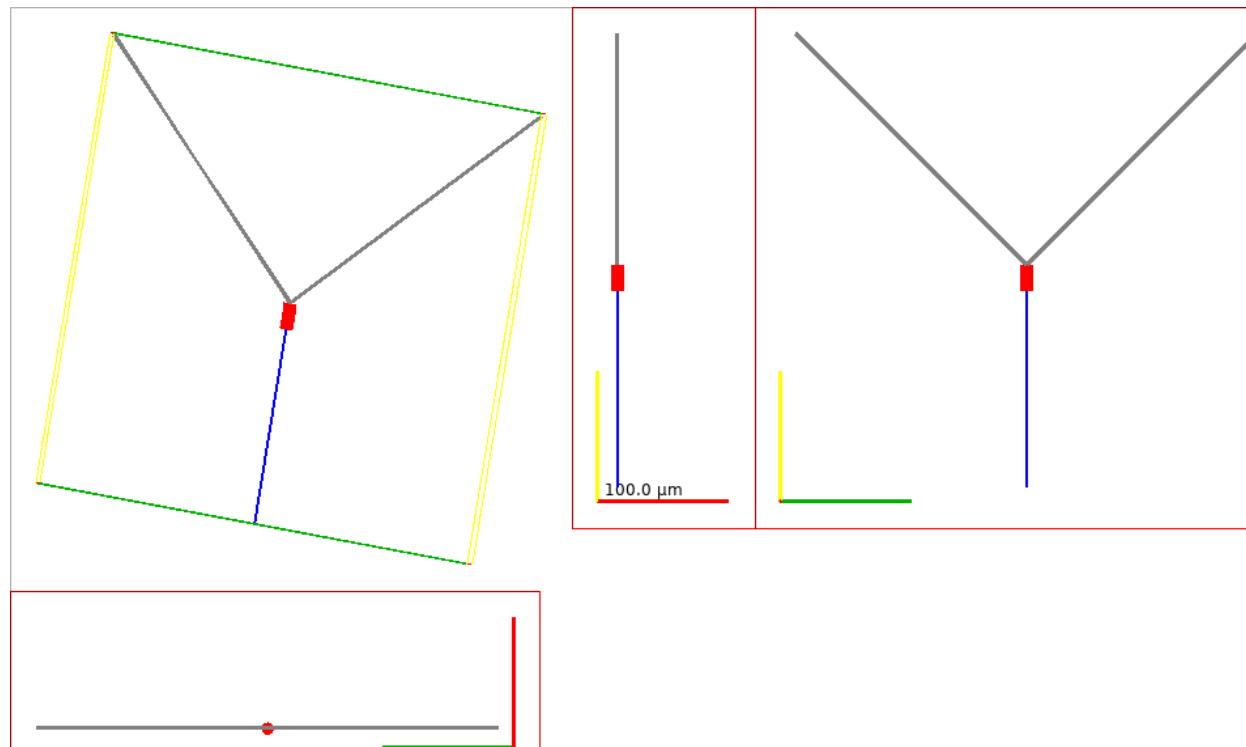


Fig. 3.16: Figure showing the morphology of the OLM cell generated from the NeuroML definition.

Declaring the network

We now use our cell in a network. Similar to our previous example, we are going to only create a network with one cell, and an *explicit input* to the cell:

```
def create_olm_network():
    """Create the network

    :returns: name of network nml file
    """
    net_doc = NeuroMLDocument(id="network",
```

(continues on next page)

(continued from previous page)

```

        notes="OLM cell network")
net_doc_fn = "olm_example_net.nml"
net_doc.includes.append(IncludeType(href=create_olm_cell()))
# Create a population: convenient to create many cells of the same type
pop = Population(id="pop0", notes="A population for our cell",
                  component="olm", size=1, type="populationList")
pop.instances.append(Instance(id=1, location=Location(0., 0., 0.)))
# Input
pulsegen = PulseGenerator(id="pg_olm", notes="Simple pulse generator", delay=
                           "100ms", duration="100ms", amplitude="0.08nA")

exp_input = ExplicitInput(target="pop0[0]", input="pg_olm")

net = Network(id="single_olm_cell_network", note="A network with a single_
population")
net_doc.pulse_generators.append(pulsegen)
net.explicit_inputs.append(exp_input)
net.populations.append(pop)
net_doc.networks.append(net)

pynml.write_neuroml2_file(nml2_doc=net_doc, nml2_file_name=net_doc_fn,_
validate=True)
return net_doc_fn

```

We start in the same way, by creating a new NeuroML document and including our cell file into it. We then create a *population* comprising of a single cell. We create a *pulse generator* as an *explicit input*, which targets our population. Note that as the schema documentation for `ExplicitInput` notes, any current source (any component that *extends basePointCurrent*) can be used as an `ExplicitInput`.

We add all of these to the *network* and save (and validate) our network file. The NeuroML file generated is below:

```

<neuroml xmlns="http://www.neuroml.org/schema/neuroml2" xmlns:xs="http://www.w3.org/
  2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"_
  xsi:schemaLocation="http://www.neuroml.org/schema/neuroml2 https://raw.github.com/
  NeuroML/NeuroML2/development/Schemas/NeuroML2/NeuroML_v2.3.xsd" id="network">
  <notes>OLM cell network</notes>
  <include href="olm.cell.nml"/>
  <pulseGenerator id="pg_olm" delay="100ms" duration="100ms" amplitude="0.08nA">
    <notes>Simple pulse generator</notes>
  </pulseGenerator>
  <network id="single_olm_cell_network">
    <population id="pop0" component="olm" size="1" type="populationList">
      <notes>A population for our cell</notes>
      <instance id="1">
        <location x="0." y="0." z="0."/>
      </instance>
    </population>
    <explicitInput target="pop0[0]" input="pg_olm"/>
  </network>
</neuroml>

```

The generated NeuroML model

Before we look at simulating the model, we can inspect our model to check for correctness. All our NeuroML files were validated when they were created already, so we do not need to run this step again. However, if required, this can be easily done:

```
pynml -validate olm*nml
```

Next, we can visualise our model using the information noted in the [visualising NeuroML models](#) page (including the `-v` verbose option for more information on the cell):

```
pynml-summary olm_example_net.nml -v
*****
* NeuroMLDocument: network
*
* ComponentType: ['Bezaire_HCNolm_tau', 'Bezaire_Kdrfast_betaq', 'Bezaire_KvAolm_taub
  ↴', 'Bezaire_Nav_alphah']
* IonChannel: ['HCNolm', 'Kdrfast', 'KvAolm', 'Nav', 'leak_chan']
* PulseGenerator: ['pg_olm']
*
* Cell: olm
*   <Segment|0|Seg0_soma_0>
*     Parent segment: None (root segment)
*     (0.0, 0.0, 0.0), diam 10.0um -> (0.0, 10.0, 0.0), diam 10.0um; seg length: 10.
  ↴0 um
*       Surface area: 314.1592653589793 um2, volume: 785.3981633974482 um3
*   <Segment|1|Seg1_soma_0>
*     Parent segment: 0
*     None -> (0.0, 20.0, 0.0), diam 10.0um; seg length: 10.0 um
*       Surface area: 314.1592653589793 um2, volume: 785.3981633974482 um3
*   <Segment|2|Seg0_axon_0>
*     Parent segment: 0
*     (0.0, 0.0, 0.0), diam 1.5um -> (0.0, -75.0, 0.0), diam 1.5um; seg length: 75.0
  ↴um
*       Surface area: 353.4291735288517 um2, volume: 132.53594007331938 um3
*   <Segment|3|Seg1_axon_0>
*     Parent segment: 2
*     None -> (0.0, -150.0, 0.0), diam 1.5um; seg length: 75.0 um
*       Surface area: 353.4291735288517 um2, volume: 132.53594007331938 um3
*   <Segment|4|Seg0_dend_0>
*     Parent segment: 1
*     (0.0, 20.0, 0.0), diam 3.0um -> (100.0, 120.0, 0.0), diam 3.0um; seg length: 141.4213562373095 um
*       Surface area: 1332.8648814475098 um2, volume: 999.6486610856323 um3
*   <Segment|5|Seg1_dend_0>
*     Parent segment: 4
*     None -> (177.0, 197.0, 0.0), diam 3.0um; seg length: 108.89444430272832 um
*       Surface area: 1026.3059587145826 um2, volume: 769.7294690359369 um3
*   <Segment|6|Seg0_dend_1>
*     Parent segment: 1
*     (0.0, 20.0, 0.0), diam 3.0um -> (-100.0, 120.0, 0.0), diam 3.0um; seg length: 141.4213562373095 um
*       Surface area: 1332.8648814475098 um2, volume: 999.6486610856323 um3
*   <Segment|7|Seg1_dend_1>
*     Parent segment: 6
*     None -> (-177.0, 197.0, 0.0), diam 3.0um; seg length: 108.89444430272832 um
*       Surface area: 1026.3059587145826 um2, volume: 769.7294690359369 um3
```

(continues on next page)

(continued from previous page)

```

*   Total length of 8 segments: 670.6316010800756 um; total area: 6053.518558099847 um2
*
*   SegmentGroup: all, 8 member(s),      0 included group(s);      contains 8 segments in total
*   SegmentGroup: soma_group, 2 member(s),      1 included group(s);      contains 2 segments in total
*   SegmentGroup: axon_group, 2 member(s),      1 included group(s);      contains 2 segments in total
*   SegmentGroup: dendrite_group,        4 member(s),      2 included group(s);      contains 4 segments in total
*   SegmentGroup: soma_0,            2 member(s),      0 included group(s);      contains 2 segments in total
*   SegmentGroup: axon_0,            2 member(s),      0 included group(s);      contains 2 segments in total
*   SegmentGroup: dend_0,            2 member(s),      0 included group(s);      contains 2 segments in total
*   SegmentGroup: dend_1,            2 member(s),      0 included group(s);      contains 2 segments in total
*
*   Channel density: leak_all on all; conductance of 0.01 mS_per_cm2 through ion chan leak_chan with ion non_specific, erev: -67mV
*   Channel is on <Segment|0/Seg0_soma_0>, total conductance: 0.1 S_per_m2 x 3. 1415926535897934e-10 m2 = 3.1415926535897936e-11 S (31.41592653589794 pS)
*   Channel is on <Segment|1/Seg1_soma_0>, total conductance: 0.1 S_per_m2 x 3. 1415926535897934e-10 m2 = 3.1415926535897936e-11 S (31.41592653589794 pS)
*   Channel is on <Segment|2/Seg0_axon_0>, total conductance: 0.1 S_per_m2 x 3. 534291735288517e-10 m2 = 3.534291735288518e-11 S (35.34291735288518 pS)
*   Channel is on <Segment|3/Seg1_axon_0>, total conductance: 0.1 S_per_m2 x 3. 534291735288517e-10 m2 = 3.534291735288518e-11 S (35.34291735288518 pS)
*   Channel is on <Segment|4/Seg0_dend_0>, total conductance: 0.1 S_per_m2 x 1. 3328648814475097e-09 m2 = 1.3328648814475097e-10 S (133.28648814475096 pS)
*   Channel is on <Segment|5/Seg1_dend_0>, total conductance: 0.1 S_per_m2 x 1. 0263059587145826e-09 m2 = 1.0263059587145826e-10 S (102.63059587145825 pS)
*   Channel is on <Segment|6/Seg0_dend_1>, total conductance: 0.1 S_per_m2 x 1. 3328648814475097e-09 m2 = 1.3328648814475097e-10 S (133.28648814475096 pS)
*   Channel is on <Segment|7/Seg1_dend_1>, total conductance: 0.1 S_per_m2 x 1. 0263059587145826e-09 m2 = 1.0263059587145826e-10 S (102.63059587145825 pS)
*   Channel density: HCN0m_soma on soma_group; conductance of 0.5 mS_per_cm2 through ion chan HCN0m with ion h, erev: -32.9mV
*   Channel is on <Segment|0/Seg0_soma_0>, total conductance: 5.0 S_per_m2 x 3. 1415926535897934e-10 m2 = 1.5707963267948968e-09 S (1570.796326794897 pS)
*   Channel is on <Segment|1/Seg1_soma_0>, total conductance: 5.0 S_per_m2 x 3. 1415926535897934e-10 m2 = 1.5707963267948968e-09 S (1570.796326794897 pS)
*   Channel density: Kdrfast_soma on soma_group; conductance of 73.37 mS_per_cm2 through ion chan Kdrfast with ion k, erev: -77mV
*   Channel is on <Segment|0/Seg0_soma_0>, total conductance: 733.7 S_per_m2 x 3. 1415926535897934e-10 m2 = 2.3049865299388314e-07 S (230498.65299388315 pS)
*   Channel is on <Segment|1/Seg1_soma_0>, total conductance: 733.7 S_per_m2 x 3. 1415926535897934e-10 m2 = 2.3049865299388314e-07 S (230498.65299388315 pS)
*   Channel density: Kdrfast_dendrite on dendrite_group; conductance of 105.8 mS_per_cm2 through ion chan Kdrfast with ion k, erev: -77mV
*   Channel is on <Segment|4/Seg0_dend_0>, total conductance: 1058.0 S_per_m2 x 1. 3328648814475097e-09 m2 = 1.4101710445714652e-06 S (1410171.0445714653 pS)
*   Channel is on <Segment|5/Seg1_dend_0>, total conductance: 1058.0 S_per_m2 x 1. 0263059587145826e-09 m2 = 1.0858317043200284e-06 S (1085831.7043200284 pS)

```

(continues on next page)

(continued from previous page)

```

*      Channel is on <Segment/6/Seg0_dend_1>, total conductance: 1058.0 S_per_m2 x_
↳ 1.3328648814475097e-09 m2 = 1.4101710445714652e-06 S (1410171.0445714653 pS)
*      Channel is on <Segment/7/Seg1_dend_1>, total conductance: 1058.0 S_per_m2 x_
↳ 1.0263059587145826e-09 m2 = 1.0858317043200284e-06 S (1085831.7043200284 pS)
*      Channel density: Kdrfast_axon on axon_group; conductance of 117.392 mS_per_
↳ cm2 through ion chan Kdrfast with ion k, erev: -77mV
*      Channel is on <Segment/2/Seg0_axon_0>, total conductance: 1173.92 S_per_m2 x_
↳ 3.534291735288517e-10 m2 = 4.1489757538898964e-07 S (414897.57538898964 pS)
*      Channel is on <Segment/3/Seg1_axon_0>, total conductance: 1173.92 S_per_m2 x_
↳ 3.534291735288517e-10 m2 = 4.1489757538898964e-07 S (414897.57538898964 pS)
*      Channel density: KvAolm_soma on soma_group; conductance of 4.95 mS_per_
↳ cm2 through ion chan KvAolm with ion k, erev: -77mV
*      Channel is on <Segment/0/Seg0_soma_0>, total conductance: 49.5 S_per_m2 x 3.
↳ 1415926535897934e-10 m2 = 1.5550883635269477e-08 S (15550.883635269476 pS)
*      Channel is on <Segment/1/Seg1_soma_0>, total conductance: 49.5 S_per_m2 x 3.
↳ 1415926535897934e-10 m2 = 1.5550883635269477e-08 S (15550.883635269476 pS)
*      Channel density: KvAolm_dendrite on dendrite_group; conductance of 2.8 mS_
↳ per_cm2 through ion chan KvAolm with ion k, erev: -77mV
*      Channel is on <Segment/4/Seg0_dend_0>, total conductance: 28.0 S_per_m2 x 1.
↳ 3328648814475097e-09 m2 = 3.7320216680530273e-08 S (37320.21668053028 pS)
*      Channel is on <Segment/5/Seg1_dend_0>, total conductance: 28.0 S_per_m2 x 1.
↳ 0263059587145826e-09 m2 = 2.8736566844008313e-08 S (28736.566844008314 pS)
*      Channel is on <Segment/6/Seg0_dend_1>, total conductance: 28.0 S_per_m2 x 1.
↳ 3328648814475097e-09 m2 = 3.7320216680530273e-08 S (37320.21668053028 pS)
*      Channel is on <Segment/7/Seg1_dend_1>, total conductance: 28.0 S_per_m2 x 1.
↳ 0263059587145826e-09 m2 = 2.8736566844008313e-08 S (28736.566844008314 pS)
*      Channel density: Nav_soma on soma_group; conductance of 10.7 mS_per_cm2_
↳ through ion chan Nav with ion na, erev: 50mV
*      Channel is on <Segment/0/Seg0_soma_0>, total conductance: 107.0 S_per_m2 x 3.
↳ 1415926535897934e-10 m2 = 3.361504139341079e-08 S (33615.04139341079 pS)
*      Channel is on <Segment/1/Seg1_soma_0>, total conductance: 107.0 S_per_m2 x 3.
↳ 1415926535897934e-10 m2 = 3.361504139341079e-08 S (33615.04139341079 pS)
*      Channel density: Nav_dendrite on dendrite_group; conductance of 23.4 mS_per_
↳ cm2 through ion chan Nav with ion na, erev: 50mV
*      Channel is on <Segment/4/Seg0_dend_0>, total conductance: 234.0 S_per_m2 x 1.
↳ 3328648814475097e-09 m2 = 3.118903822587173e-07 S (311890.3822587173 pS)
*      Channel is on <Segment/5/Seg1_dend_0>, total conductance: 234.0 S_per_m2 x 1.
↳ 0263059587145826e-09 m2 = 2.401555943392123e-07 S (240155.59433921232 pS)
*      Channel is on <Segment/6/Seg0_dend_1>, total conductance: 234.0 S_per_m2 x 1.
↳ 3328648814475097e-09 m2 = 3.118903822587173e-07 S (311890.3822587173 pS)
*      Channel is on <Segment/7/Seg1_dend_1>, total conductance: 234.0 S_per_m2 x 1.
↳ 0263059587145826e-09 m2 = 2.401555943392123e-07 S (240155.59433921232 pS)
*      Channel density: Nav_axon on axon_group; conductance of 17.12 mS_per_cm2_
↳ through ion chan Nav with ion na, erev: 50mV
*      Channel is on <Segment/2/Seg0_axon_0>, total conductance: 171.20000000000002_
↳ S_per_m2 x 3.534291735288517e-10 m2 = 6.050707450813942e-08 S (60507.07450813943 pS)
*      Channel is on <Segment/3/Seg1_axon_0>, total conductance: 171.20000000000002_
↳ S_per_m2 x 3.534291735288517e-10 m2 = 6.050707450813942e-08 S (60507.07450813943 pS)
*
*      Specific capacitance on all: 1.3 uF_per_cm2
*      Capacitance of <Segment/0/Seg0_soma_0>, total capacitance: 0.
↳ 01300000000000001 F_per_m2 x 3.1415926535897934e-10 m2 = 4.084070449666732e-12 F_
↳ (4.084070449666732 pF)
*      Capacitance of <Segment/1/Seg1_soma_0>, total capacitance: 0.
↳ 01300000000000001 F_per_m2 x 3.1415926535897934e-10 m2 = 4.084070449666732e-12 F_
↳ (4.084070449666732 pF)

```

(continues on next page)

(continued from previous page)

```

*      Capacitance of <Segment|2|Seg0_axon_0>, total capacitance: 0.
↳ 013000000000000001 F_per_m2 x 3.534291735288517e-10 m2 = 4.594579255875073e-12 F (4.
↳ 594579255875073 pF)
*      Capacitance of <Segment|3|Seg1_axon_0>, total capacitance: 0.
↳ 013000000000000001 F_per_m2 x 3.534291735288517e-10 m2 = 4.594579255875073e-12 F (4.
↳ 594579255875073 pF)
*      Capacitance of <Segment|4|Seg0_dend_0>, total capacitance: 0.
↳ 013000000000000001 F_per_m2 x 1.3328648814475097e-09 m2 = 1.732724345881763e-11 F
↳ (17.32724345881763 pF)
*      Capacitance of <Segment|5|Seg1_dend_0>, total capacitance: 0.
↳ 013000000000000001 F_per_m2 x 1.0263059587145826e-09 m2 = 1.3341977463289574e-11 F
↳ (13.341977463289574 pF)
*      Capacitance of <Segment|6|Seg0_dend_1>, total capacitance: 0.
↳ 013000000000000001 F_per_m2 x 1.3328648814475097e-09 m2 = 1.732724345881763e-11 F
↳ (17.32724345881763 pF)
*      Capacitance of <Segment|7|Seg1_dend_1>, total capacitance: 0.
↳ 013000000000000001 F_per_m2 x 1.0263059587145826e-09 m2 = 1.3341977463289574e-11 F
↳ (13.341977463289574 pF)
*
* Network: single_olm_cell_network
*
* 1 cells in 1 populations
* Population: pop0 with 1 components of type olm
*   Locations: [(0, 0, 0), ...]
*
* 0 connections in 0 projections
*
* 0 inputs in 0 input lists
*
* 1 explicit inputs (outside of input lists)
*   Explicit Input of type pg_olm to pop0(cell 0), destination: unspecified
*
*****

```

We can check the connectivity graph of the model:

```
pynml -graph 10 olm_example_net.nml
```

which will give us this figure:

3.7.2 Simulating the model

Now that we have declared and inspected our network model and all its components, we can proceed to simulate it. We do this in the `main` function:

```

def main():
    """Main function

    Include the NeuroML model into a LEMS simulation file, run it, plot some
    data.
    """
    # Simulation bits
    sim_id = "olm_example_sim"
    simulation = LEMSSimulation(sim_id=sim_id, duration=600, dt=0.01, simulation_
    ↴seed=123)

```

(continues on next page)

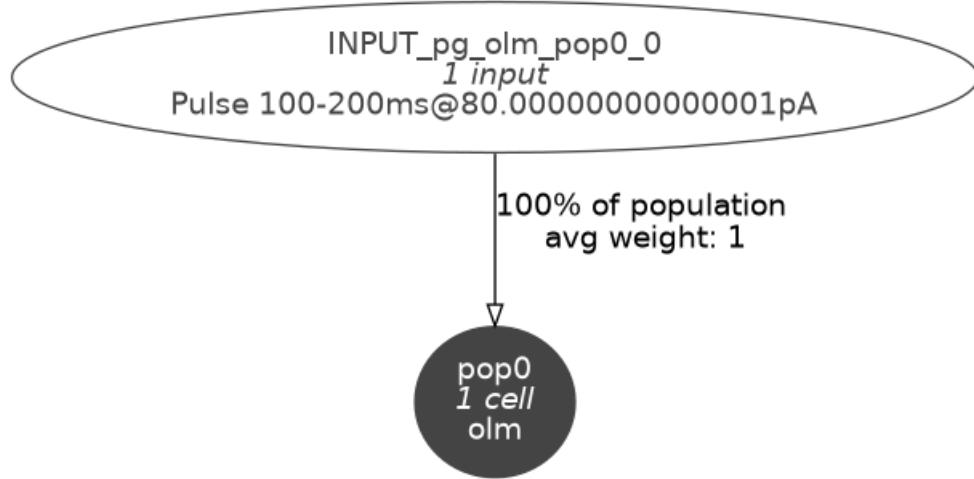


Fig. 3.17: Level 10 network graph generated by pynml

(continued from previous page)

```

# Include the NeuroML model file
simulation.include_neuroml2_file(create_olm_network())
# Assign target for the simulation
simulation.assign_simulation_target("single_olm_cell_network")

# Recording information from the simulation
simulation.create_output_file(id="output0", file_name=sim_id + ".dat")
simulation.add_column_to_output_file("output0", column_id="pop0_0_v", quantity=
"pop0[0]/v")
simulation.add_column_to_output_file("output0",
                                    column_id="pop0_0_v_Seg0_soma_0",
                                    quantity="pop0/0/olm/0/v")
simulation.add_column_to_output_file("output0",
                                    column_id="pop0_0_v_Seg1_soma_0",
                                    quantity="pop0/0/olm/1/v")
simulation.add_column_to_output_file("output0",
                                    column_id="pop0_0_v_Seg0_axon_0",
                                    quantity="pop0/0/olm/2/v")
simulation.add_column_to_output_file("output0",
                                    column_id="pop0_0_v_Seg1_axon_0",
                                    quantity="pop0/0/olm/3/v")
simulation.add_column_to_output_file("output0",
                                    column_id="pop0_0_v_Seg0_dend_0",
                                    quantity="pop0/0/olm/4/v")
simulation.add_column_to_output_file("output0",
                                    column_id="pop0_0_v_Seg1_dend_0",
                                    quantity="pop0/0/olm/6/v")
simulation.add_column_to_output_file("output0",
                                    column_id="pop0_0_v_Seg0_dend_1",
                                    quantity="pop0/0/olm/5/v")
simulation.add_column_to_output_file("output0",
                                    column_id="pop0_0_v_Seg1_dend_1",
                                    quantity="pop0/0/olm/7/v")

# Save LEMS simulation to file
sim_file = simulation.save_to_file()

```

(continues on next page)

(continued from previous page)

```
# Run the simulation using the NEURON simulator
pynml.run_lems_with_jneuroml_neuron(sim_file, max_memory="2G", nogui=True,
                                      plot=False, skip_run=False)

# Plot the data
plot_data(sim_id)
```

Here we first create a LEMSSimulation instance and include our network NeuroML file in it. We must inform LEMS what the target of the simulation is. In our case, it's the id of our network, single_olm_cell_network:

```
sim_id = "olm_example_sim"
simulation = LEMSSimulation(sim_id=sim_id, duration=600, dt=0.01, simulation_
seed=123)
# Include the NeuroML model file
simulation.include_neuroml2_file(create_olm_network())
# Assign target for the simulation
simulation.assign_simulation_target("single_olm_cell_network")
```

We also want to record some information, so we create an output file first with an id of output0:

```
simulation.create_output_file(id="output0", file_name=sim_id + ".dat")
```

Now, we can record any quantity that is exposed by NeuroML (any exposure). Here, for example, we add columns for the membrane potentials v of the different segments of the *cell*.

```
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg0_soma_0",
                                      quantity="pop0/0/olm/0/v")
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg1_soma_0",
                                      quantity="pop0/0/olm/1/v")
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg0_axon_0",
                                      quantity="pop0/0/olm/2/v")
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg1_axon_0",
                                      quantity="pop0/0/olm/3/v")
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg0_dend_0",
                                      quantity="pop0/0/olm/4/v")
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg1_dend_0",
                                      quantity="pop0/0/olm/6/v")
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg0_dend_1",
                                      quantity="pop0/0/olm/5/v")
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg1_dend_1",
                                      quantity="pop0/0/olm/7/v")
```

The path required to point to the quantity (exposure) to be recorded needs to be correctly provided. Here, where we use a *population list* that includes an *instance* of the cell, it is: population_id/instance_id/cell component type/segment id/exposure. (See tickets 15 and 16)

We then save the LEMS simulation file, and run our simulation with the *NEURON* simulator (since the default *jNeuroML* simulator can only simulate single compartment cells).

```
# Run the simulation using the NEURON simulator
pynml.run_lems_with_jneuroml_neuron(sim_file, max_memory="2G", nogui=True,
                                      plot=False, skip_run=False)
```

3.7.3 Plotting the recorded variables

To plot the variables that we recorded, we write a simple function that reads the data and uses the generate_plot utility function which generates the membrane potential graphs for different segments.

```
def plot_data(sim_id):
    """Plot the sim data.

    Load the data from the file and plot the graph for the membrane potential
    using the pynml generate_plot utility function.

    :sim_id: ID of simulation

    """
    data_array = np.loadtxt(sim_id + ".dat")
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 1]], "Membrane potential"
    ↪(soma seg 0)", show_plot_already=False, save_figure_to=sim_id + "_seg0_soma0-v.png",
    ↪xaxis="time (s)", yaxis="membrane potential (V)")
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 2]], "Membrane potential"
    ↪(soma seg 1)", show_plot_already=False, save_figure_to=sim_id + "_seg1_soma0-v.png",
    ↪xaxis="time (s)", yaxis="membrane potential (V)")
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 3]], "Membrane potential"
    ↪(axon seg 0)", show_plot_already=False, save_figure_to=sim_id + "_seg0_axon0-v.png",
    ↪xaxis="time (s)", yaxis="membrane potential (V)")
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 4]], "Membrane potential"
    ↪(axon seg 1)", show_plot_already=False, save_figure_to=sim_id + "_seg1_axon0-v.png",
    ↪xaxis="time (s)", yaxis="membrane potential (V)")
```

This concludes this example. Here we have seen how to create, simulate, record, and visualise a multi-compartment neuron. In the next section, you will find an interactive notebook where you can play with this example.

3.8 Interactive multi-compartment OLM cell example

To run this interactive Jupyter Notebook, please click on the rocket icon  in the top panel. For more information, please see [how to use this documentation](#). Please uncomment the line below if you use the Google Colab. (It does not include these packages by default).

```
#%pip install pyneuroml neuromllite NEURON
```

```
#!/usr/bin/env python3
"""
Multi-compartmental OLM cell example

File: olm-example.py

Copyright 2021 NeuroML contributors
Authors: Padraig Gleeson, Ankur Sinha
```

(continues on next page)

(continued from previous page)

```
"""

```

```
from neuroml import (NeuroMLDocument, IncludeType, Population, PulseGenerator,
                     ExplicitInput, Network, SegmentGroup, Member, Property, Include, Instance, Location)
from CellBuilder import (create_cell, add_segment, add_channel_density, set_init_memb_
    _potential, set_resistivity, set_specific_capacitance, get_seg_group_by_id)
from pyneuroml import pynml
from pyneuroml.lems import LEMSSimulation
import numpy as np
```

```
ModuleNotFoundError                                     Traceback (most recent call last)
Input In [2], in <cell line: 11>()
      1 #!/usr/bin/env python3
      2 """
      3 Multi-compartmental OLM cell example
      4
      (...)

      8 Authors: Padraig Gleeson, Ankur Sinha
      9 """
--> 11 from neuroml import (NeuroMLDocument, IncludeType, Population,_
    +PulseGenerator, ExplicitInput, Network, SegmentGroup, Member, Property, Include,_
    +Instance, Location)
     12 from CellBuilder import (create_cell, add_segment, add_channel_density,_
    +set_init_memb_potential, set_resistivity, set_specific_capacitance, get_seg_
    +group_by_id)
     13 from pyneuroml import pynml

ModuleNotFoundError: No module named 'neuroml'
```

The CellBuilder module file can be found in the same folder as the Python script. It is used to define the helper functions that we use in our main file.

3.8.1 Declaring the NeuroML model

Create the cell

In this example, we do not create the ion channels. We include ion channels that are already provided in NeuroML files.

```
def create_olm_cell():
    """Create the complete cell.

    :returns: cell object
    """
    nml_cell_doc = NeuroMLDocument(id="olm_cell")
    cell = create_cell("olm")
    nml_cell_file = cell.id + ".cell.nml"

    # Add two soma segments
    diam = 10.0
    soma_0 = add_segment(cell,
                         prox=[0.0, 0.0, 0.0, diam],
                         dist=[0.0, 10., 0.0, diam],
```

(continues on next page)

(continued from previous page)

```

        name="Seg0_soma_0",
        group="soma_0")

soma_1 = add_segment(cell,
                     prox=None,
                     dist=[0.0, 10. + 10., 0.0, diam],
                     name="Seg1_soma_0",
                     parent=soma_0,
                     group="soma_0")

# Add axon segments
diam = 1.5
axon_0 = add_segment(cell,
                      prox=[0.0, 0.0, 0.0, diam],
                      dist=[0.0, -75, 0.0, diam],
                      name="Seg0_axon_0",
                      parent=soma_0,
                      fraction_along=0.0,
                      group="axon_0")
axon_1 = add_segment(cell,
                      prox=None,
                      dist=[0.0, -150, 0.0, diam],
                      name="Seg1_axon_0",
                      parent=axon_0,
                      group="axon_0")

# Add 2 dendrite segments

diam = 3.0
dend_0_0 = add_segment(cell,
                       prox=[0.0, 20, 0.0, diam],
                       dist=[100, 120, 0.0, diam],
                       name="Seg0_dend_0",
                       parent=soma_1,
                       fraction_along=1,
                       group="dend_0")

dend_1_0 = add_segment(cell,
                       prox=None,
                       dist=[177, 197, 0.0, diam],
                       name="Seg1_dend_0",
                       parent=dend_0_0,
                       fraction_along=1,
                       group="dend_0")

dend_0_1 = add_segment(cell,
                       prox=[0.0, 20, 0.0, diam],
                       dist=[-100, 120, 0.0, diam],
                       name="Seg0_dend_1",
                       parent=soma_1,
                       fraction_along=1,
                       group="dend_1")
dend_1_1 = add_segment(cell,
                       prox=None,
                       dist=[-177, 197, 0.0, diam],
                       name="Seg1_dend_1",

```

(continues on next page)

(continued from previous page)

```

        parent=dend_0_1,
        fraction_along=1,
        group="dend_1")

# XXX: For segment groups to be correctly mapped to sections in NEURON,
# they must include the correct neurolex ID
for section_name in ["soma_0", "axon_0", "dend_0", "dend_1"]:
    section_group = get_seg_group_by_id(section_name, cell)
    section_group.neuro_lex_id = 'sa0864921383'

den_seg_group = get_seg_group_by_id("dendrite_group", cell)
den_seg_group.includes.append(Include(segment_groups="dend_0"))
den_seg_group.includes.append(Include(segment_groups="dend_1"))
den_seg_group.properties.append(Property(tag="color", value="0.8 0 0"))

ax_seg_group = get_seg_group_by_id("axon_group", cell)
ax_seg_group.includes.append(Include(segment_groups="axon_0"))
ax_seg_group.properties.append(Property(tag="color", value="0 0.8 0"))

soma_seg_group = get_seg_group_by_id("soma_group", cell)
soma_seg_group.includes.append(Include(segment_groups="soma_0"))

soma_seg_group.properties.append(Property(tag="color", value="0 0 0.8"))

# Other cell properties
set_init_memb_potential(cell, "-67mV")
set_resistivity(cell, "0.15 kohm_cm")
set_specific_capacitance(cell, "1.3 uF_per_cm2")

# channels
# leak
add_channel_density(cell, nml_cell_doc,
                    cd_id="leak_all",
                    cond_density="0.01 mS_per_cm2",
                    ion_channel="leak_chan",
                    ion_chan_def_file="olm-example/leak Chan.channel.nml",
                    erev="-67mV",
                    ion="non_specific")

# HCNolm_soma
add_channel_density(cell, nml_cell_doc,
                    cd_id="HCNolm_soma",
                    cond_density="0.5 mS_per_cm2",
                    ion_channel="HCNolm",
                    ion_chan_def_file="olm-example/HCNolm.channel.nml",
                    erev="-32.9mV",
                    ion="h",
                    group="soma_group")

# Kdrfast_soma
add_channel_density(cell, nml_cell_doc,
                    cd_id="Kdrfast_soma",
                    cond_density="73.37 mS_per_cm2",
                    ion_channel="Kdrfast",
                    ion_chan_def_file="olm-example/Kdrfast.channel.nml",
                    erev="-77mV",
                    ion="k",
                    group="soma_group")

```

(continues on next page)

(continued from previous page)

```

# Kdrfast_dendrite
add_channel_density(cell, nml_cell_doc,
                    cd_id="Kdrfast_dendrite",
                    cond_density="105.8 mS_per_cm2",
                    ion_channel="Kdrfast",
                    ion_chan_def_file="olm-example/Kdrfast.channel.nml",
                    erev="-77mV",
                    ion="k",
                    group="dendrite_group")

# Kdrfast_axon
add_channel_density(cell, nml_cell_doc,
                    cd_id="Kdrfast_axon",
                    cond_density="117.392 mS_per_cm2",
                    ion_channel="Kdrfast",
                    ion_chan_def_file="olm-example/Kdrfast.channel.nml",
                    erev="-77mV",
                    ion="k",
                    group="axon_group")

# KvAolm_soma
add_channel_density(cell, nml_cell_doc,
                    cd_id="KvAolm_soma",
                    cond_density="4.95 mS_per_cm2",
                    ion_channel="KvAolm",
                    ion_chan_def_file="olm-example/KvAolm.channel.nml",
                    erev="-77mV",
                    ion="k",
                    group="soma_group")

# KvAolm_dendrite
add_channel_density(cell, nml_cell_doc,
                    cd_id="KvAolm_dendrite",
                    cond_density="2.8 mS_per_cm2",
                    ion_channel="KvAolm",
                    ion_chan_def_file="olm-example/KvAolm.channel.nml",
                    erev="-77mV",
                    ion="k",
                    group="dendrite_group")

# Nav_soma
add_channel_density(cell, nml_cell_doc,
                    cd_id="Nav_soma",
                    cond_density="10.7 mS_per_cm2",
                    ion_channel="Nav",
                    ion_chan_def_file="olm-example/Nav.channel.nml",
                    erev="50mV",
                    ion="na",
                    group="soma_group")

# Nav_dendrite
add_channel_density(cell, nml_cell_doc,
                    cd_id="Nav_dendrite",
                    cond_density="23.4 mS_per_cm2",
                    ion_channel="Nav",
                    ion_chan_def_file="olm-example/Nav.channel.nml",
                    erev="50mV",
                    ion="na",
                    group="dendrite_group")

# Nav_axon
add_channel_density(cell, nml_cell_doc,

```

(continues on next page)

(continued from previous page)

```

        cd_id="Nav_axon",
        cond_density="17.12 mS_per_cm2",
        ion_channel="Nav",
        ion_chan_def_file="olm-example/Nav.channel.nml",
        erev="50mV",
        ion="na",
        group="axon_group")

nml_cell_doc.cells.append(cell)
pynml.write_neuroml2_file(nml_cell_doc, nml_cell_file, True, True)
return nml_cell_file

```

Create the network

```

def create_olm_network():
    """Create the network

    :returns: name of network nml file
    """
    net_doc = NeuroMLDocument(id="network",
                               notes="OLM cell network")
    net_doc_fn = "olm_example_net.nml"
    net_doc.includes.append(IncludeType(href=create_olm_cell()))
    # Create a population: convenient to create many cells of the same type
    pop = Population(id="pop0", notes="A population for our cell",
                      component="olm", size=1, type="populationList")
    pop.instances.append(Instance(id=1, location=Location(0., 0., 0.)))
    # Input
    pulsegen = PulseGenerator(id="pg_olm", notes="Simple pulse generator", delay=
    "100ms", duration="100ms", amplitude="0.08nA")

    exp_input = ExplicitInput(target="pop0[0]", input="pg_olm")

    net = Network(id="single_olm_cell_network", note="A network with a single_
    population")
    net_doc.pulse_generators.append(pulsegen)
    net.explicit_inputs.append(exp_input)
    net.populations.append(pop)
    net_doc.networks.append(net)

    pynml.write_neuroml2_file(nml2_doc=net_doc, nml2_file_name=net_doc_fn,
    validate=True)
    return net_doc_fn

```

3.8.2 Plot the data we record

```
def plot_data(sim_id):
    """Plot the sim data.

    Load the data from the file and plot the graph for the membrane potential
    using the pynml generate_plot utility function.

    :sim_id: ID of simulation

    """
    data_array = np.loadtxt(sim_id + ".dat")
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 1]], "Membrane potential\u2192(soma seg 0)", show_plot_already=False, save_figure_to=sim_id + "_seg0_soma0-v.png",
    → xaxis="time (s)", yaxis="membrane potential (V)")
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 2]], "Membrane potential\u2192(soma seg 1)", show_plot_already=False, save_figure_to=sim_id + "_seg1_soma0-v.png",
    → xaxis="time (s)", yaxis="membrane potential (V)")
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 3]], "Membrane potential\u2192(axon seg 0)", show_plot_already=False, save_figure_to=sim_id + "_seg0_axon0-v.png",
    → xaxis="time (s)", yaxis="membrane potential (V)")
    pynml.generate_plot([data_array[:, 0]], [data_array[:, 4]], "Membrane potential\u2192(axon seg 1)", show_plot_already=False, save_figure_to=sim_id + "_seg1_axon0-v.png",
    → xaxis="time (s)", yaxis="membrane potential (V)")
```

3.8.3 Create and run the simulation

```
def main():
    """Main function

    Include the NeuroML model into a LEMS simulation file, run it, plot some
    data.

    """
    # Simulation bits
    sim_id = "olm_example_sim"
    simulation = LEMSSimulation(sim_id=sim_id, duration=600, dt=0.01, simulation_
    → seed=123)
    # Include the NeuroML model file
    simulation.include_neuroml2_file(create_olm_network())
    # Assign target for the simulation
    simulation.assign_simulation_target("single_olm_cell_network")

    # Recording information from the simulation
    simulation.create_output_file(id="output0", file_name=sim_id + ".dat")
    simulation.add_column_to_output_file("output0", column_id="pop0_0_v", quantity=
    → "pop0[0]/v")
    simulation.add_column_to_output_file("output0",
                                         column_id="pop0_0_v_Seg0_soma_0",
                                         quantity="pop0/0/olm/0/v")
    simulation.add_column_to_output_file("output0",
                                         column_id="pop0_0_v_Seg1_soma_0",
                                         quantity="pop0/0/olm/1/v")
    simulation.add_column_to_output_file("output0",
                                         column_id="pop0_0_v_Seg0_axon_0",
                                         quantity="pop0/0/olm/2/v")
```

(continues on next page)

(continued from previous page)

```

simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg1_axon_0",
                                      quantity="pop0/0/olm/3/v")
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg0_dend_0",
                                      quantity="pop0/0/olm/4/v")
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg1_dend_0",
                                      quantity="pop0/0/olm/6/v")
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg0_dend_1",
                                      quantity="pop0/0/olm/5/v")
simulation.add_column_to_output_file("output0",
                                      column_id="pop0_0_v_Seg1_dend_1",
                                      quantity="pop0/0/olm/7/v")
# Save LEMS simulation to file
sim_file = simulation.save_to_file()

# Run the simulation using the NEURON simulator
pynml.run_lems_with_jneuroml_neuron(sim_file, max_memory="2G", nogui=True,
                                     plot=False, skip_run=False)

# Plot the data
plot_data(sim_id)

```

```

if __name__ == "__main__":
    main()

```

```

pyNeuroML >>> Running jnml on olm.cell.nml with pre args: -validate, post args: , -in dir: ., verbose: True, report: True, exit on fail: False
pyNeuroML >>> Executing: (java -Xmx400M -jar "/usr/share/java/jNeuroML-0.10.3.jar" -validate "olm.cell.nml" ) in directory: .
pyNeuroML >>> *** Problem running command:
pyNeuroML >>>           Command 'java -Xmx400M -jar "/usr/share/java/jNeuroML-0.10.3.jar" -validate "olm.cell.nml"' returned non-zero exit status 1.
pyNeuroML >>> jNeuroML >> jNeuroML v0.10.3
pyNeuroML >>> jNeuroML >> Validating: /home/asinha/Documents/02_Code/00_mine/2020-OSB/NeuroML-Documentation/source/Userdocs/NML2_examples/olm.cell.nml
pyNeuroML >>> jNeuroML >> WARNING: An illegal reflective access operation has occurred
pyNeuroML >>> jNeuroML >> WARNING: Illegal reflective access by com.sun.xml.bind.v2.runtime.reflect.opt.Injector (file:/usr/share/java/jNeuroML-0.10.3.jar) to method java.lang.ClassLoader.defineClass(java.lang.String,byte[],int,int)
pyNeuroML >>> jNeuroML >> WARNING: Please consider reporting this to the maintainers of com.sun.xml.bind.v2.runtime.reflect.opt.Injector
pyNeuroML >>> jNeuroML >> WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
pyNeuroML >>> jNeuroML >> WARNING: All illegal access operations will be denied in a future release
pyNeuroML >>> jNeuroML >> Valid against schema
pyNeuroML >>> jNeuroML >> Test: 10005 (Segment Group used in the include element of segmentGroup should exist) failed! ... SegmentGroup: soma_group, includes: soma_0
pyNeuroML >>> jNeuroML >> Test: 10005 (Segment Group used in the include element of segmentGroup should exist) failed! ... SegmentGroup: axon_group, includes: axon_0

```

(continues on next page)

(continued from previous page)

```

pyNeuroML >>> jNeuroML >>    Test: 10005 (Segment Group used in the include_
  ↵element of segmentGroup should exist) failed! ... SegmentGroup: dendrite_group,_
  ↵includes: dend_0
pyNeuroML >>> jNeuroML >>    Test: 10005 (Segment Group used in the include_
  ↵element of segmentGroup should exist) failed! ... SegmentGroup: dendrite_group,_
  ↵includes: dend_1
pyNeuroML >>> jNeuroML >>    No warnings
pyNeuroML >>> jNeuroML >>
pyNeuroML >>> jNeuroML >>    Validated 1 files: 0 passed, 1 failed
pyNeuroML >>> jNeuroML >>
pyNeuroML >>> jNeuroML >>
pyNeuroML >>> *** Problem running command:
pyNeuroML >>>           Command 'java -Xmx400M -jar "/usr/share/java/jNeuroML-0.10.
  ↵.jar" -validate "olm_example_net.nml"' returned non-zero exit status 1.
pyNeuroML >>> jNeuroML >>    jNeuroML v0.10.3
pyNeuroML >>> jNeuroML >>    Validating: /home/asinha/Documents/02_Code/00_mine/
  ↵2020-OSB/NeuroML-Documentation/source/Userdocs/NML2_examples/olm_example_net.nml
pyNeuroML >>> jNeuroML >>    WARNING: An illegal reflective access operation has_
  ↵occurred
pyNeuroML >>> jNeuroML >>    WARNING: Illegal reflective access by com.sun.xml.bind.
  ↵v2.runtime.reflect.opt.Injector (file:/usr/share/java/jNeuroML-0.10.3.jar) to_
  ↵method java.lang.ClassLoader.defineClass(java.lang.String,byte[],int,int)
pyNeuroML >>> jNeuroML >>    WARNING: Please consider reporting this to the_
  ↵maintainers of com.sun.xml.bind.v2.runtime.reflect.opt.Injector
pyNeuroML >>> jNeuroML >>    WARNING: Use --illegal-access=warn to enable warnings_
  ↵of further illegal reflective access operations
pyNeuroML >>> jNeuroML >>    WARNING: All illegal access operations will be denied_
  ↵in a future release
pyNeuroML >>> jNeuroML >>    Valid against schema
pyNeuroML >>> jNeuroML >>    Test: 10005 (Segment Group used in the include_
  ↵element of segmentGroup should exist) failed! ... SegmentGroup: soma_group,_
  ↵includes: soma_0
pyNeuroML >>> jNeuroML >>    Test: 10005 (Segment Group used in the include_
  ↵element of segmentGroup should exist) failed! ... SegmentGroup: axon_group,_
  ↵includes: axon_0
pyNeuroML >>> jNeuroML >>    Test: 10005 (Segment Group used in the include_
  ↵element of segmentGroup should exist) failed! ... SegmentGroup: dendrite_group,_
  ↵includes: dend_0
pyNeuroML >>> jNeuroML >>    Test: 10005 (Segment Group used in the include_
  ↵element of segmentGroup should exist) failed! ... SegmentGroup: dendrite_group,_
  ↵includes: dend_1
pyNeuroML >>> jNeuroML >>    No warnings
pyNeuroML >>> jNeuroML >>
pyNeuroML >>> jNeuroML >>    Validated 1 files: 0 passed, 1 failed
pyNeuroML >>> jNeuroML >>
pyNeuroML >>> jNeuroML >>
pyNeuroML >>> Written LEMS Simulation olm_example_sim to file: LEMS_olm_example_
  ↵sim.xml
pyNeuroML >>> Generating plot: Membrane potential (soma seg 0)

```

```

/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1688:_
  ↵MatplotlibDeprecationWarning:
The set_window_title function was deprecated in Matplotlib 3.4 and will be removed_
  ↵two minor releases later. Use manager.set_window_title or GUI-specific methods_
  ↵instead.

```

(continues on next page)

(continued from previous page)

```

fig.canvas.set_window_title(title)
/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1727: UserWarning: marker is
↳ redundantly defined by the 'marker' keyword argument and the fmt string "o" (->_
↳ marker='o'). The keyword argument will take precedence.
plt.plot(xvalues[i], yvalues[i], 'o', marker=marker, markersize=markersize,
↳ linestyle=linestyle, linewidth=linewidth, label=label)
/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1688:_  

↳ MatplotlibDeprecationWarning:
The set_window_title function was deprecated in Matplotlib 3.4 and will be removed
↳ two minor releases later. Use manager.set_window_title or GUI-specific methods
↳ instead.
fig.canvas.set_window_title(title)
/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1727: UserWarning: marker is
↳ redundantly defined by the 'marker' keyword argument and the fmt string "o" (->_
↳ marker='o'). The keyword argument will take precedence.
plt.plot(xvalues[i], yvalues[i], 'o', marker=marker, markersize=markersize,
↳ linestyle=linestyle, linewidth=linewidth, label=label)

pyNeuroML >>> Saved image to olm_example_sim_seg0_soma0-v.png of plot: Membrane
↳ potential (soma seg 0)
pyNeuroML >>> Generating plot: Membrane potential (soma seg 1)
pyNeuroML >>> Saved image to olm_example_sim_seg1_soma0-v.png of plot: Membrane
↳ potential (soma seg 1)
pyNeuroML >>> Generating plot: Membrane potential (axon seg 0)

```

```

/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1688:_  

↳ MatplotlibDeprecationWarning:
The set_window_title function was deprecated in Matplotlib 3.4 and will be removed
↳ two minor releases later. Use manager.set_window_title or GUI-specific methods
↳ instead.
fig.canvas.set_window_title(title)
/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1727: UserWarning: marker is
↳ redundantly defined by the 'marker' keyword argument and the fmt string "o" (->_
↳ marker='o'). The keyword argument will take precedence.
plt.plot(xvalues[i], yvalues[i], 'o', marker=marker, markersize=markersize,
↳ linestyle=linestyle, linewidth=linewidth, label=label)
/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1688:_  

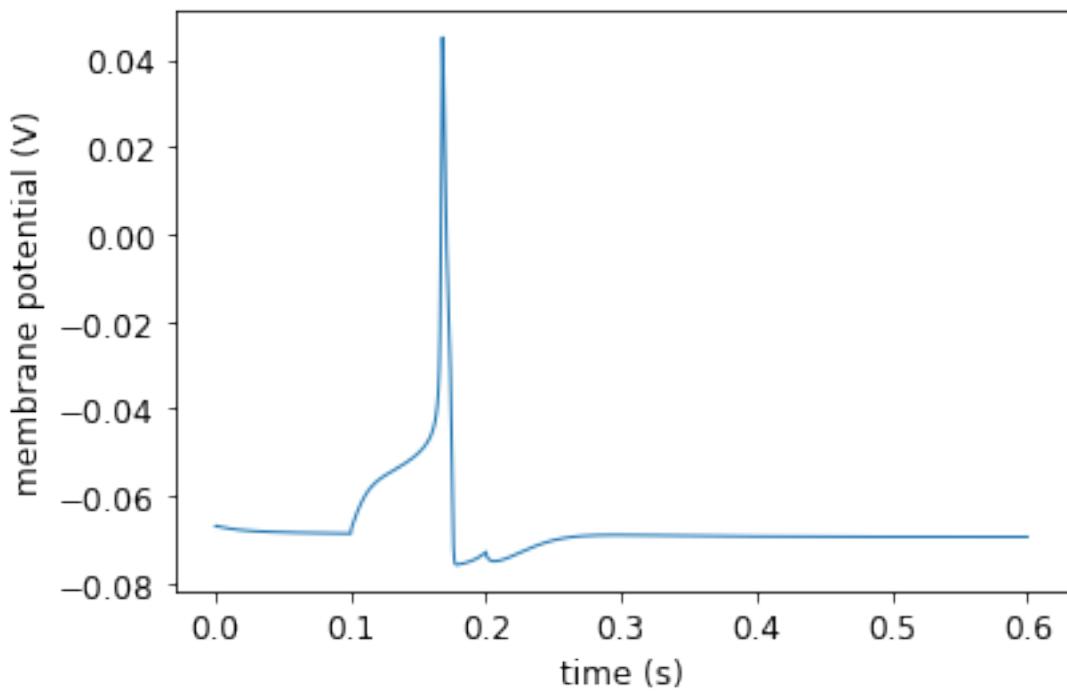
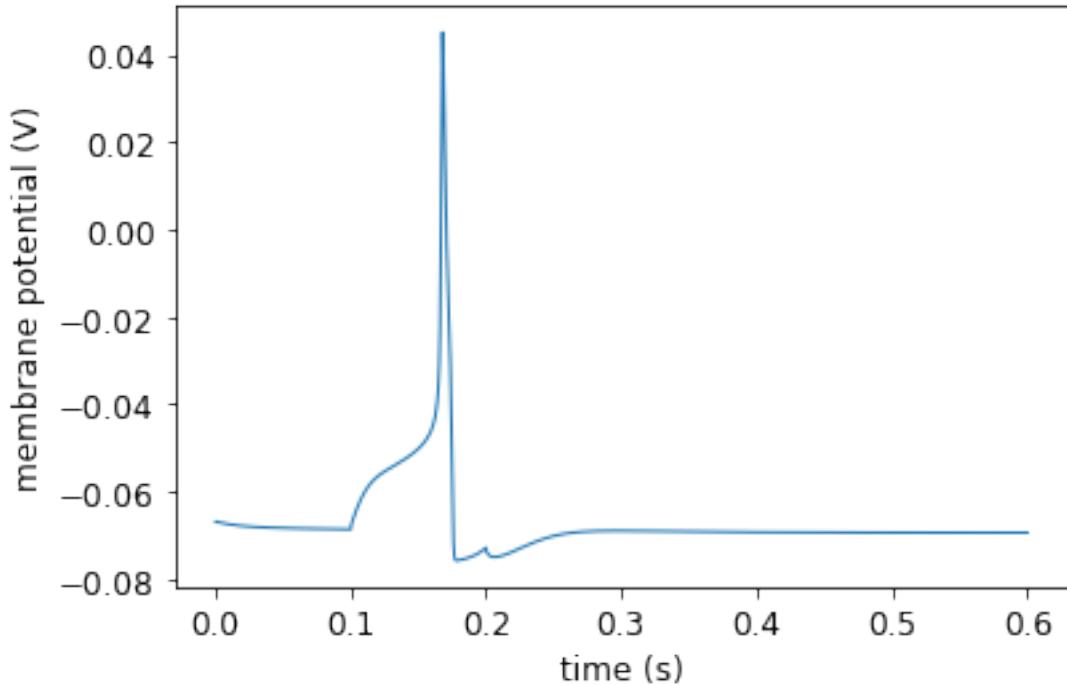
↳ MatplotlibDeprecationWarning:
The set_window_title function was deprecated in Matplotlib 3.4 and will be removed
↳ two minor releases later. Use manager.set_window_title or GUI-specific methods
↳ instead.
fig.canvas.set_window_title(title)
/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1727: UserWarning: marker is
↳ redundantly defined by the 'marker' keyword argument and the fmt string "o" (->_
↳ marker='o'). The keyword argument will take precedence.
plt.plot(xvalues[i], yvalues[i], 'o', marker=marker, markersize=markersize,
↳ linestyle=linestyle, linewidth=linewidth, label=label)

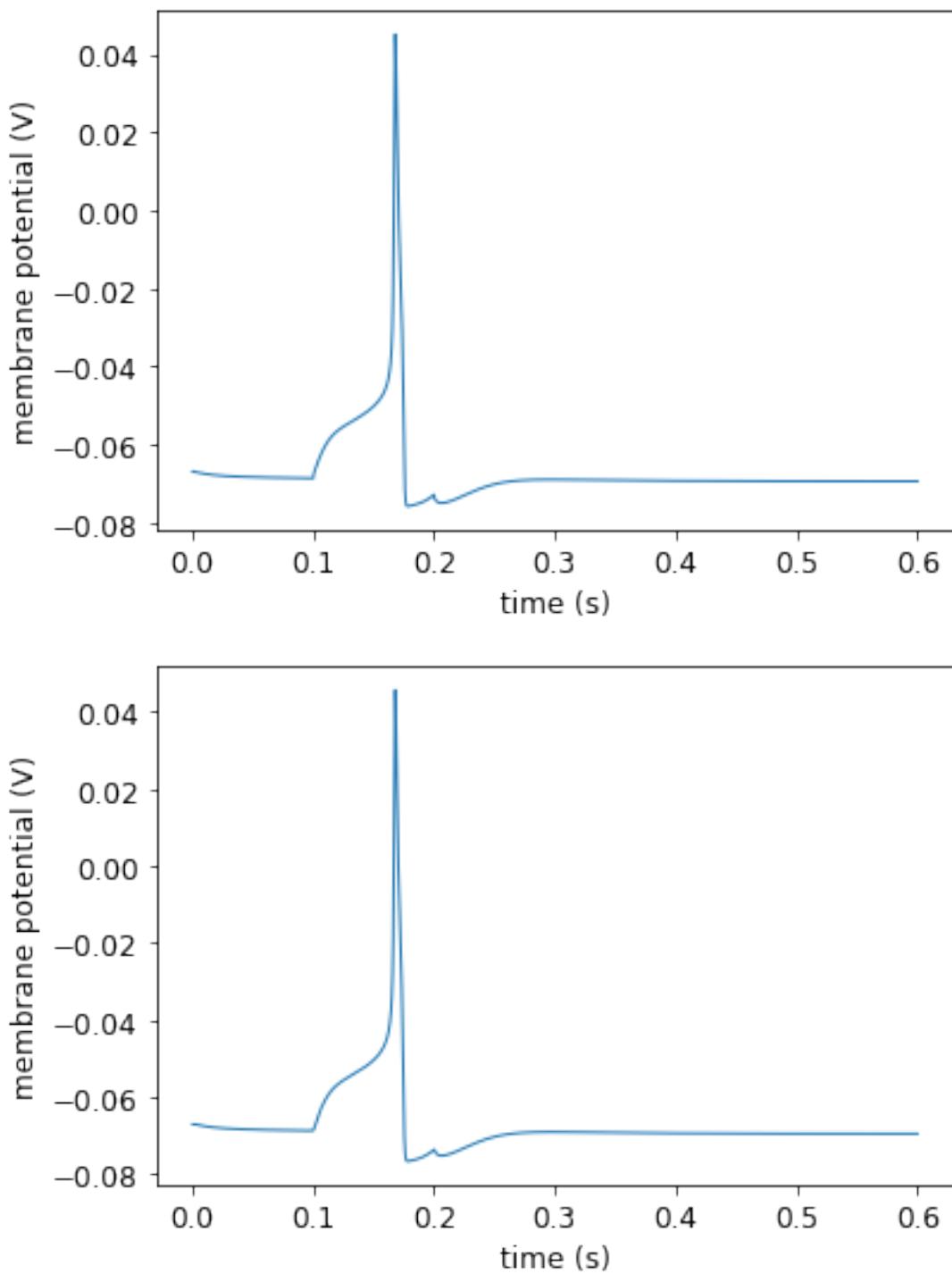
```

```

pyNeuroML >>> Saved image to olm_example_sim_seg0_axon0-v.png of plot: Membrane
↳ potential (axon seg 0)
pyNeuroML >>> Generating plot: Membrane potential (axon seg 1)
pyNeuroML >>> Saved image to olm_example_sim_seg1_axon0-v.png of plot: Membrane
↳ potential (axon seg 1)

```





FINDING AND SHARING NEUROML MODELS

There are an increasing number of repositories where you can find NeuroML models, many of which will be accepting submissions from the community who wish to share their work in this format.

4.1 NeuroML-DB: The NeuroML Database

Read the NeuroML-DB preprint!

A preprint of a manuscript describing NeuroML-DB and its current features is available [here](#).

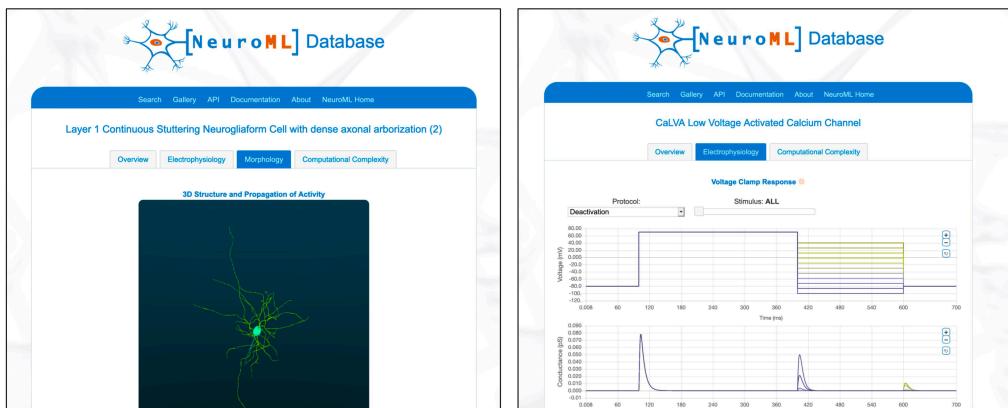


Fig. 4.1: The NeuroML Database contains NeuroML files for many [cells](#) (left above), [channels](#) (right) and synapses taken from Open Source Brain, Blue Brain Project, Allen Institute and more.

The [NeuroML Database](#) is a relational database that provides a means for sharing NeuroML model descriptions and their components. One of its goals is to contribute to an efficient tool chain for model development using NeuroML. This emphasis allows the database design and subsequent searching to take advantage of this specific format. In particular, the NeuroML database allows for efficient searches over the components of models and metadata that are associated with a hierarchical NeuroML model description.

The NeuroML Database is developed and maintained by the [ICON Lab](#) at [Arizona State University](#).

To submit your NeuroML model to NeuroML-DB, please see the information on [this page](#).

4.2 Open Source Brain

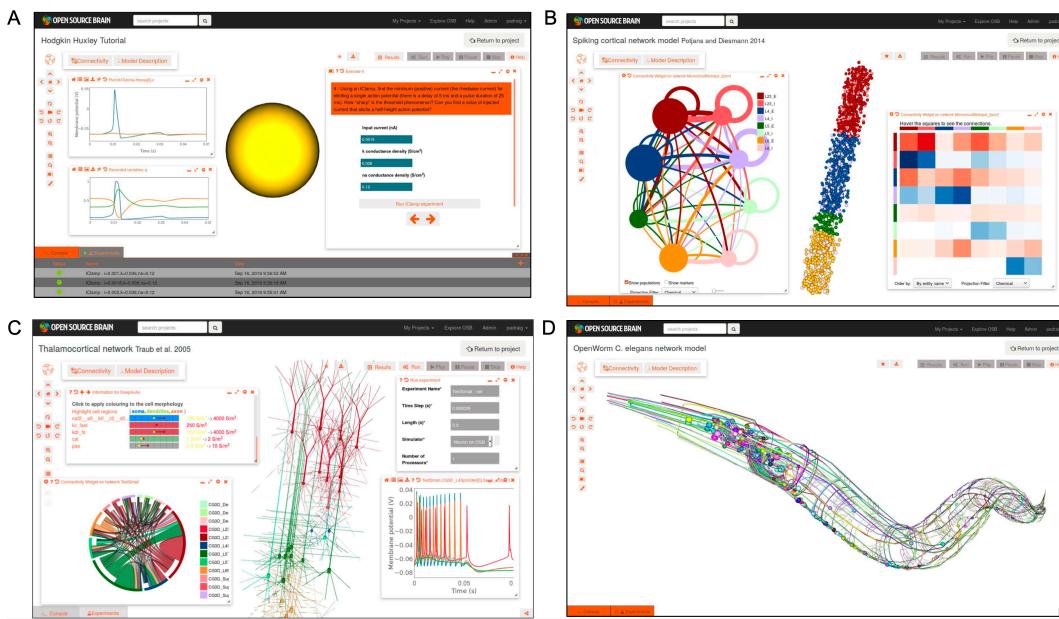


Fig. 4.2: Examples of NeuroML 2 models visualised on Open Source Brain. A) [Hodgkin Huxley model](#) interactive tutorial. B) [Integrate and fire network model](#) of cortical column ([Potjans and Diesmann 2014](#)), showing network connectivity. C) [Cortical model](#) with multicompartmental cells ([Traub et al. 2005](#)), showing network properties and simulated membrane potential activity. D) Model of *C. elegans* nervous system from [OpenWorm project](#). All visualisation/analysis/simulation enabled due to models being in standardised NeuroML format.

[Open Source Brain](#) is a platform for sharing, viewing, analyzing, and simulating standardized models from different brain regions and species.

To add your NeuroML model to Open Source Brain, please see the information on [this page](#).

4.3 Other related projects

Note: Needs introductory text.

4.4 NeuroMorpho.Org

[NeuroMorpho.Org](#) is a database of digitally reconstructed neurons. This resource can be used to retrieve reconstructed neuronal morphologies of multiple cell types from a number of species. The database can be browsed by neuron type, brain area, species, contributing lab, or cells can be searched for according to various morphometric criteria or the associated metadata.

There is a utility present on the site to view the cells in 3D (based on Robert Cannon's Cvapp), which can also save the morphologies in NeuroML 2 format.

A tutorial on getting data from [NeuroMorpho.Org](#) in NeuroML format can be found [here](#).

4.5 OpenWorm

The [OpenWorm](#) project aims to create a simulation platform to build digital in-silico living systems, starting with a *C. elegans* virtual organism simulation. The simulations and associated tools are being developed in a fully open source manner.

NeuroML is being used for the description of the 302 neurons in the [worm's nervous system](#), both for morphological description of the cells and their electrical properties.

4.6 Allen Institute

Multiple cell models as produced by the [Allen Institute](#) as part of their large scale brain modelling efforts are available in NeuroML format [here](#).

4.7 Blue Brain Project

The detailed cortical cell models from the [Blue Brain Project](#) have been converted to NeuroML format, along with the ion channels from the [Channelpedia database](#). See [here](#) for details.

CREATING NEUROML MODELS

There are 3 main ways of developing a new model of a neuronal system in NeuroML

1) Reuse elements from previous NeuroML models

There are an increasing number of resources where you can find and analyse previously developed NeuroML models to use as the basis for a new model. See [here](#) for details.

2) Writing models from scratch using Python NeuroML tools

The toolchain around NeuroML means that it is possible to create a model in NeuroML format from the start. Please see the [Getting Started with NeuroML section](#) for quick examples on how you can use [pyNeuroML](#) to create NeuroML models and run them.

3) Convert a published model developed in a simulator specific format to NeuroML

Most computational models used in publications are released in the particular format used by the authors during their research, often in a general purpose simulator like [NEURON](#). Many of these can be found on [ModelDB](#). Converting one of these to NeuroML format will mean that all further developments/modifications of the model will be standards compliant, and will give access to all of the NeuroML compliant tools for visualising/analysing/optimising/sharing the model, as well as providing multiple options for executing the model across multiple simulators.

The rest of this page is a **step by step guide** to creating a new NeuroML model based on an existing published model, verifying its behaviour, and sharing it with the community on the Open Source Brain platform.

5.1 Converting cell models to NeuroML and sharing them on Open Source Brain

The figure below is taken from the supplementary information of the [Open Source Brain paper](#), and gives a quick overview of the steps required and tools available for converting a model to NeuroML and sharing it on the OSB platform.

5.1.1 Step 1) Find the original model code

While it should in principle be possible to create the model based only on the description in the accompanying publication, having the original code will be invaluable for identifying all parameters related to the model and being able to verify the dynamical behaviour of the NeuroML equivalent against the original version.

Scripts for an increasing number of published models are available on [ModelDB](#), or the authors can be contacted to obtain the original scripts.

Verifying that these scripts reproduces some aspect of the published model by running them locally is an important first step.

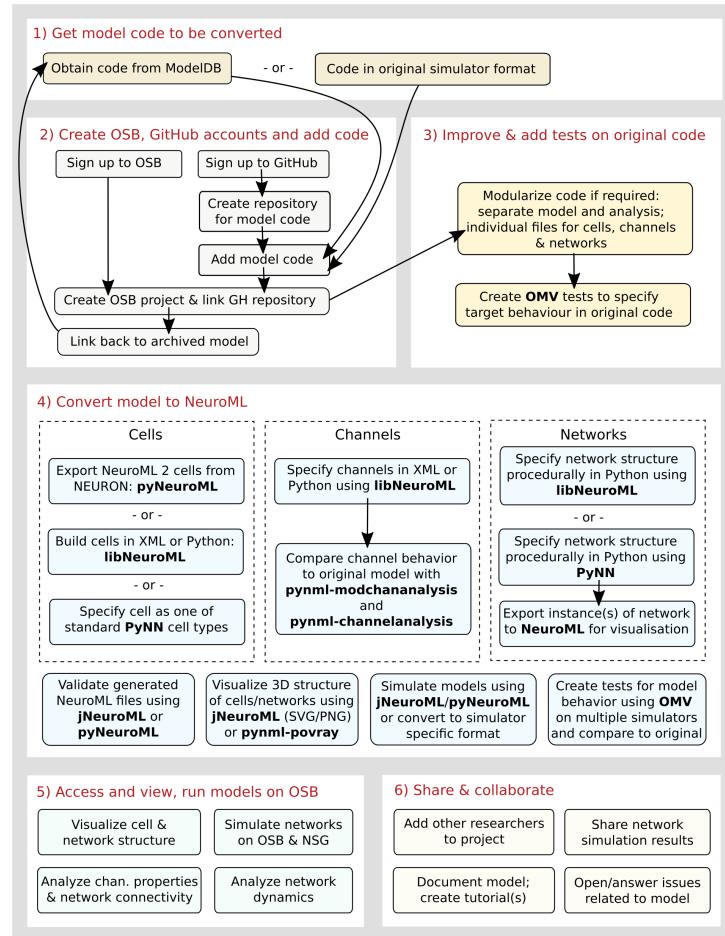


Fig. 5.1: Procedures and tools to convert models from native formats to NeuroML and PyNN (Taken from Gleeson et al. 2019 [GCM+19])

5.1.2 Step 2) Create GitHub and OSB accounts for sharing the code

2a) Sign up to GitHub and OSB

Sign up to [GitHub](#) to be able to share the updated code publicly. Next, sign up to [Open Source Brain](#), and adding a reference to your GitHub user account will help link between the two resources.

2b) Create GitHub repository

Create a new [GitHub repository](#) for your new model. There are plenty of examples of repositories containing NeuroML on [OSB](#). It's fine to share the code under your own user account, but if you would like to host it at <https://github.com/OpenSourceBrain>, please get in contact with the OSB team.

Now you can commit the scripts for original version of the model to your GitHub repository. **Please check what the license/redistribution conditions are for the code!** Authors who have shared their code on [ModelDB](#) are generally happy for the code to be reused, but it is good to get in contact with them as a courtesy to let them know your plans with the model. They will generally be very supportive as long as the original publications are referenced, and will often have useful information on any updated versions of the model. Adding or updating a [README file](#) will be valuable for anyone who comes across the model on GitHub.

2c) Create OSB project

Now you can create a project on OSB which will point to the GitHub repository and will be able to find any NeuroML models committed to it. You can also add a link back to the original archived version on ModelDB, and even reuse your README on GitHub as a description. For more details on this see [here](#).

5.1.3 Step 3) Improve and test original model code

With the original simulator code shared on GitHub, and a README updated to describe it, new users will be able to clone the repository and start using the code as shared by the authors. Some updates may be required and any changes from the original version will be recorded under the Git history visible on GitHub.

3a) Make simpler/modularised versions of original model scripts

Many of the model scripts which get released on ModelDB aim to reproduce one or two of the figures from the associated publication. However, these scripts can be quite complex, and mix simulation with some analysis of the results. They don't always provide a single, simple run of the model with standard parameters, which would be the target for a first version of the model in NeuroML.

Therefore it would be useful to create some additional scripts (reusing cell/channel definition files as much as possible) illustrating the baseline behaviour of the model, including:

- A simple script with a single cell (or one for each if multiple cells present) - applying a simple current pulse into each (e.g. [example1](#), [example2](#))
- A single compartment (soma only) example with all the ion channels (ideally one where channels can easily be added/commented out) - apply current pulse ([example](#) in NEURON)
- A passive version of multi-compartmental cell with multiple locations recorded
- A multi-compartmental cell with multiple channels and calcium dynamics, with the channels specified in separate files

These will be much easier to compare to equivalents in NeuroML.

3b) Add OMV tests

Optional, but recommended.

This step is optional, but highly recommended to create automated tests on the behaviour of the model.

Once you have some scripts which illustrate (in plots/saved data) the baseline expected behaviour of your model (spike-times, rate of firing etc.), it would be good to put some checks in place which can be run to ensure this behaviour stays consistent across changes/commits to your repository, different versions of the underlying simulator, as well as providing a target for what the NeuroML version of the model should produce.

The **Open Source Brain Model validation framework (OMV)** is designed for exactly this, allowing small scripts to be added to your repository stating what files to execute in what simulation engine and what the expected properties of generated output should be. These tests can be run on your local machine during development, but can also be easily integrated with [GitHub Actions](#), allowing tests across multiple simulators to be run every time there is a commit to the repository ([example](#)).

To start using this for your project, install OMV and test running it on your local machine (`omv all`) on some standard examples (e.g. [Hay et al.](#)).

Add OMV tests for your native simulator scripts ([example](#)), e.g. test the spike times of cell when simple current pulse applied. Commit this file to GitHub, along with a GitHub Actions workflow ([example](#)), and look for runs under the Actions tab of your project on GitHub.

Later, you can add OMV tests too for the equivalent NeuroML versions, reusing the Model Emergent Property (`*.mep`) file ([example](#)), thus testing that the behaviours of the 2 versions are the same (within a certain tolerance).

5.1.4 4) Create a version of the model in NeuroML 2

4a) Create a LEMS Simulation file to run the model

A *LEMS Simulation file* is required to specify how to run a simulation of the NeuroML model, how long to run, what to plot/save etc. Create a LEMS*.xml ([example](#)) with *.net.nml ([example](#)) and *.cell.nml ([example](#)) for **a cell with only a soma** (don't try to match a full multi-compartmental cell with all channels to the original version at this early stage).

Start off with only passive parameters (capacitance, axial resistance and 1 leak current) set; gradually add channels as in 4b) below; apply a current pulse and save soma membrane potential to file.

Ensure all *.nml files are *valid*. Ensure the LEMS*.xml runs with `jnml`; visually compare the behaviour with original simple script from the previous section.

Ensure the LEMS*.xml runs with `jnml -neuron`, producing similar behaviour. If there is a good correspondence, add OMV tests for the NeuroML version, using the Model Emergent Property (`*.mep`) file from the original script's test.

When ready, commit the LEMS/NeuroML code to GitHub.

4b) Convert channels to NeuroML

Restructure/annotate/comment channel files in the original model to be as clear as possible and ideally have all use the same overall structure (e.g. see mod files [here](#)).

(Optional) Create a (Python) script/notebook which contains the core activation variable expressions for the channels; this can be useful to restructure/test/plot/alter units of the expressions before generating the equivalent in NeuroML ([example](#)).

If you are using NEURON, use `pynml-modchananalysis` to generate plots of the activation variables for the channels in the mod files ([example1](#), [example2](#)).

Start from an existing similar example of an ion channel in NeuroML ([examples1](#), [examples2](#), [examples3](#)).

Use `pynml-channelanalysis` to generate similar plots for your NeuroML based channels as your mod channels; these can easily be plotted for adding to your GitHub repo as summary pages ([example1](#), [example2](#)).

Create a script to load the output of mod analysis and nml analysis and compare the outputs ([example](#)).

4c) Compare single compartment cell with channels

Ensure you have a passive soma example in NeuroML which reproduces the behaviour of an equivalent passive version in the original format (from steps 3a and 4a above).

Gradually test the cell with passive conductance and *each channel individually*. Plot v along with rate variables for each channel & compare how they look during current pulse ([example in NEURON](#) vs [example in NeuroML](#) and [LEMS](#))

Test these in `jnml` first, then in Neuron with `jnml -neuron`.

When you are happy with each of the channels, try the soma with all of the channels in place, with the same channel density as present in the soma of the original cell.

4d) Compare multi-compartment cell incorporating channels

If the model was created in NEURON, export the 3D morphology from the original NEURON scripts using `pyNeuroML` ([example](#)); this will be easier if there is a hoc script with just a single cell instance as in section 1). While there is the option to use `includeBiophysicalProperties=True` and this will attempt to export the conductance densities on different groups, it may be better to consolidate these and add them afterwards using correctly named groups and the most efficient representation of conductance density to group relationships ([example](#)).

```
from pyneuroml.neuron import export_to_neuroml2
..
export_to_neuroml2("test.hoc", "test.morphonly.cell.nml",  
    includeBiophysicalProperties=False)
```

Alternatively manually add the `<channelDensity>` elements to the cell file (as [here](#)).

You can use the tools for [visualising NeuroML Models](#) to compare how these versions look against the originals.

As with the single compartment example, it's best to **start off with the passive case**, i.e no active channels on the soma or dendrites, and compare that to the original code (for membrane potential at multiple locations!), and gradually add channels.

Many projects on OSB were originally converted from the original format (NEURON, GENESIS, etc.) to NeuroML v1 using `neuroConstruct` (see [here](#) for a list of these). `neuroConstruct` has good support for export to NeuroML v2, and this code could form the basis for your conversion. More on using `neuroConstruct` [here](#) and details on conversion of models to NeuroML v1 [here](#).

Note: you can also export other morphologies from [NeuroMorpho.org](#) in NeuroML2 format ([example](#)) to try out different reconstructions of the same cell type with your complement of channels.

5.1.5 4e) (Re)optimising cell models

You can use [Neurotune](#) inside pyNeuroML to re-optimize your cell models. An example is [here](#), and a full sequence of optimising a NeuroML model against data in NWB can be found [here](#).

5.1.6 4f) Create an equivalent network model in NeuroML

Creating an equivalent of a complex network model originally built in hoc for example in NeuroML is not trivial. The guide to network building with libNeuroML [here](#) is a good place to start.

See also [NeuroMLlite](#).

5.1.7 5) Access, view and run your model on OSB

When you're happy that a version of the model is behaving correctly in NeuroML, you can try visualising it on OSB.

See [here](#) for more details about viewing and simulating projects on OSB.

5.1.8 6) Share and collaborate

There is more information on how you can disseminate and promote your model once it is on OSB in the main documentation for that platform: <https://docs.opensourcebrain.org>.

Consider sharing parts of the model on [other NeuroML supporting resources](#) (e.g. cell and channel files on NeuroML-DB).

VALIDATING NEUROML MODELS

Validate NeuroML 2 files before using them.

It is good practice to validate NeuroML 2 files to check them for correctness before using them.

Models described in NeuroML must adhere to the NeuroML *specification*. This allows all NeuroML models to be checked for correctness: **validation**. There are a number of ways of validating NeuroML model files.

6.1 Using the command line tools

Both `pynml` (provided by *pyNeuroML*) and `jnml` (provided by *jNeuroML*) can validate individual NeuroML files:

Usage:

```
# For NeuroML 2
jnml -validate <NML file(s)>
pynml <NML file(s)> -validate

# For NeuroML 1 (deprecated)
jnml -validatev1 <NML file>
pynml <NML file(s)> -validatev1
```

6.2 Using the Python API

The *pyNeuroML* Python API provides a number of methods to validate NeuroML 2 files. The first is the aptly named `validate_neuroml2` function:

```
from pyneuroml.pynml import validate_neuroml2

...
validate_neuroml2(nml_filename)
```

Similarly, the `validate_neuroml1` function can be used to validate NeuroML v1 files.

If you are loading NeuroML files into your Python script, the `read_neuroml2_file` function also includes validation:

```
from pyneuroml.pynml import read_neuroml2_file

.....



read_neuroml2_file(nml_filename, include_includes=True, check_validity_pre_
include=True)
```

This will read (load) the provided NeuroML 2 file and all the files that are recursively included by it, and validate them all while it loads them.

VISUALISING NEUROML MODELS

A number of the *NeuroML software tools* can be used to easily visualise models described in NeuroML.

7.1 Get a quick summary of your model

7.1.1 Using command line tools

You can get a quick summary of your NeuroML model using the `pynml-summary` command line tool that is provided by *pyNeuroML*:

```
Usage:  
pynml-summary <NeuroML file>
```

For example, to get a quick summary of the Primary Auditory Cortex model by Dave Beeman (see it [here](#) on Open Source Brain), one can run:

```
pynml-summary MediumNet.net.nml

*****
* NeuroMLDocument: network_ACnet2
*
* PulseGenerator: ['BackgroundRandomIClamps']
*
* Network: network_ACnet2 (temperature: 6.3 degC)
*
*   60 cells in 2 populations
*     Population: baskets_12 with 12 components of type bask
*       Locations: [(372.5585, 75.3425, 459.2106), ...]
*       Properties: color=0.0 0.19921875 0.59765625;
*     Population: pyramids_48 with 48 components of type pyr_4_sym
*       Locations: [(64.2564, 0.6838, 94.8305), ...]
*       Properties: color=0.796875 0.0 0.0;
*
*   984 connections in 4 projections
*     Projection: SmallNet_bask_bask from baskets_12 to baskets_12, synapse: GABA_syn_inh
*       60 connections: [(Connection 0: 3:0(0.41661) -> 0:0(0.68577)), ...]
*     Projection: SmallNet_bask_pyr from baskets_12 to pyramids_48, synapse: GABA_syn
*       336 connections: [(Connection 0: 10:0(0.05824) -> 0:6(0.02628)), ...]
*     Projection: SmallNet_pyr_bask from pyramids_48 to baskets_12, synapse: AMPA_syn_inh
```

(continues on next page)

(continued from previous page)

```

*      252 connections: [(Connection 0: 1:0(0.89734) -> 0:1(0.09495)), ...]
*      Projection: SmallNet_pyr_pyr from pyramidal_48 to pyramidal_48, synapse: AMPA_
  ↵syn
*      336 connections: [(Connection 0: 14:0(0.52814) -> 0:3(0.10797)), ...]
*
*      14 inputs in 1 input lists
*      Input list: BackgroundRandomIClamps to pyramidal_48, component=
  ↵BackgroundRandomIClamps
*      14 inputs: [(Input 0: 37:0(0.500000)), ...]
*
*****

```

7.1.2 Using pyNeuroML

You can also get a summary of your model from within your *pyNeuroML* script itself using the `summary` function:

```

import pynml.pynml

...
pynml.pynml.summary(nml2_doc)

```

7.2 View the 3D structure of your model

7.2.1 Using command line tools

You can generate an image of the 3D structure of the NeuroML model using the `pynml` command provided by *pyNeuroML*, or using the `jnml` command provided by *jNeuroML*:

```

Usage:
pynml -png/-svg <NeuroML file>
jnml -png/-svg <NeuroML file>

```

For example, to generate a PNG image of the Auditory Cortex model used above, we can use (use `-svg` to generate a vectorised SVG image instead of a PNG):

```
pynml -png MediumNet.net.nml
```

This generates the following image showing different views of the network :

You can also generate graphical representations that can be viewed with the Persistence of Vision Raytracer (POV-Ray) tool using the `pynml-povray` tool. For example:

```

pynml-povray MediumNet.net.nml -scalez 8
povray Antialias=On Antialias_Depth=10 Antialias_Threshold=0.1 Output_to_File=y_
  ↵Output_File_Type=N Output_File_Name=Acnet-medium.povray +W1200 +H900 MediumNet.net.
  ↵nml.pov

```

generates this image:

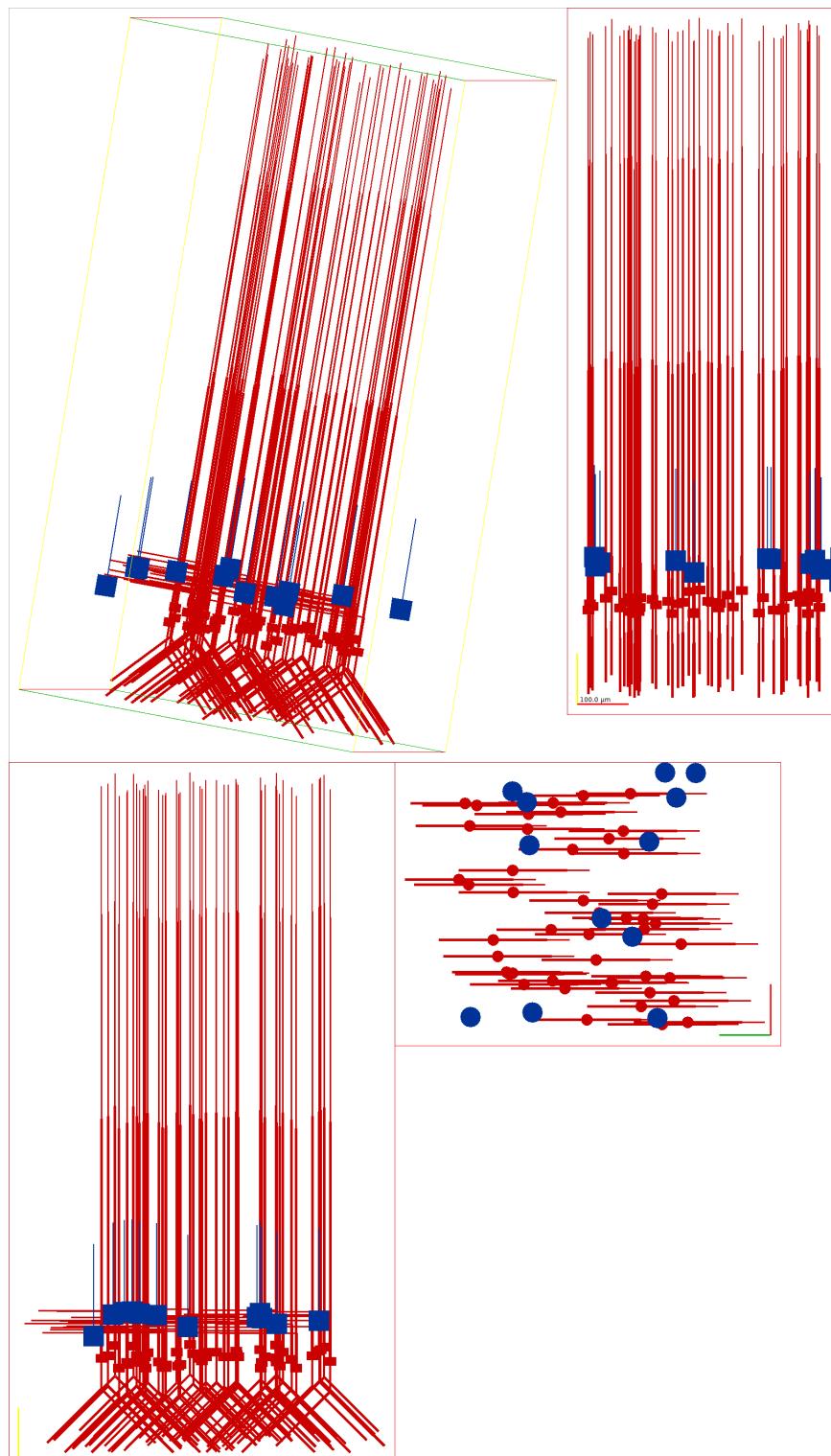


Fig. 7.1: Graphical view of the Auditory Cortex model generated with pynml

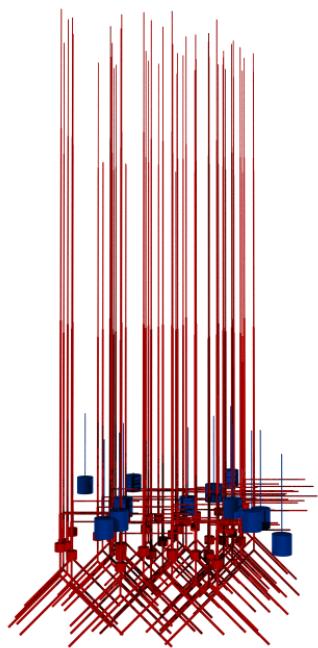


Fig. 7.2: Graphical view of the Auditory Cortex model generated with pynml-povray and POV-Ray

You can also use POV-Ray interactively. Please refer to the [official website](#) for more information on installing and using POV-Ray. On Fedora Linux systems, you can install it from the Fedora repositories using dnf:

```
sudo dnf install povray
```

7.2.2 Using pyNeuroML

You can also generate these figures from within your *pyNeuroML* script itself using the `nml2_to_png` and `nml2_to_svg` functions:

```
import pynml.pynml

...
pynml.pynml.nml2_to_png(nml2_doc)
pynml.pynml.nml2_to_svg(nml2_doc)
```

Open Source Brain uses NeuroML.

The [Open Source Brain platform](#) generates the interactive visualisations from NeuroML sources. See the Auditory Cortex model on Open Source Brain [here](#).

7.3 View the connectivity graph of your model

7.3.1 Using command line tools

Use levels to generate connectivity graphs with different levels of detail.

Positive values for levels will generate figures at the population level, while negative values will generate them at the level of cells.

You can generate an image of the 3D structure of the NeuroML model using pynml:

```
Usage:  
pynml <NeuroML file> -graph <level, engine>
```

For example, to generate a PNG image of the Auditory Cortex model used above, we can use:

```
pynml MediumNet.net.nml -graph 1d
```

This generates the following image showing different views of the network :

You can modify the level of detail included in the graph by using different values of levels. For example, this command generates a level 5 graph:

```
pynml MediumNet.net.nml -graph 5d
```



Fig. 7.3: Level 1 network graph generated by pynml

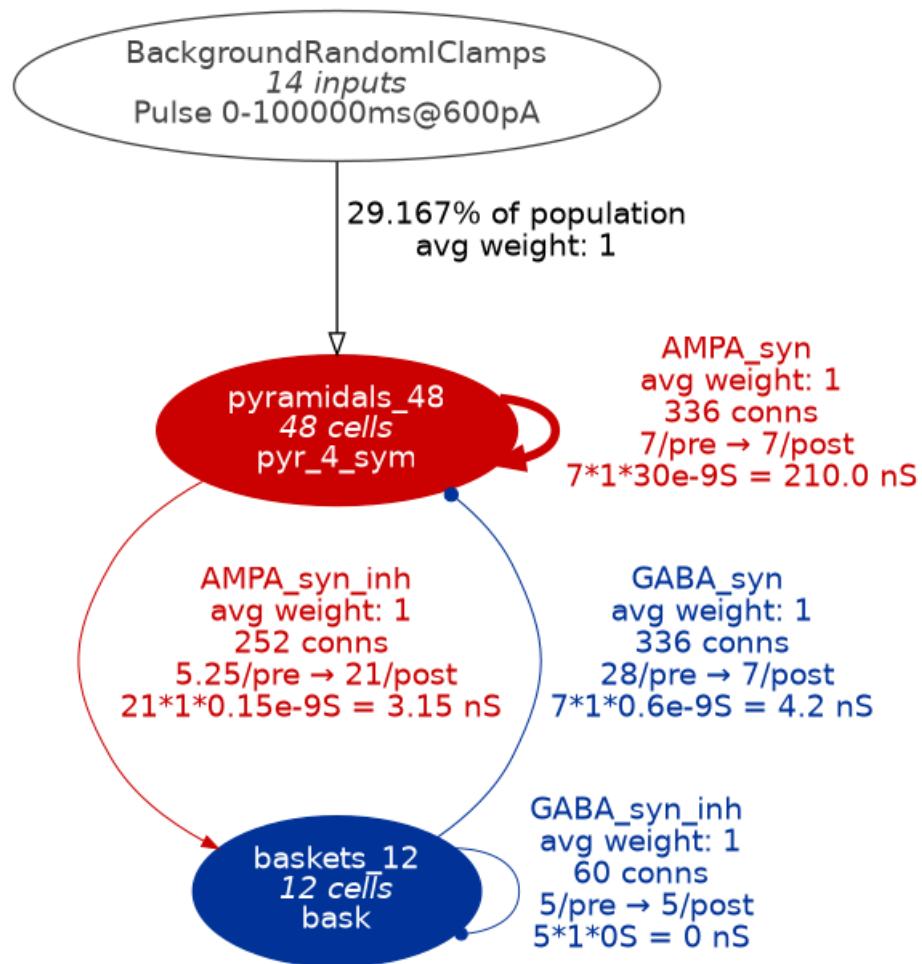


Fig. 7.4: Level 5 network graph generated by pynml

7.3.2 Using pyNeuroML

You can also generate these figures from within your *pyNeuroML* script itself using the `generate_nmlgraph` function:

```
import pynml.pynml

...
pynml.pynml.generate_nmlgraph(nml2_doc, leven="1", engine="dot")
```

7.4 View the connectivity matrices of the model

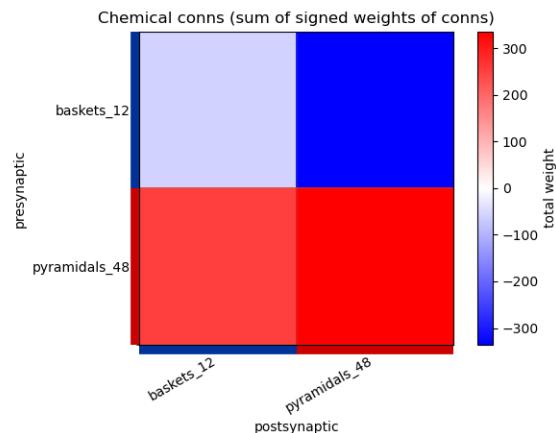
You can generate the connectivity matrices of projections between neuronal populations of the NeuroML model using `pynml`:

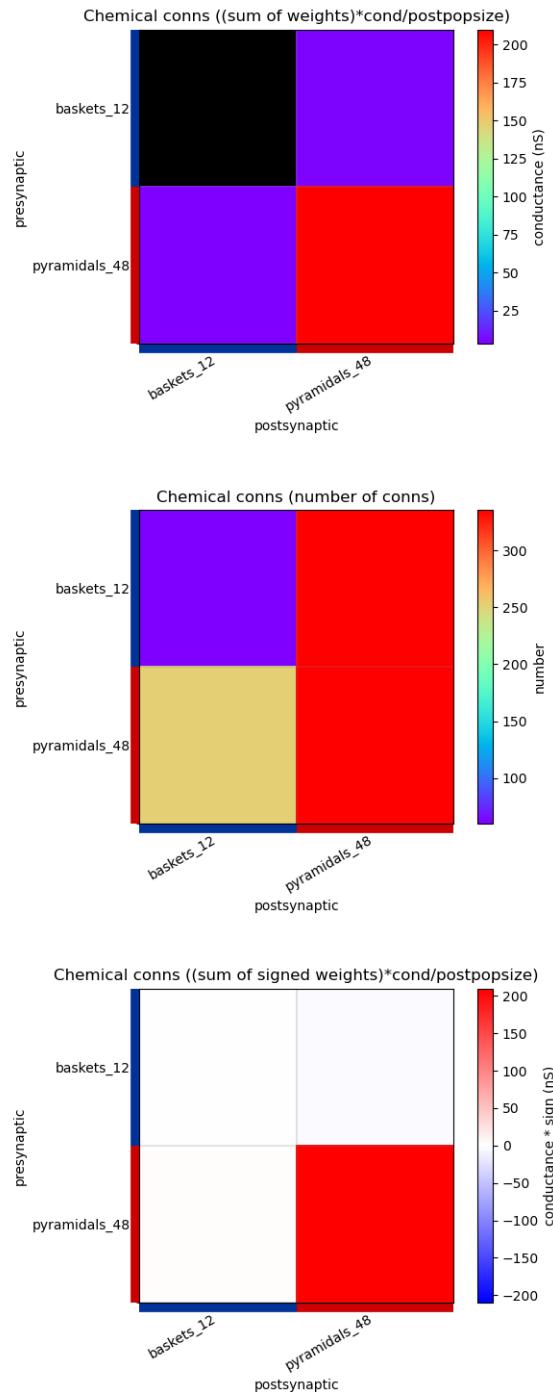
```
Usage:  
pynml <NeuroML file> -matrix <level>
```

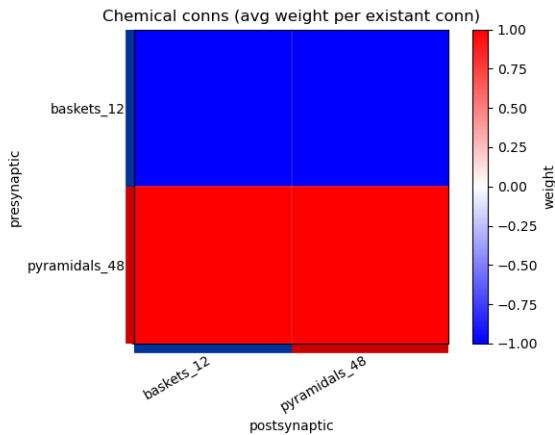
For example, to generate a PNG image of the connectivity matrices in the Auditory Cortex model used above, we can use:

```
pynml MediumNet.net.nml -matrix 1
```

This generates the following images showing different views of the connectivity matrices in the network :







7.5 View graph of the simulation instance of the model

7.5.1 Using command line tools

When you have created a simulation instance of the NeuroML model using LEMS, you can also visualise this using pynml or jnml:

```
Usage:  
pynml <LEMS simulation file> -lems-graph  
jnml <LEMS simulation file> -lems-graph
```

For example, to generate the LEMS graph for the *Izhikevich neuron network example*, we will use:

```
jnml LEMS_example_izhikevich2007network_sim.xml -lems-graph
```

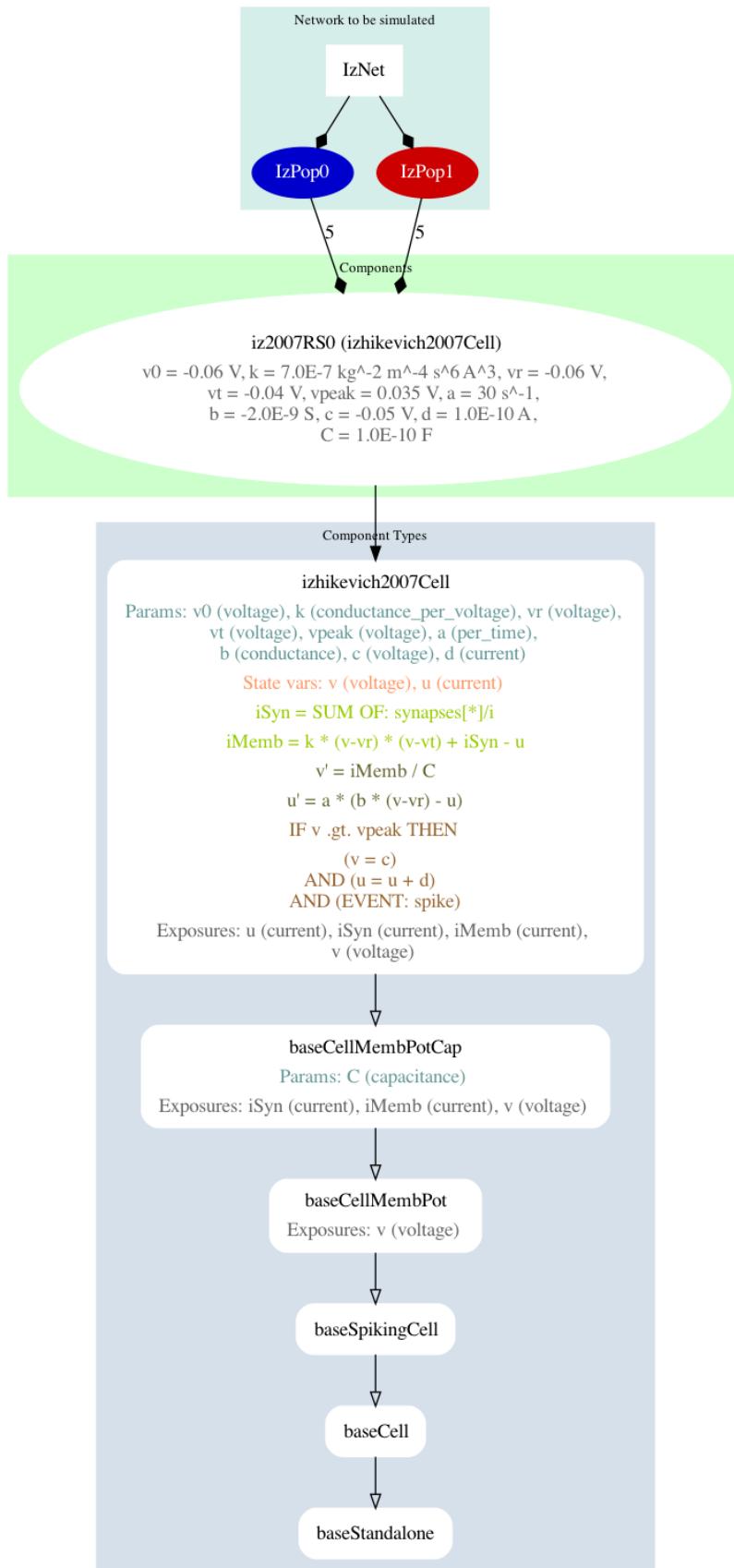
will generate:

Note that the `-lems-graph` option does not take options for levels of detail. It shows all the details of the simulation instance, and so is better suited for simpler models rather than detailed conductance based network models. For example, for the Auditory Cortex model, this very very detailed image is generated (please click to open: it is too large to display in the page).

7.5.2 Using pyNeuroML

You can also generate these figures from within your *pyNeuroML* script itself using the `generate_lemsgraph` function:

```
import pyneuroml.pynml  
...  
pyneuroml.pynml.generate_lemsgraph(lems_file)
```



7.6 Viewing/analysing ion channel dynamics

There is a dedicated section on [visualising and analysing ion channel models](#).

7.7 Visualising and analysing ion channel models

A core part of NeuroML is the ability to specify voltage dependent (and potentially concentration dependent) membrane conductances, which are due to ion channels.

7.7.1 Help converting/examining channels in NeuroML

Converting your own ion channel models to NeuroML is facilitated by examples (e.g. a simple HH Na⁺ channel) and the specification documentation (e.g. for `<IonChannelHH>`, `<gateHHrates>`, `<HHExpLinearRate>`, but there are also a number of software tools which can be used to view the internal properties of the ion channels, as well as their behaviour.

Converting cell models to NeuroML

Note: there is a full guide to [Converting cell models to NeuroML and sharing them on Open Source Brain](#) which uses some of the tools and methods below.

1) Use jnml -info (note not in pynml yet...)

jNeuroML can be used on channel files for a quick summary of the contents.

```
> jnml NaConductance.channel.nml -info

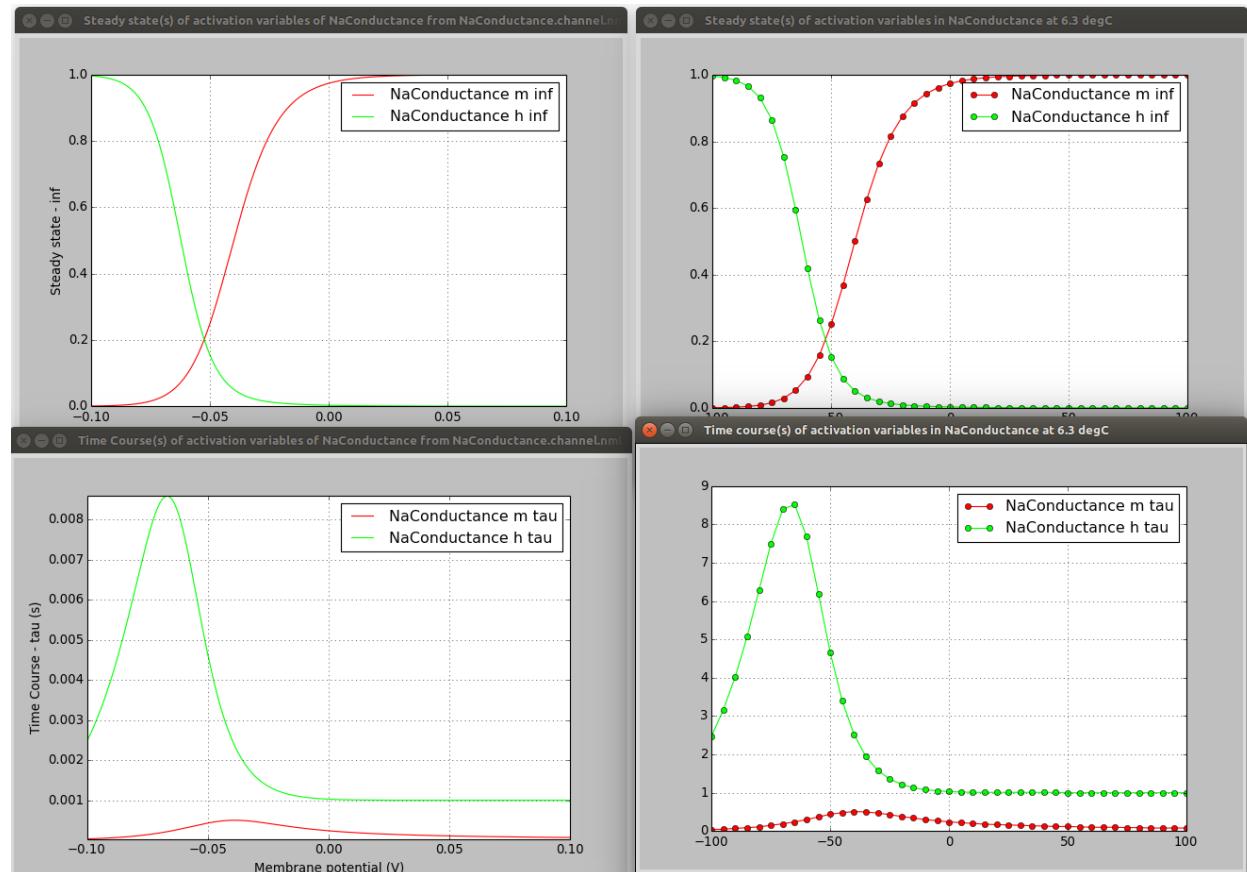
jNeuroML v0.12.0

Information on contents of NeuroML 2 file
Ion Channel NaConductance:
  ID: NaConductance
  Description: HH Na Channel
  Gates:
    gate m:
      instances: 3
      forward rate: 1e3 * (v - (-0.04)) / 0.01 / (1 - exp(-(v - (-0.04)) / 0.01))
      reverse rate: 4e3 * exp((v - (-0.065)) / -0.018)
    gate h:
      instances: 1
      forward rate: 70 * exp((v - (-0.065)) / -0.02)
      reverse rate: 1e3 / (1 + exp((v - (-0.035)) / 0.01))
```

2) Use pynml-channelanalysis in pyNeuroML

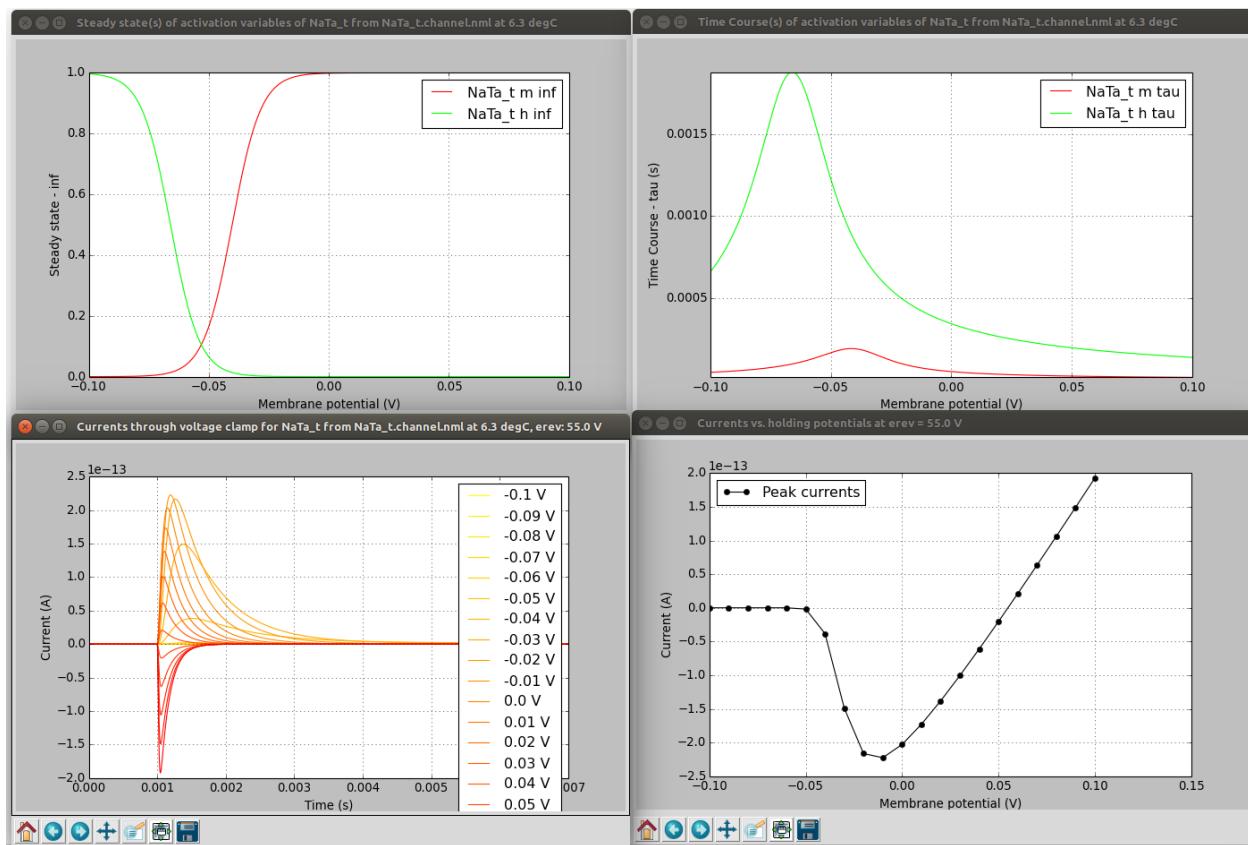
`pyNeuroML` comes with 2 utilities which help enable examination of the properties of ion channels, both based on NeuroML and NEURON mod files.

```
pynml-channelanalysis NaConductance.channel.nml      # Analyse a NeuroML 2 channel
pynml-modchananalysis NaConductance                 # Analyse a NEURON channel e.g.-
    ↪from NaConductance.mod
```

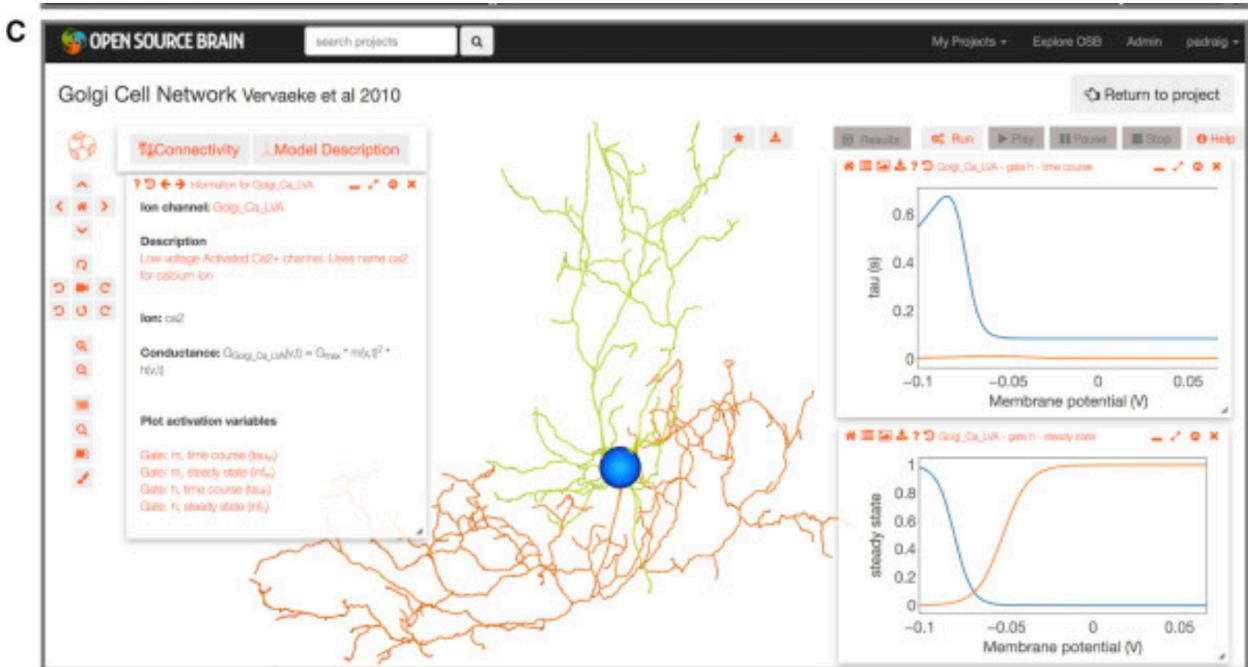


`pynml-channelanalysis` includes a number of options for generating graphs of channel activity under different conditions (see [here](#) for details).

```
pynml-channelanalysis NaTa_t.channel.nml -erev 55 -stepTargetVoltage 10 -
    ↪clampDuration 5 -i -duration 7 -clampDelay 1
```



3) Load cell model on to OSBv1 & analyse the channels...



4) Export to one of the supported simulators

Exporting to Neuron say (`jnml LEMS_NML2_Ex5_DetCell.xml -neuron`) will produce mod files with the “flattened” equations:

```
...
DERIVATIVE states {
    rates()
    m_q' = rate_m_q
    h_q' = rate_h_q
}

PROCEDURE rates() {

    m_forwardRate_x = (v - m_forwardRate_midpoint) / m_forwardRate_scale ?_
    ↪evaluable
    if (m_forwardRate_x != 0) {
        m_forwardRate_r = m_forwardRate_rate * m_forwardRate_x / (1 - exp(0 - m_
    ↪forwardRate_x)) ? evaluable cdv
    } else if (m_forwardRate_x == 0) {
        m_forwardRate_r = m_forwardRate_rate ? evaluable cdv
    }
}
...
```

Exporting to Brian 2 (`jnml LEMS_NML2_Ex5_DetCell.xml -brian2`) will also produce a large file with the explicit expressions...

```
...
hhcell_eqs=Equations(''
    dbioPhys1_membraneProperties_NaConductances_NaConductance_m_q/dt = ((bioPhys1_
    ↪membraneProperties_NaConductances_NaConductance_m_inf - bioPhys1_membraneProperties_
    ↪NaConductances_NaConductance_m_q) / bioPhys1_membraneProperties_NaConductances_
    ↪NaConductance_m_tau) : 1
    dbioPhys1_membraneProperties_NaConductances_NaConductance_h_q/dt = ((bioPhys1_
    ↪membraneProperties_NaConductances_NaConductance_h_inf - bioPhys1_membraneProperties_
    ↪NaConductances_NaConductance_h_q) / bioPhys1_membraneProperties_NaConductances_
    ↪NaConductance_h_tau) : 1
    dbioPhys1_membraneProperties_KConductances_KConductance_n_q/dt = ((bioPhys1_
    ↪membraneProperties_KConductances_KConductance_n_inf - bioPhys1_membraneProperties_
    ↪KConductances_KConductance_n_q) / bioPhys1_membraneProperties_KConductances_
    ↪KConductance_n_tau) : 1
    dv/dt = ((iChannels + iSyn) / totCap) : volt
    morph1_0_LEN = 1.0 * meter : meter
...
    bioPhys1_membraneProperties_KConductances_erev = -0.077 * volt : volt
    bioPhys1_membraneProperties_KConductances_condDensity = 360.0 * kilogram**-1 *_
    ↪meter**-4 * second**3 * amp**2 : kilogram**-1 * meter**-4 * second**3 * amp**2
    bioPhys1_membraneProperties_KConductances_KConductance_conductance = 1.0E-11 *_
    ↪siemens : siemens
    bioPhys1_membraneProperties_KConductances_KConductance_n_instances = 4.0: 1
    bioPhys1_membraneProperties_KConductances_KConductance_n_forwardRate_rate = 100.0_
    ↪* second**-1 : second**-1
    bioPhys1_membraneProperties_KConductances_KConductance_n_forwardRate_midpoint = -_
    ↪0.055 * volt : volt
    bioPhys1_membraneProperties_KConductances_KConductance_n_forwardRate_scale = 0.01_
    ↪* volt : volt
)
...
```

(continues on next page)

(continued from previous page)

```
bioPhys1_membraneProperties_KConductances_KConductance_n_reverseRate_rate = 125.0_
→ * second**-1 : second**-1
  bioPhys1_membraneProperties_KConductances_KConductance_n_reverseRate_midpoint = -
→ 0.065 * volt : volt
  bioPhys1_membraneProperties_KConductances_KConductance_n_reverseRate_scale = -0.
→ 08 * volt : volt
```

Both very verbose, but it's possible to see at least what explicit expressions are being used for the channels...

TODO: Add details of analysis options on <https://neuroml-db.org...>

SIMULATING NEUROML MODELS

Validate NeuroML 2 files before using them.

It is good practice to *validate NeuroML 2 files* to check them for correctness before simulating them.

8.1 Using Open Source Brain

Models that have already been converted to NeuroML and added to the [Open Source Brain](#) platform can be simulated through your browser.

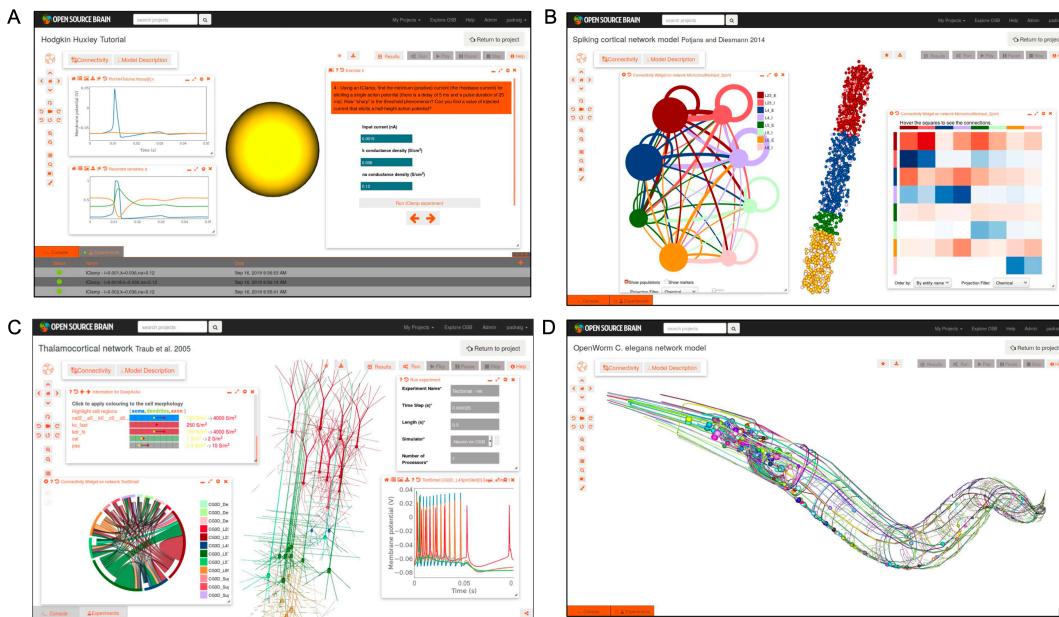


Fig. 8.1: Examples of NeuroML 2 models visualised on Open Source Brain. A) [Hodgkin Huxley model](#) interactive tutorial. B) Integrate and fire network model of cortical column (Potjans and Diesmann 2014), showing network connectivity. C) Cortical model with multicompartmental cells (Traub et al. 2005), showing network properties and simulated membrane potential activity. D) Model of *C. elegans* nervous system from [OpenWorm project](#). All visualisation/analysis/simulation enabled due to models being in standardised NeuroML format.

Most of the OSB example projects feature prebuilt NeuroML models which can be simulated in this way.

A discussion on the steps required for sharing your own models on OSB, with a view to simulating them on the platform, can be found [here](#).

8.2 Using jNeuroML/pyNeuroML

jLEMS is the reference implementation of the LEMS language in Java, and can be used to simulate single compartment models written in NeuroML/LEMS. It is included in both *jNeuroML* and *pyNeuroML*.

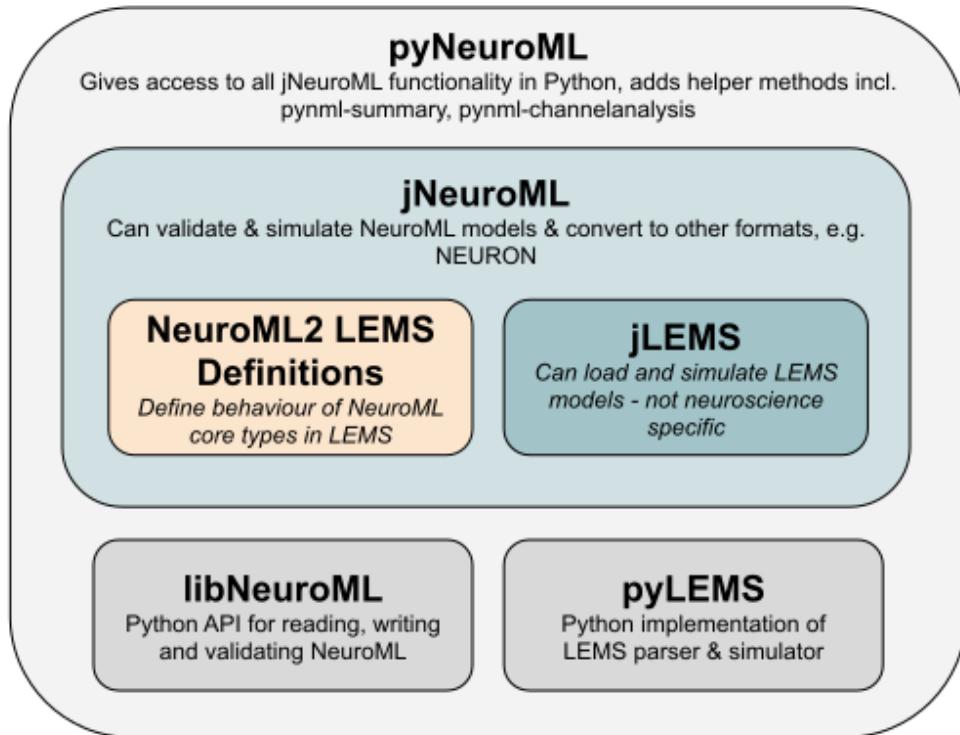


Fig. 8.2: Relationship between *jLEMS*, *jNeuroML* and *pyNeuroML*.

jNeuroML and *pyNeuroML* can be used at the command line as follows, when a *LEMS Simulation file* has been created to describe what to simulate/plot/save:

```
# Simulate the model using jNeuroML  
jnml <LEMS simulation file>  
  
# Simulate the model using pyNeuroML  
pynml <LEMS simulation file>
```

You can also run LEMS simulations using jNeuroML straight from a Python script using the *pyNeuroML* API:

```
from pyneuroml.pynml import run_lems_with_jneuroml  
  
...  
  
run_lems_with_jneuroml(lems_file_name)
```

8.3 Using NEURON

For more complex models that can not be simulated using jLEMS (e.g. incorporating multicompartmental cells), we can use the **NEURON** simulator, also using *jNeuroML* or *pyNeuroML*, pointing at a *LEMS Simulation file* describing what to simulate, and using the `-neuron` option:

```
# Simulate the model using NEURON with python/hoc/mod files generated by jNeuroML
jnml <LEMS simulation file> -neuron -run

# Simulate the model using NEURON with python/hoc/mod files generated by pyNeuroML
pynml <LEMS simulation file> -neuron -run
```

You can also run LEMS simulations using the NEURON simulator using the *pyNeuroML* API:

```
from pyneuroml.pynml import run_lems_with_jneuroml_neuron

...
run_lems_with_jneuroml_neuron(lems_file_name)
```

There is a **dedicated page on NEURON/NeuroML interactions** [here](#).

8.4 Using NetPyNE

You can also generate and run **NetPyNE** code from NeuroML. To generate and run NetPyNE code, use *jNeuroML* or *pyNeuroML*:

```
# Simulate the model using NetPyNE with python/hoc/mod files generated by jNeuroML
jnml <LEMS simulation file> -netpyne -run

# Simulate the model using NetPyNE with python/hoc/mod files generated by pyNeuroML
pynml <LEMS simulation file> -netpyne -run
```

The main generated Python file name will end in `_netpyne.py`.

You can also run LEMS simulations using the NetPyNE simulator using the *pyNeuroML* API:

```
from pyneuroml.pynml import run_lems_with_jneuroml_netpyne

...
run_lems_with_jneuroml_netpyne(lems_file_name)
```

There is a **dedicated page on NetPyNE/NeuroML interactions** [here](#).

8.5 Using Brian2

You can export single component NeuroML models to Python scripts for running them using the [Brian2](#) simulator:

```
# Using jnml
jnml <LEMS simulation file> -brian2

# Using pynml
pynml <LEMS simulation file> -brian2
```

You can also run LEMS simulations using the Brian2 simulator using the [pyNeuroML](#) API:

```
from pyneuroml.pynml import run_lems_with_jneuroml_brian2

...
run_lems_with_jneuroml_brian2(lems_file_name)
```

There is a **dedicated page on Brian/NeuroML interactions** [here](#).

8.6 Using MOOSE

You can export NeuroML models to the MOOSE simulator format using [jNeuroML](#) or [pyNeuroML](#), pointing at a [LEMS Simulation file](#) describing what to simulate, and using the `-moose` option:

```
# Using jnml
jnml <LEMS simulation file> -moose

# Using pynml
pynml <LEMS simulation file> -moose
```

There is a **dedicated page on MOOSE/NeuroML interactions** [here](#).

8.7 Using EDEN

The EDEN simulator can load and simulate NeuroML v2 models.

There is a **dedicated page on EDEN/NeuroML interactions** [here](#).

8.8 Using Arbor

You can import NeuroML models to the Arbor simulator.

There is a **dedicated page on Arbor/NeuroML interactions** [here](#).

OPTIMISING/FITTING NEUROML MODELS

pyNeuroML includes the NeuroMLTuner module for the tuning and optimisation of NeuroML models against provided data. This uses the `Neurotune` Python package for the fitting of models using evolutionary computation.

This page will walk through an example model optimisation.

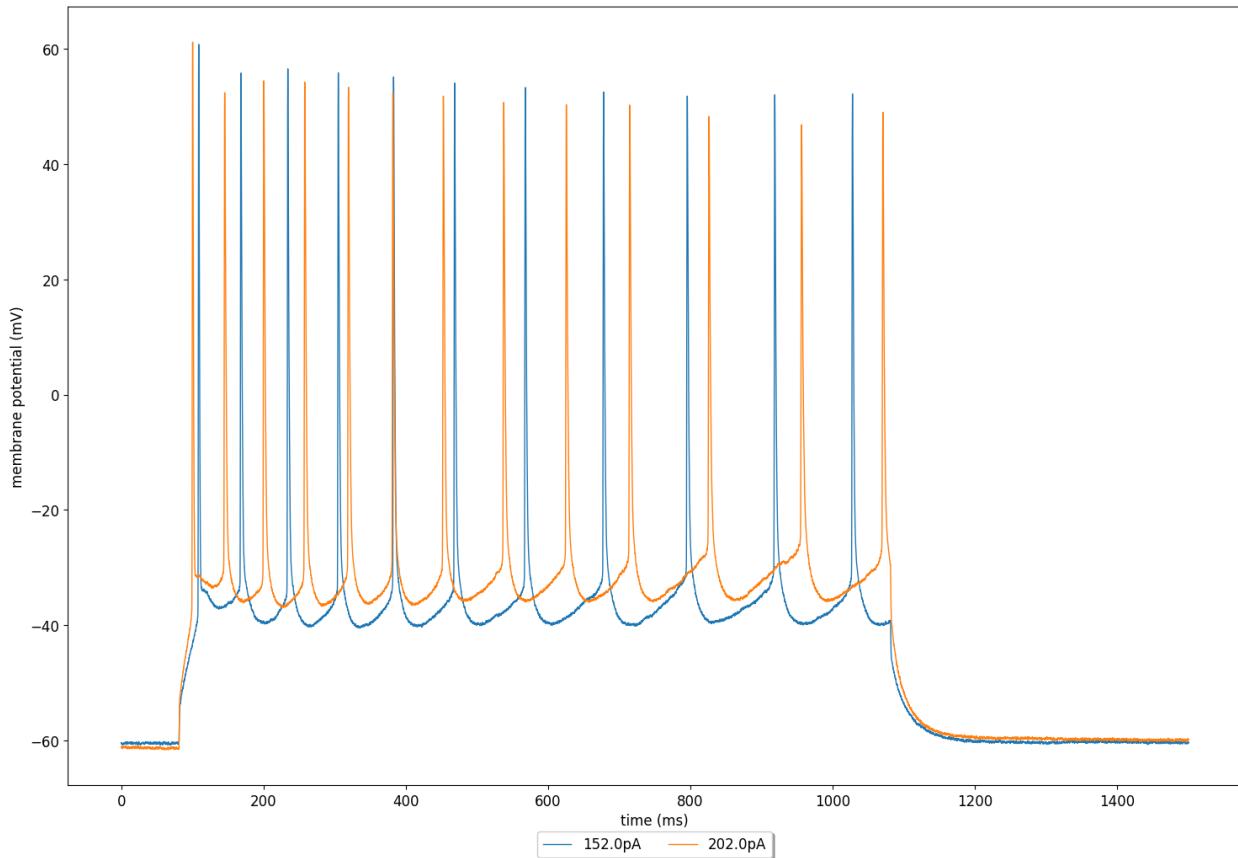


Fig. 9.1: Membrane potential from the experimental data.

The Python script used to run the optimisation and generate the graphs is given below. This can be adapted for other optimisations.

```
#!/usr/bin/env python3
"""
Example file showing the tuning of an Izhikevich neuron using pyNeuroML.
```

(continues on next page)

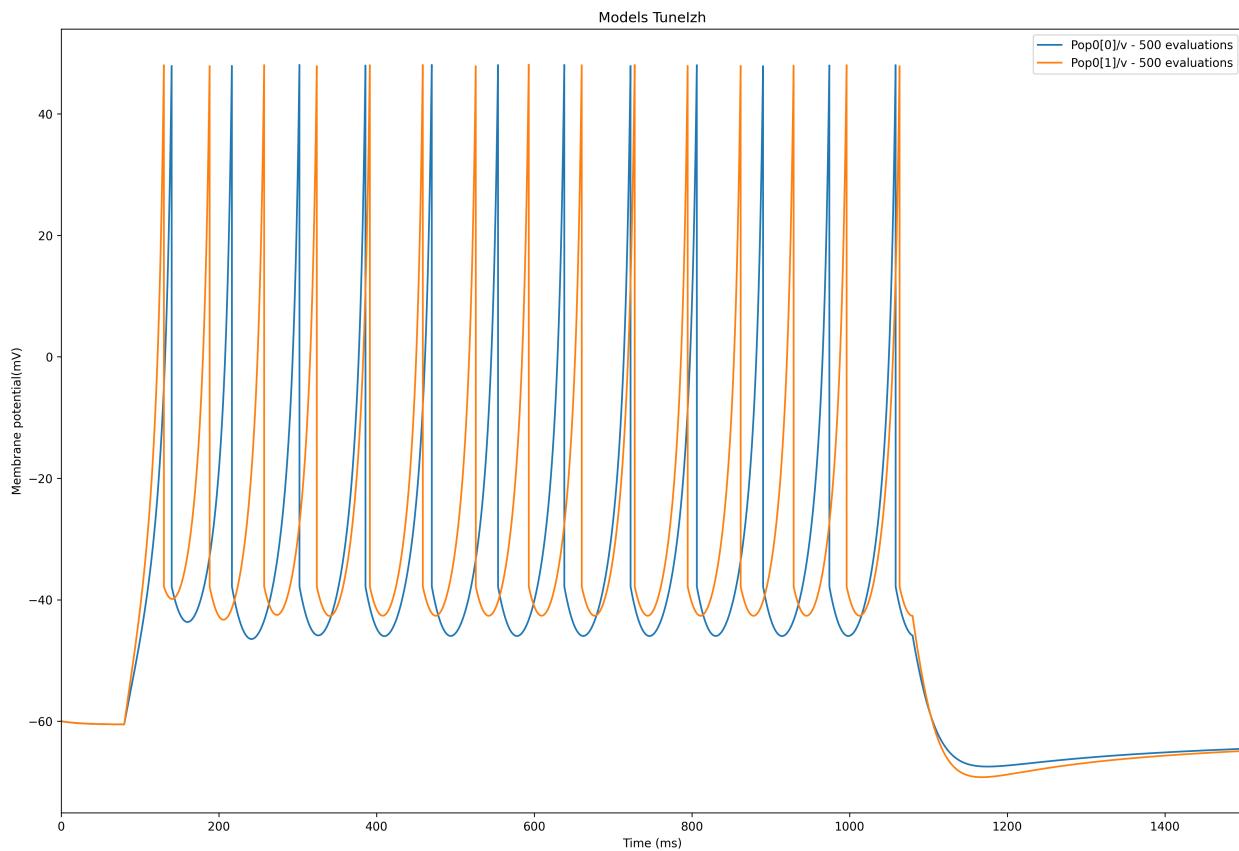


Fig. 9.2: Membrane potential obtained from the model with highest fitness.

(continued from previous page)

```

File: source/Userdocs/NML2_examples/tune-izhikevich.py

Copyright 2021 NeuroML contributors
"""

from pyneuroml.tune.NeuroMLTuner import run_optimisation
import pynwb # type: ignore
import numpy as np
from pyelectro.utils import simple_network_analysis
from typing import List, Dict, Tuple
from pyneuroml.pynml import write_neuroml2_file
from pyneuroml.pynml import generate_plot
from pyneuroml.pynml import run_lems_with_jneuroml
from neuroml import (
    NeuroMLDocument,
    Izhikevich2007Cell,
    PulseGenerator,
    Network,
    Population,
    ExplicitInput,
)
from hdmf.container import Container
from pyneuroml.lems.LEMSSimulation import LEMSSimulation

import sys

def get_data_metrics(datafile: Container) -> Tuple[Dict, Dict, Dict]:
    """Analyse the data to get metrics to tune against.

    :returns: metrics from pyelectro analysis, currents, and the membrane potential values

    """
    analysis_results = {}
    currents = {}
    memb_vals = {}
    total_acquisitions = len(datafile.acquisition)

    for acq in range(1, total_acquisitions):
        print("Going over acquisition # {}".format(acq))

        # stimulus lasts about 1000ms, so we take about the first 1500 ms
        data_v = (
            datafile.acquisition["CurrentClampSeries_{}".format(acq)].data[:15000] * 1000.0
        )
        # get sampling rate from the data
        sampling_rate = datafile.acquisition[
            "CurrentClampSeries_{}".format(acq)].rate
        # generate time steps from sampling rate
        data_t = np.arange(0, len(data_v) / sampling_rate, 1.0 / sampling_rate) * 1000.0

```

(continues on next page)

(continued from previous page)

```

# run the analysis
analysis_results[acq] = simple_network_analysis({acq: data_v}, data_t)

# extract current from description, but can be extracted from other
# locations also, such as the CurrentClampStimulus series.
data_i = (
    datafile.acquisition["CurrentClampSeries_{{:02d}}".format(acq)]
    .description.split("(")[1]
    .split("~")[1]
    .split(" ")[0]
)
currents[acq] = data_i
memb_vals[acq] = (data_t, data_v)

return (analysis_results, currents, memb_vals)

def tune_izh_model(acq_list: List, metrics_from_data: Dict, currents: Dict) -> Dict:
    """Tune networks model against the data.

    Here we generate a network with the necessary number of Izhikevich cells,
    one for each current stimulus, and tune them against the experimental data.

    :param acq_list: list of indices of acquisitions/sweeps to tune against
    :type acq_list: list
    :param metrics_from_data: dictionary with the sweep number as index, and
        the dictionary containing metrics generated from the analysis
    :type metrics_from_data: dict
    :param currents: dictionary with sweep number as index and stimulus current
        value
    """
    # length of simulation of the cells---should match the length of the
    # experiment
    sim_time = 1500.0
    # Create a NeuroML template network simulation file that we will use for
    # the tuning
    template_doc = NeuroMLDocument(id="IzhTuneNet")
    # Add an Izhikevich cell with some parameters to the document
    template_doc.izhikevich2007_cells.append(
        Izhikevich2007Cell(
            id="Izh2007",
            C="100pF",
            v0="-60mV",
            k="0.7nS_per_mV",
            vr="-60mV",
            vt="-40mV",
            vpeak="35mV",
            a="0.03per_ms",
            b="-2nS",
            c="-50.0mV",
            d="100pA",
        )
    )
    template_doc.networks.append(Network(id="Network0"))
    # Add a cell for each acquisition list

```

(continues on next page)

(continued from previous page)

```

popsize = len(acq_list)
template_doc.networks[0].populations.append(
    Population(id="Pop0", component="Izh2007", size=popsize)
)

# Add a current source for each cell, matching the currents that
# were used in the experimental study.
counter = 0
for acq in acq_list:
    template_doc.pulse_generators.append(
        PulseGenerator(
            id="Stim{}".format(counter),
            delay="80ms",
            duration="1000ms",
            amplitude="{{$}pA".format(currents[acq]),
        )
    )
    template_doc.networks[0].explicit_inputs.append(
        ExplicitInput(
            target="Pop0[{}]" .format(counter), input="Stim{}".format(counter)
        )
    )
    counter = counter + 1

# Print a summary
print(template_doc.summary())

# Write to a neuroml file and validate it.
reference = "TuneIzhFergusonPyr3"
template_filename = "{}.net.nml".format(reference)
write_neuroml2_file(template_doc, template_filename, validate=True)

# Now for the tuning bits

# format is type:id/variable:id/units
# supported types: cell/channel/izhikevich2007cell
# supported variables:
# - channel: vShift
# - cell: channelDensity, vShift_channelDensity, channelDensityNernst,
# erev_id, erev_ion, specificCapacitance, resistivity
# - izhikevich2007Cell: all available attributes

# we want to tune these parameters within these ranges
# param: (min, max)
parameters = {
    "izhikevich2007Cell:Izh2007/C/pF": (100, 300),
    "izhikevich2007Cell:Izh2007/k/nS_per_mV": (0.01, 2),
    "izhikevich2007Cell:Izh2007/vr/mV": (-70, -50),
    "izhikevich2007Cell:Izh2007/vt/mV": (-60, 0),
    "izhikevich2007Cell:Izh2007/vpeak/mV": (35, 70),
    "izhikevich2007Cell:Izh2007/a/per_ms": (0.001, 0.4),
    "izhikevich2007Cell:Izh2007/b/nS": (-10, 10),
    "izhikevich2007Cell:Izh2007/c/mV": (-65, -10),
    "izhikevich2007Cell:Izh2007/d/pA": (50, 500),
}
# type: Dict[str, Tuple[float, float]]

```

(continues on next page)

(continued from previous page)

```

# Set up our target data and so on
ctr = 0
target_data = {}
weights = {}
for acq in acq_list:
    # data to fit to:
    # format: path/to/variable:metric
    # metric from pyelectro, for example:
    # https://pyelectro.readthedocs.io/en/latest/pyelectro.html?highlight=mean_
    # spike_frequency#pyelectro.analysis.mean_spike_frequency
    mean_spike_frequency = "Pop0[{}]/v:mean_spike_frequency".format(ctr)
    average_last_1percent = "Pop0[{}]/v:average_last_1percent".format(ctr)
    first_spike_time = "Pop0[{}]/v:first_spike_time".format(ctr)

    # each metric can have an associated weight
    weights[mean_spike_frequency] = 1
    weights[average_last_1percent] = 1
    weights[first_spike_time] = 1

    # value of the target data from our data set
    target_data[mean_spike_frequency] = metrics_from_data[acq][
        "{}:mean_spike_frequency".format(acq)]
    ]
    target_data[average_last_1percent] = metrics_from_data[acq][
        "{}:average_last_1percent".format(acq)]
    ]
    target_data[first_spike_time] = metrics_from_data[acq][
        "{}:first_spike_time".format(acq)]
    ]

    # only add these if the experimental data includes them
    # these are only generated for traces with spikes
    if "{}:average_maximum".format(acq) in metrics_from_data[acq]:
        average_maximum = "Pop0[{}]/v:average_maximum".format(ctr)
        weights[average_maximum] = 1
        target_data[average_maximum] = metrics_from_data[acq][
            "{}:average_maximum".format(acq)]
        ]
    if "{}:average_minimum".format(acq) in metrics_from_data[acq]:
        average_minimum = "Pop0[{}]/v:average_minimum".format(ctr)
        weights[average_minimum] = 1
        target_data[average_minimum] = metrics_from_data[acq][
            "{}:average_minimum".format(acq)]
        ]

    ctr = ctr + 1

    # simulator to use
    simulator = "jNeuroML"

    return run_optimisation(
        # Prefix for new files
        prefix="TuneIzh",
        # Name of the NeuroML template file
        neuroml_file=template_filename,
        # Name of the network

```

(continues on next page)

(continued from previous page)

```

target="Network0",
    # Parameters to be fitted
parameters=list(parameters.keys()),
    # Our max and min constraints
min_constraints=[v[0] for v in parameters.values()],
max_constraints=[v[1] for v in parameters.values()],
    # Weights we set for parameters
weights=weights,
    # The experimental metrics to fit to
target_data=target_data,
    # Simulation time
sim_time=sim_time,
    # EC parameters
population_size=100,
max_evaluations=500,
num_selected=30,
num_offspring=50,
mutation_rate=0.9,
num_elites=3,
    # Seed value
seed=12345,
    # Simulator
simulator=simulator,
dt=0.025,
show_plot_already='--nogui' not in sys.argv,
save_to_file="fitted_izhikevich_fitness.png",
save_to_file_scatter="fitted_izhikevich_scatter.png",
save_to_file_hist="fitted_izhikevich_hist.png",
save_to_file_output="fitted_izhikevich_output.png",
num_parallel_evaluations=4,
)

def run_fitted_cell_simulation(
    sweeps_to_tune_against: List, tuning_report: Dict, simulation_id: str
) -> None:
    """Run a simulation with the values obtained from the fitting

    :param tuning_report: tuning report from the optimser
    :type tuning_report: Dict
    :param simulation_id: text id of simulation
    :type simulation_id: str

    """
    # get the fittest variables
    fittest_vars = tuning_report["fittest vars"]
    C = str(fittest_vars["izhikevich2007Cell:Izh2007/C/pF"]) + "pF"
    k = str(fittest_vars["izhikevich2007Cell:Izh2007/k/nS_per_mV"]) + "nS_per_mV"
    vr = str(fittest_vars["izhikevich2007Cell:Izh2007/vr/mV"]) + "mV"
    vt = str(fittest_vars["izhikevich2007Cell:Izh2007/vt/mV"]) + "mV"
    vpeak = str(fittest_vars["izhikevich2007Cell:Izh2007/vpeak/mV"]) + "mV"
    a = str(fittest_vars["izhikevich2007Cell:Izh2007/a/per_ms"]) + "per_ms"
    b = str(fittest_vars["izhikevich2007Cell:Izh2007/b/nS"]) + "nS"
    c = str(fittest_vars["izhikevich2007Cell:Izh2007/c/mV"]) + "mV"
    d = str(fittest_vars["izhikevich2007Cell:Izh2007/d/pA"]) + "pA"

```

(continues on next page)

(continued from previous page)

```

# Create a simulation using our obtained parameters.
# Note that the tuner generates a graph with the fitted values already, but
# we want to keep a copy of our fitted cell also, so we'll create a NeuroML
# Document ourselves also.
sim_time = 1500.0
simulation_doc = NeuroMLDocument(id="FittedNet")
# Add an Izhikevich cell with some parameters to the document
simulation_doc.izhikevich2007_cells.append(
    Izhikevich2007Cell(
        id="Izh2007",
        C=C,
        v0="-60mV",
        k=k,
        vr=vr,
        vt=vt,
        vpeak=vpeak,
        a=a,
        b=b,
        c=c,
        d=d,
    )
)
simulation_doc.networks.append(Network(id="Network0"))
# Add a cell for each acquisition list
popsize = len(sweeps_to_tune_against)
simulation_doc.networks[0].populations.append(
    Population(id="Pop0", component="Izh2007", size=popsize)
)

# Add a current source for each cell, matching the currents that
# were used in the experimental study.
counter = 0
for acq in sweeps_to_tune_against:
    simulation_doc.pulse_generators.append(
        PulseGenerator(
            id="Stim{}".format(counter),
            delay="80ms",
            duration="1000ms",
            amplitude="{}pA".format(currents[acq]),
        )
    )
    simulation_doc.networks[0].explicit_inputs.append(
        ExplicitInput(
            target="Pop0[{}]".format(counter), input="Stim{}".format(counter)
        )
    )
    counter = counter + 1

# Print a summary
print(simulation_doc.summary())

# Write to a neuroml file and validate it.
reference = "FittedIzhFergusonPyr3"
simulation_filename = "{}.net.nml".format(reference)
write_neuroml2_file(simulation_doc, simulation_filename, validate=True)

```

(continues on next page)

(continued from previous page)

```

simulation = LEMSSimulation(
    sim_id=simulation_id,
    duration=sim_time,
    dt=0.1,
    target="Network0",
    simulation_seed=54321,
)
simulation.include_neuroml2_file(simulation_filename)
simulation.create_output_file("output0", "{}.v.dat".format(simulation_id))
counter = 0
for acq in sweeps_to_tune_against:
    simulation.add_column_to_output_file(
        "output0", "Pop0[{}].format(counter), "Pop0[{}]/v".format(counter)
    )
    counter = counter + 1
simulation_file = simulation.save_to_file()
# simulate
run_lems_with_jneuroml(simulation_file, max_memory="2G", nogui=True, plot=False)

def plot_sim_data(
    sweeps_to_tune_against: List, simulation_id: str, memb_pots: Dict
) -> None:
    """Plot data from our fitted simulation

    :param simulation_id: string id of simulation
    :type simulation_id: str
    """
    # Plot
    data_array = np.loadtxt("%s.v.dat" % simulation_id)

    # construct data for plotting
    counter = 0
    time_vals_list = []
    sim_v_list = []
    data_v_list = []
    data_t_list = []
    stim_vals = []
    for acq in sweeps_to_tune_against:
        stim_vals.append("{}pA".format(currents[acq]))

        # remains the same for all columns
        time_vals_list.append(data_array[:, 0] * 1000.0)
        sim_v_list.append(data_array[:, counter + 1] * 1000.0)

        data_v_list.append(memb_pots[acq][1])
        data_t_list.append(memb_pots[acq][0])

        counter = counter + 1

    # Model membrane potential plot
    generate_plot(
        xvalues=time_vals_list,
        yvalues=sim_v_list,
        labels=stim_vals,
        title="Membrane potential (model)",
    )

```

(continues on next page)

(continued from previous page)

```

show_plot_already=False,
save_figure_to="%s-model-v.png" % simulation_id,
xaxis="time (ms)",
yaxis="membrane potential (mV)",
)
# data membrane potential plot
generate_plot(
    xvalues=data_t_list,
    yvalues=data_v_list,
    labels=stim_vals,
    title="Membrane potential (exp)",
    show_plot_already=False,
    save_figure_to="%s-exp-v.png" % simulation_id,
    xaxis="time (ms)",
    yaxis="membrane potential (mV)",
)

if __name__ == "__main__":
    # set the default size for generated plots
    # https://matplotlib.org/stable/tutorials/introductory/customizing.html#a-sample-
    #matplotlibrc-file
    import matplotlib as mpl
    mpl.rcParams["figure.figsize"] = [18, 12]

    io = pynwb.NWBHDF5IO("./FergusonEtAl2015_PYR3.nwb", "r")
    datafile = io.read()

    analysis_results, currents, memb_pots = get_data_metrics(datafile)

    # Choose what sweeps to tune against.
    # There are 33 sweeps: 1..33.
    # sweeps_to_tune_against = [1, 2, 15, 30, 31, 32, 33]
    sweeps_to_tune_against = [16, 21]
    report = tune_izh_model(sweeps_to_tune_against, analysis_results, currents)

    simulation_id = "fitted_izhikevich_sim"
    run_fitted_cell_simulation(sweeps_to_tune_against, report, simulation_id)

    plot_sim_data(sweeps_to_tune_against, simulation_id, memb_pots)

    # close the data file
    io.close()

```

9.1 Loading data and calculating metrics to use for optimisation

The first step in the optimisation of the model is to obtain the data that the model is to be fitted against. In this example, we will use the data set of CA1 pyramidal cell recordings using an intact whole hippocampus preparation, including recordings of rebound firing [FHA+15]. The data set is provided in the Neurodata Without Borders (NWB) format. It can be downloaded [here](#) on the Open Source Brain repository, and can also be viewed on the [NWB Explorer](#) web application:

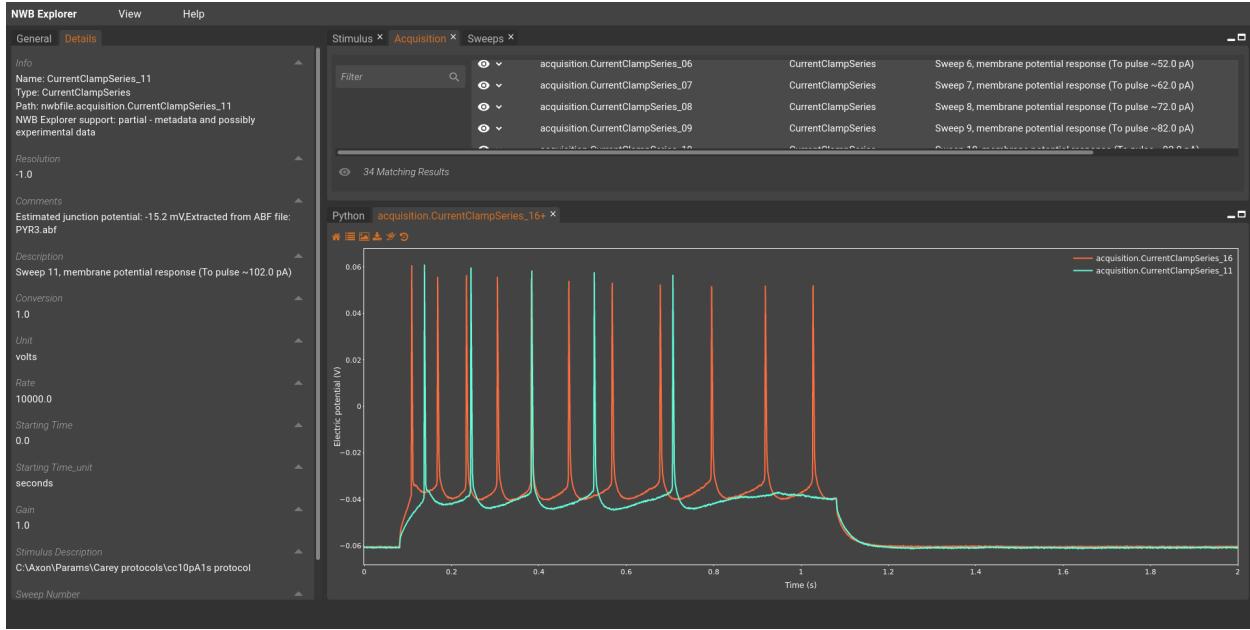


Fig. 9.3: Screenshot showing two recordings from FergusonEtAl2015_PYR3.nwb in NWB Explorer.

For this example, we will use the `FergusonEtAl2015_PYR3.nwb` data file. We use the `PyNWB` package to read it, and then pass the loaded data to our `get_data_metrics` function to extract the metrics we want to use for model fitting.

```
io = pynwb.NWBHDF5IO("./FergusonEtAl2015_PYR3.nwb", "r")
datafile = io.read()

analysis_results, currents, memb_pots = get_data_metrics(datafile)
```

Similar to `libNeuroML`, PyNWB provides a Python object model to interact with NWB files. You can learn more on using PyNWB in its [documentation](#).

Here, the data file includes recordings from multiple (33 in total) current clamp experiments that are numbered from 1 through 33. We iterate over each recording individually to extract the membrane potential values and store them in `data_v`. For each, we also calculate the time stamps for the recordings from the provided sampling rate. We pass this information to the `simple_network_analysis` function provided by the `PyElectro` Python package to calculate features (metrics) that we will use for fitting a neuron model.

```
def get_data_metrics(datafile: Container) -> Tuple[Dict, Dict, Dict]:
    """Analyse the data to get metrics to tune against.

    :returns: metrics from pyelectro analysis, currents, and the membrane potential
    values
```

(continues on next page)

(continued from previous page)

```

"""
analysis_results = {}
currents = {}
memb_vals = {}
total_acquisitions = len(datafile.acquisition)

for acq in range(1, total_acquisitions):
    print("Going over acquisition # {}".format(acq))

    # stimulus lasts about 1000ms, so we take about the first 1500 ms
    data_v = (
        datafile.acquisition["CurrentClampSeries_{}".format(acq)]._
        data[:15000] * 1000.0
    )
    # get sampling rate from the data
    sampling_rate = datafile.acquisition[_
        "CurrentClampSeries_{}".format(acq)].rate
    # generate time steps from sampling rate
    data_t = np.arange(0, len(data_v) / sampling_rate, 1.0 / sampling_rate) *_
    1000.0
    # run the analysis
    analysis_results[acq] = simple_network_analysis({acq: data_v}, data_t)

    # extract current from description, but can be extracted from other
    # locations also, such as the CurrentClampStimulus series.
    data_i = (
        datafile.acquisition["CurrentClampSeries_{}".format(acq)]._
        description.split("(")[1]
        .split("~")[1]
        .split(" ")[0]
    )
    currents[acq] = data_i
    memb_vals[acq] = (data_t, data_v)

return (analysis_results, currents, memb_vals)

```

The features calculated by PyElectro for each recording, which we store in `analysis_results`, can be seen below:

```

Going over acquisition # 1
pyelectro >>> { '1:average_last_1percent': -60.4182980855306,
pyelectro >>>     '1:max_peak_no': 0,
pyelectro >>>     '1:maximum': -57.922367,
pyelectro >>>     '1:mean_spike_frequency': 0,
pyelectro >>>     '1:min_peak_no': 0,
pyelectro >>>     '1:minimum': -60.729984}
Going over acquisition # 2
pyelectro >>> { '2:average_last_1percent': -60.2773068745931,
pyelectro >>>     '2:max_peak_no': 0,
pyelectro >>>     '2:maximum': -56.182865,
pyelectro >>>     '2:mean_spike_frequency': 0,
pyelectro >>>     '2:min_peak_no': 0,
pyelectro >>>     '2:minimum': -60.882572}
Going over acquisition # 3
pyelectro >>> { '3:average_last_1percent': -60.175174713134766,
pyelectro >>>     '3:max_peak_no': 0,

```

(continues on next page)

(continued from previous page)

```

pyelectro >>>      '3:maximum': -54.22974,
pyelectro >>>      '3:mean_spike_frequency': 0,
pyelectro >>>      '3:min_peak_no': 0,
pyelectro >>>      '3:minimum': -60.7605}
Going over acquisition # 4
pyelectro >>> {   '4:average_last_1percent': -60.11576716105143,
pyelectro >>>      '4:max_peak_no': 0,
pyelectro >>>      '4:maximum': -49.133305,
pyelectro >>>      '4:mean_spike_frequency': 0,
pyelectro >>>      '4:min_peak_no': 0,
pyelectro >>>      '4:minimum': -60.607914}
Going over acquisition # 5
pyelectro >>> {   '5:average_last_1percent': -59.628299713134766,
pyelectro >>>      '5:max_peak_no': 0,
pyelectro >>>      '5:maximum': -48.645023,
pyelectro >>>      '5:mean_spike_frequency': 0,
pyelectro >>>      '5:min_peak_no': 0,
pyelectro >>>      '5:minimum': -60.241703}
Going over acquisition # 6
pyelectro >>> {   '6:average_last_1percent': -60.04679743448893,
pyelectro >>>      '6:max_peak_no': 0,
pyelectro >>>      '6:maximum': -45.16602,
pyelectro >>>      '6:mean_spike_frequency': 0,
pyelectro >>>      '6:min_peak_no': 0,
pyelectro >>>      '6:minimum': -60.699467}
Going over acquisition # 7
pyelectro >>> {   '7:average_last_1percent': -59.88566462198893,
pyelectro >>>      '7:average_maximum': 66.3147,
pyelectro >>>      '7:average_minimum': -47.9126,
pyelectro >>>      '7:first_spike_time': 482.2000000000005,
pyelectro >>>      '7:max_peak_no': 2,
pyelectro >>>      '7:maximum': 67.38281,
pyelectro >>>      '7:mean_spike_frequency': 0,
pyelectro >>>      '7:min_peak_no': 1,
pyelectro >>>      '7:minimum': -60.15015}
Going over acquisition # 8
pyelectro >>> {   '8:average_last_1percent': -60.5531857808431,
pyelectro >>>      '8:average_maximum': 64.63623,
pyelectro >>>      '8:average_minimum': -46.05103,
pyelectro >>>      '8:first_spike_time': 280.8,
pyelectro >>>      '8:max_peak_no': 2,
pyelectro >>>      '8:maximum': 65.460205,
pyelectro >>>      '8:mean_spike_frequency': 0,
pyelectro >>>      '8:min_peak_no': 1,
pyelectro >>>      '8:minimum': -61.096195}
Going over acquisition # 9
pyelectro >>> {   '9:average_last_1percent': -60.2797482808431,
pyelectro >>>      '9:average_maximum': 62.187195,
pyelectro >>>      '9:average_minimum': -45.867924,
pyelectro >>>      '9:first_spike_time': 192.1000000000002,
pyelectro >>>      '9:interspike_time_covar': 0.12604539832023948,
pyelectro >>>      '9:max_interspike_time': 233.69999999999993,
pyelectro >>>      '9:max_peak_no': 4,
pyelectro >>>      '9:maximum': 64.42261,
pyelectro >>>      '9:mean_spike_frequency': 4.984216647283602,
pyelectro >>>      '9:min_interspike_time': 172.29999999999995,

```

(continues on next page)

(continued from previous page)

```

pyelectro >>>      '9:min_peak_no': 3,
pyelectro >>>      '9:minimum': -60.91309,
pyelectro >>>      '9:peak_decay_exponent': -0.008125577959852965,
pyelectro >>>      '9:peak_linear_gradient': -0.00764805557429393,
pyelectro >>>      '9:spike_broadening': 0.891046031985908,
pyelectro >>>      '9:spike_frequency_adaptation': -0.05850411039850073,
pyelectro >>>      '9:spike_width_adaptation': 0.02767948174212246,
pyelectro >>>      '9:trough_decay_exponent': -0.00318728965696189,
pyelectro >>>      '9:trough_phase_adaptation': -0.05016262394149966}

Going over acquisition # 10
pyelectro >>> {   '10:average_last_1percent': -60.4292844136556,
pyelectro >>>      '10:average_maximum': 59.680183,
pyelectro >>>      '10:average_minimum': -44.731144,
pyelectro >>>      '10:first_spike_time': 156.9,
pyelectro >>>      '10:interspike_time_covar': 0.5779639183148945,
pyelectro >>>      '10:max_interspike_time': 438.5000000000001,
pyelectro >>>      '10:max_peak_no': 5,
pyelectro >>>      '10:maximum': 62.072758,
pyelectro >>>      '10:mean_spike_frequency': 4.553215708594194,
pyelectro >>>      '10:min_interspike_time': 132.6000000000005,
pyelectro >>>      '10:min_peak_no': 4,
pyelectro >>>      '10:minimum': -60.882572,
pyelectro >>>      '10:peak_decay_exponent': -0.009585017031582235,
pyelectro >>>      '10:peak_linear_gradient': -0.005260966229876463,
pyelectro >>>      '10:spike_broadening': 0.8756680523402136,
pyelectro >>>      '10:spike_frequency_adaptation': -0.04780471479504103,
pyelectro >>>      '10:spike_width_adaptation': 0.014551159295128686,
pyelectro >>>      '10:trough_decay_exponent': -0.006056437839814275,
pyelectro >>>      '10:trough_phase_adaptation': -0.04269124477477909}

Going over acquisition # 11
pyelectro >>> {   '11:average_last_1percent': -60.84635798136393,
pyelectro >>>      '11:average_maximum': 58.73414,
pyelectro >>>      '11:average_minimum': -43.800358,
pyelectro >>>      '11:first_spike_time': 138.1,
pyelectro >>>      '11:interspike_time_covar': 0.18317620745649893,
pyelectro >>>      '11:max_interspike_time': 179.9999999999999,
pyelectro >>>      '11:max_peak_no': 5,
pyelectro >>>      '11:maximum': 61.03516,
pyelectro >>>      '11:mean_spike_frequency': 7.033585370142431,
pyelectro >>>      '11:min_interspike_time': 106.5000000000003,
pyelectro >>>      '11:min_peak_no': 4,
pyelectro >>>      '11:minimum': -61.248783,
pyelectro >>>      '11:peak_decay_exponent': -0.010372120562797708,
pyelectro >>>      '11:peak_linear_gradient': -0.0074866848970347325,
pyelectro >>>      '11:spike_broadening': 0.8498887121142943,
pyelectro >>>      '11:spike_frequency_adaptation': -0.052381521145715274,
pyelectro >>>      '11:spike_width_adaptation': 0.025414793671653328,
pyelectro >>>      '11:trough_decay_exponent': -0.007552906636393599,
pyelectro >>>      '11:trough_phase_adaptation': -0.04473631982885844}

Going over acquisition # 12
pyelectro >>> {   '12:average_last_1percent': -61.085208892822266,
pyelectro >>>      '12:average_maximum': 58.481857,
pyelectro >>>      '12:average_minimum': -42.974857,
pyelectro >>>      '12:first_spike_time': 127.4000000000002,
pyelectro >>>      '12:interspike_time_covar': 0.16275057704467177,
pyelectro >>>      '12:max_interspike_time': 136.5,

```

(continues on next page)

(continued from previous page)

```

pyelectro >>> '12:max_peak_no': 6,
pyelectro >>> '12:maximum': 61.737064,
pyelectro >>> '12:mean_spike_frequency': 8.791981712678037,
pyelectro >>> '12:min_interspike_time': 84.60000000000001,
pyelectro >>> '12:min_peak_no': 5,
pyelectro >>> '12:minimum': -61.462406,
pyelectro >>> '12:peak_decay_exponent': -0.015987075984851808,
pyelectro >>> '12:peak_linear_gradient': -0.009383652380440125,
pyelectro >>> '12:spike_broadening': 0.8205694396572242,
pyelectro >>> '12:spike_frequency_adaptation': -0.04259621402895674,
pyelectro >>> '12:spike_width_adaptation': 0.0228428573204052,
pyelectro >>> '12:trough_decay_exponent': -0.012180031782655684,
pyelectro >>> '12:trough_phase_adaptation': 0.0003391093549627843}

Going over acquisition # 13
pyelectro >>> { '13:average_last_1percent': -60.6233762105306,
pyelectro >>> '13:average_maximum': 56.980137,
pyelectro >>> '13:average_minimum': -42.205814,
pyelectro >>> '13:first_spike_time': 122.9,
pyelectro >>> '13:interspike_time_covar': 0.1989630987796946,
pyelectro >>> '13:max_interspike_time': 138.9000000000001,
pyelectro >>> '13:max_peak_no': 8,
pyelectro >>> '13:maximum': 61.30982,
pyelectro >>> '13:mean_spike_frequency': 9.743875278396434,
pyelectro >>> '13:min_interspike_time': 75.4,
pyelectro >>> '13:min_peak_no': 7,
pyelectro >>> '13:minimum': -60.974125,
pyelectro >>> '13:peak_decay_exponent': -0.01856642711769415,
pyelectro >>> '13:peak_linear_gradient': -0.009561076077386068,
pyelectro >>> '13:spike_broadening': 0.803052014150605,
pyelectro >>> '13:spike_frequency_adaptation': -0.028079821139188187,
pyelectro >>> '13:spike_width_adaptation': 0.01504310702538977,
pyelectro >>> '13:trough_decay_exponent': -0.013208504624154408,
pyelectro >>> '13:trough_phase_adaptation': -0.025379665674913895}

Going over acquisition # 14
pyelectro >>> { '14:average_last_1percent': -60.723270416259766,
pyelectro >>> '14:average_maximum': 55.986195,
pyelectro >>> '14:average_minimum': -41.54587,
pyelectro >>> '14:first_spike_time': 114.8,
pyelectro >>> '14:interspike_time_covar': 0.34335772265161896,
pyelectro >>> '14:max_interspike_time': 194.39999999999998,
pyelectro >>> '14:max_peak_no': 9,
pyelectro >>> '14:maximum': 60.882572,
pyelectro >>> '14:mean_spike_frequency': 8.785416209092906,
pyelectro >>> '14:min_interspike_time': 74.4000000000002,
pyelectro >>> '14:min_peak_no': 8,
pyelectro >>> '14:minimum': -61.126713,
pyelectro >>> '14:peak_decay_exponent': -0.022541304298167242,
pyelectro >>> '14:peak_linear_gradient': -0.008000988888256885,
pyelectro >>> '14:spike_broadening': 0.8009562042583951,
pyelectro >>> '14:spike_frequency_adaptation': -0.022434594832088855,
pyelectro >>> '14:spike_width_adaptation': 0.010424354074822617,
pyelectro >>> '14:trough_decay_exponent': -0.018754566142487872,
pyelectro >>> '14:trough_phase_adaptation': -0.019014319738344054}

Going over acquisition # 15
pyelectro >>> { '15:average_last_1percent': -60.99833552042643,
pyelectro >>> '15:average_maximum': 55.89295,

```

(continues on next page)

(continued from previous page)

```

pyelectro >>>      '15:average_minimum': -40.78892,
pyelectro >>>      '15:first_spike_time': 113.2,
pyelectro >>>      '15:interspike_time_covar': 0.6436697297327385,
pyelectro >>>      '15:max_interspike_time': 311.30000000000007,
pyelectro >>>      '15:max_peak_no': 8,
pyelectro >>>      '15:maximum': 60.91309,
pyelectro >>>      '15:mean_spike_frequency': 8.201523140011716,
pyelectro >>>      '15:min_interspike_time': 71.60000000000001,
pyelectro >>>      '15:min_peak_no': 7,
pyelectro >>>      '15:minimum': -61.370853,
pyelectro >>>      '15:peak_decay_exponent': -0.025953113905923406,
pyelectro >>>      '15:peak_linear_gradient': -0.008306657481016694,
pyelectro >>>      '15:spike_broadening': 0.7782197474305453,
pyelectro >>>      '15:spike_frequency_adaptation': -0.030010388928284993,
pyelectro >>>      '15:spike_width_adaptation': 0.012108100430332605,
pyelectro >>>      '15:trough_decay_exponent': -0.01262141611091244,
pyelectro >>>      '15:trough_phase_adaptation': -0.025746376793896804}

Going over acquisition # 16
pyelectro >>> {      '16:average_last_1percent': -60.380863189697266,
pyelectro >>>      '16:average_maximum': 54.52382,
pyelectro >>>      '16:average_minimum': -39.78882,
pyelectro >>>      '16:first_spike_time': 108.9,
pyelectro >>>      '16:interspike_time_covar': 0.23652534644271225,
pyelectro >>>      '16:max_interspike_time': 123.00000000000011,
pyelectro >>>      '16:max_peak_no': 11,
pyelectro >>>      '16:maximum': 60.7605,
pyelectro >>>      '16:mean_spike_frequency': 10.8837614279495,
pyelectro >>>      '16:min_interspike_time': 59.30000000000001,
pyelectro >>>      '16:min_peak_no': 10,
pyelectro >>>      '16:minimum': -60.79102,
pyelectro >>>      '16:peak_decay_exponent': -0.03301104578192642,
pyelectro >>>      '16:peak_linear_gradient': -0.007545664792227554,
pyelectro >>>      '16:spike_broadening': 0.7546314677569799,
pyelectro >>>      '16:spike_frequency_adaptation': -0.013692136410690963,
pyelectro >>>      '16:spike_width_adaptation': 0.008664177864358623,
pyelectro >>>      '16:trough_decay_exponent': -0.023836469122601508,
pyelectro >>>      '16:trough_phase_adaptation': -0.010081239597430595}

Going over acquisition # 17
pyelectro >>> {      '17:average_last_1percent': -60.552982330322266,
pyelectro >>>      '17:average_maximum': 54.44642,
pyelectro >>>      '17:average_minimum': -39.008247,
pyelectro >>>      '17:first_spike_time': 105.6,
pyelectro >>>      '17:interspike_time_covar': 0.19651182311074483,
pyelectro >>>      '17:max_interspike_time': 106.29999999999995,
pyelectro >>>      '17:max_peak_no': 10,
pyelectro >>>      '17:maximum': 60.63843,
pyelectro >>>      '17:mean_spike_frequency': 11.506008693428791,
pyelectro >>>      '17:min_interspike_time': 58.60000000000002,
pyelectro >>>      '17:min_peak_no': 9,
pyelectro >>>      '17:minimum': -61.03516,
pyelectro >>>      '17:peak_decay_exponent': -0.03684157763477531,
pyelectro >>>      '17:peak_linear_gradient': -0.009090863209461205,
pyelectro >>>      '17:spike_broadening': 0.7534694949245309,
pyelectro >>>      '17:spike_frequency_adaptation': -0.023373852901912264,
pyelectro >>>      '17:spike_width_adaptation': 0.011268432654001511,
pyelectro >>>      '17:trough_decay_exponent': -0.020590018720385343,

```

(continues on next page)

(continued from previous page)

```

pyelectro >>>      '17:trough_phase_adaptation': -0.00906121257722172}
Going over acquisition # 18
pyelectro >>> {   '18:average_last_1percent': -60.64799372355143,
pyelectro >>>     '18:average_maximum': 53.783077,
pyelectro >>>     '18:average_minimum': -38.424683,
pyelectro >>>     '18:first_spike_time': 104.2,
pyelectro >>>     '18:interspike_time_covar': 0.2502832189694502,
pyelectro >>>     '18:max_interspike_time': 124.59999999999991,
pyelectro >>>     '18:max_peak_no': 11,
pyelectro >>>     '18:maximum': 60.63843,
pyelectro >>>     '18:mean_spike_frequency': 11.420740063956146,
pyelectro >>>     '18:min_interspike_time': 53.0999999999998,
pyelectro >>>     '18:min_peak_no': 10,
pyelectro >>>     '18:minimum': -61.03516,
pyelectro >>>     '18:peak_decay_exponent': -0.04141134556729957,
pyelectro >>>     '18:peak_linear_gradient': -0.008476351143979814,
pyelectro >>>     '18:spike_broadening': 0.7403041191114148,
pyelectro >>>     '18:spike_frequency_adaptation': -0.01809717600857398,
pyelectro >>>     '18:spike_width_adaptation': 0.009174879706803092,
pyelectro >>>     '18:trough_decay_exponent': -0.02760451378209885,
pyelectro >>>     '18:trough_phase_adaptation': -0.005774543184281237}

Going over acquisition # 19
pyelectro >>> {   '19:average_last_1percent': -60.855106353759766,
pyelectro >>>     '19:average_maximum': 53.430737,
pyelectro >>>     '19:average_minimum': -37.713623,
pyelectro >>>     '19:first_spike_time': 102.50000000000001,
pyelectro >>>     '19:interspike_time_covar': 0.25333327224227414,
pyelectro >>>     '19:max_interspike_time': 114.69999999999993,
pyelectro >>>     '19:max_peak_no': 11,
pyelectro >>>     '19:maximum': 60.607914,
pyelectro >>>     '19:mean_spike_frequency': 12.47038284075321,
pyelectro >>>     '19:min_interspike_time': 51.8,
pyelectro >>>     '19:min_peak_no': 10,
pyelectro >>>     '19:minimum': -61.248783,
pyelectro >>>     '19:peak_decay_exponent': -0.04918301935790627,
pyelectro >>>     '19:peak_linear_gradient': -0.008553290998046907,
pyelectro >>>     '19:spike_broadening': 0.7301692622822238,
pyelectro >>>     '19:spike_frequency_adaptation': -0.015561797159916213,
pyelectro >>>     '19:spike_width_adaptation': 0.010054185105794627,
pyelectro >>>     '19:trough_decay_exponent': -0.03413795061471875,
pyelectro >>>     '19:trough_phase_adaptation': -0.011967671377256838}

Going over acquisition # 20
pyelectro >>> {   '20:average_last_1percent': -60.793460845947266,
pyelectro >>>     '20:average_maximum': 53.11169,
pyelectro >>>     '20:average_minimum': -37.045288,
pyelectro >>>     '20:first_spike_time': 101.4,
pyelectro >>>     '20:interspike_time_covar': 0.2865607615520669,
pyelectro >>>     '20:max_interspike_time': 123.0,
pyelectro >>>     '20:max_peak_no': 11,
pyelectro >>>     '20:maximum': 60.51636,
pyelectro >>>     '20:mean_spike_frequency': 12.624668602449184,
pyelectro >>>     '20:min_interspike_time': 49.09999999999994,
pyelectro >>>     '20:min_peak_no': 10,
pyelectro >>>     '20:minimum': -61.30982,
pyelectro >>>     '20:peak_decay_exponent': -0.053836942989781186,
pyelectro >>>     '20:peak_linear_gradient': -0.009554089132365705,

```

(continues on next page)

(continued from previous page)

```

pyelectro >>>      '20:spike_broadening': 0.7067401817806985,
pyelectro >>>      '20:spike_frequency_adaptation': -0.02046601020346991,
pyelectro >>>      '20:spike_width_adaptation': 0.01032699280307952,
pyelectro >>>      '20:trough_decay_exponent': -0.03229413870032492,
pyelectro >>>      '20:trough_phase_adaptation': -0.009902402143729637}
Going over acquisition # 21
pyelectro >>> {   '21:average_last_1percent': -59.912113189697266,
pyelectro >>>     '21:average_maximum': 51.912754,
pyelectro >>>     '21:average_minimum': -35.964966,
pyelectro >>>     '21:first_spike_time': 100.4,
pyelectro >>>     '21:interspike_time_covar': 0.31784939578511834,
pyelectro >>>     '21:max_interspike_time': 130.10000000000002,
pyelectro >>>     '21:max_peak_no': 13,
pyelectro >>>     '21:maximum': 61.15723,
pyelectro >>>     '21:mean_spike_frequency': 12.369858777445623,
pyelectro >>>     '21:min_interspike_time': 45.10000000000002,
pyelectro >>>     '21:min_peak_no': 12,
pyelectro >>>     '21:minimum': -61.614994,
pyelectro >>>     '21:peak_decay_exponent': -0.06340663337165775,
pyelectro >>>     '21:peak_linear_gradient': -0.009514554022982258,
pyelectro >>>     '21:spike_broadening': 0.6805457955255507,
pyelectro >>>     '21:spike_frequency_adaptation': -0.012321447021551269,
pyelectro >>>     '21:spike_width_adaptation': 0.007303096579113352,
pyelectro >>>     '21:trough_decay_exponent': -0.03236374133246423,
pyelectro >>>     '21:trough_phase_adaptation': -0.009705621425080494}
Going over acquisition # 22
pyelectro >>> {   '22:average_last_1percent': -60.0850461324056,
pyelectro >>>     '22:average_maximum': 51.325874,
pyelectro >>>     '22:average_minimum': -35.22746,
pyelectro >>>     '22:first_spike_time': 97.7,
pyelectro >>>     '22:interspike_time_covar': 0.3280130255436152,
pyelectro >>>     '22:max_interspike_time': 123.30000000000007,
pyelectro >>>     '22:max_peak_no': 13,
pyelectro >>>     '22:maximum': 60.91309,
pyelectro >>>     '22:mean_spike_frequency': 12.784998934583422,
pyelectro >>>     '22:min_interspike_time': 40.60000000000001,
pyelectro >>>     '22:min_peak_no': 12,
pyelectro >>>     '22:minimum': -60.51636,
pyelectro >>>     '22:peak_decay_exponent': -0.07398649685748288,
pyelectro >>>     '22:peak_linear_gradient': -0.008846573199048182,
pyelectro >>>     '22:spike_broadening': 0.673572178798493,
pyelectro >>>     '22:spike_frequency_adaptation': -0.01394751742502263,
pyelectro >>>     '22:spike_width_adaptation': 0.00745978471908774,
pyelectro >>>     '22:trough_decay_exponent': -0.03985576781966312,
pyelectro >>>     '22:trough_phase_adaptation': -0.012949190338366346}
Going over acquisition # 23
pyelectro >>> {   '23:average_last_1percent': -60.42582575480143,
pyelectro >>>     '23:average_maximum': 50.883705,
pyelectro >>>     '23:average_minimum': -34.70553,
pyelectro >>>     '23:first_spike_time': 97.0,
pyelectro >>>     '23:interspike_time_covar': 0.3025008293643565,
pyelectro >>>     '23:max_interspike_time': 113.00000000000011,
pyelectro >>>     '23:max_peak_no': 14,
pyelectro >>>     '23:maximum': 61.126713,
pyelectro >>>     '23:mean_spike_frequency': 13.453378867846421,
pyelectro >>>     '23:min_interspike_time': 37.59999999999994,

```

(continues on next page)

(continued from previous page)

```

pyelectro >>> '23:min_peak_no': 13,
pyelectro >>> '23:minimum': -60.79102,
pyelectro >>> '23:peak_decay_exponent': -0.09454776279913378,
pyelectro >>> '23:peak_linear_gradient': -0.007913299554316041,
pyelectro >>> '23:spike_broadening': 0.6664213397577756,
pyelectro >>> '23:spike_frequency_adaptation': -0.014974747741974222,
pyelectro >>> '23:spike_width_adaptation': 0.0067844847504214744,
pyelectro >>> '23:trough_decay_exponent': -0.04357925225307838,
pyelectro >>> '23:trough_phase_adaptation': -0.006437508651280852}
Going over acquisition # 24
pyelectro >>> {
    '24:average_last_1percent': -60.48380915323893,
    '24:average_maximum': 50.66572,
    '24:average_minimum': -34.043533,
    '24:first_spike_time': 95.9,
    '24:interspike_time_covar': 0.2757061784222999,
    '24:max_interspike_time': 106.29999999999995,
    '24:max_peak_no': 14,
    '24:maximum': 61.2793,
    '24:mean_spike_frequency': 13.685651121170649,
    '24:min_interspike_time': 42.5,
    '24:min_peak_no': 13,
    '24:minimum': -61.03516,
    '24:peak_decay_exponent': -0.10291055243646331,
    '24:peak_linear_gradient': -0.008156801002617309,
    '24:spike_broadening': 0.647962840377368,
    '24:spike_frequency_adaptation': -0.01033640102347251,
    '24:spike_width_adaptation': 0.0070544336550024695,
    '24:trough_decay_exponent': -0.04136814132208841,
    '24:trough_phase_adaptation': -0.007165702258804302}
Going over acquisition # 25
pyelectro >>> {
    '25:average_last_1percent': -60.056766510009766,
    '25:average_maximum': 50.34093,
    '25:average_minimum': -33.228947,
    '25:first_spike_time': 95.60000000000001,
    '25:interspike_time_covar': 0.28833246313094774,
    '25:max_interspike_time': 100.80000000000007,
    '25:max_peak_no': 14,
    '25:maximum': 61.40137,
    '25:mean_spike_frequency': 14.023732470334416,
    '25:min_interspike_time': 39.10000000000001,
    '25:min_peak_no': 13,
    '25:minimum': -60.607914,
    '25:peak_decay_exponent': -0.1047695739806843,
    '25:peak_linear_gradient': -0.00879119329941481,
    '25:spike_broadening': 0.6394636967007732,
    '25:spike_frequency_adaptation': -0.011357738090467376,
    '25:spike_width_adaptation': 0.007198805900434394,
    '25:trough_decay_exponent': -0.03898706127522965,
    '25:trough_phase_adaptation': -0.005791081007349543}
Going over acquisition # 26
pyelectro >>> {
    '26:average_last_1percent': -60.2272580464681,
    '26:average_maximum': 49.776356,
    '26:average_minimum': -32.534084,
    '26:first_spike_time': 94.89999999999999,
    '26:interspike_time_covar': 0.3174070553930572,
    '26:max_interspike_time': 115.70000000000005,

```

(continues on next page)

(continued from previous page)

```

pyelectro >>> '26:max_peak_no': 14,
pyelectro >>> '26:maximum': 61.15723,
pyelectro >>> '26:mean_spike_frequency': 14.697569248162802,
pyelectro >>> '26:min_interspike_time': 35.600000000000001,
pyelectro >>> '26:min_peak_no': 13,
pyelectro >>> '26:minimum': -60.729984,
pyelectro >>> '26:peak_decay_exponent': -0.11931629839276492,
pyelectro >>> '26:peak_linear_gradient': -0.009603020729143796,
pyelectro >>> '26:spike_broadening': 0.6253471365146865,
pyelectro >>> '26:spike_frequency_adaptation': -0.015585439927950483,
pyelectro >>> '26:spike_width_adaptation': 0.007759081127414656,
pyelectro >>> '26:trough_decay_exponent': -0.049101688628808246,
pyelectro >>> '26:trough_phase_adaptation': -0.012139424609633551}

Going over acquisition # 27
pyelectro >>> { '27:average_last_1percent': -60.3578732808431,
pyelectro >>> '27:average_maximum': 49.30769,
pyelectro >>> '27:average_minimum': -32.003548,
pyelectro >>> '27:first_spike_time': 93.60000000000001,
pyelectro >>> '27:interspike_time_covar': 0.309017296765978,
pyelectro >>> '27:max_interspike_time': 109.0,
pyelectro >>> '27:max_peak_no': 14,
pyelectro >>> '27:maximum': 61.43189,
pyelectro >>> '27:mean_spike_frequency': 14.729209154769997,
pyelectro >>> '27:min_interspike_time': 32.19999999999999,
pyelectro >>> '27:min_peak_no': 13,
pyelectro >>> '27:minimum': -60.7605,
pyelectro >>> '27:peak_decay_exponent': -0.12538878925370048,
pyelectro >>> '27:peak_linear_gradient': -0.009067695791685005,
pyelectro >>> '27:spike_broadening': 0.6066765033439258,
pyelectro >>> '27:spike_frequency_adaptation': -0.015492078035720953,
pyelectro >>> '27:spike_width_adaptation': 0.007539073044770246,
pyelectro >>> '27:trough_decay_exponent': -0.05115040330367209,
pyelectro >>> '27:trough_phase_adaptation': -0.012545852013557039}

Going over acquisition # 28
pyelectro >>> { '28:average_last_1percent': -60.3798459370931,
pyelectro >>> '28:average_maximum': 48.999027,
pyelectro >>> '28:average_minimum': -31.055996,
pyelectro >>> '28:first_spike_time': 93.4,
pyelectro >>> '28:interspike_time_covar': 0.30636145843159784,
pyelectro >>> '28:max_interspike_time': 104.70000000000005,
pyelectro >>> '28:max_peak_no': 15,
pyelectro >>> '28:maximum': 61.2793,
pyelectro >>> '28:mean_spike_frequency': 14.760147601476012,
pyelectro >>> '28:min_interspike_time': 34.20000000000002,
pyelectro >>> '28:min_peak_no': 14,
pyelectro >>> '28:minimum': -60.943607,
pyelectro >>> '28:peak_decay_exponent': -0.16496940386452533,
pyelectro >>> '28:peak_linear_gradient': -0.007631694348641044,
pyelectro >>> '28:spike_broadening': 0.5953810644123492,
pyelectro >>> '28:spike_frequency_adaptation': -0.014342884276047468,
pyelectro >>> '28:spike_width_adaptation': 0.006650416835633624,
pyelectro >>> '28:trough_decay_exponent': -0.04668774074305247,
pyelectro >>> '28:trough_phase_adaptation': 0.006738292518776989}

Going over acquisition # 29
pyelectro >>> { '29:average_last_1percent': -60.662235260009766,
pyelectro >>> '29:average_maximum': 48.664642,

```

(continues on next page)

(continued from previous page)

```

pyelectro >>> '29:average_minimum': -30.39551,
pyelectro >>> '29:first_spike_time': 93.10000000000001,
pyelectro >>> '29:interspike_time_covar': 0.5343365146969321,
pyelectro >>> '29:max_interspike_time': 183.60000000000002,
pyelectro >>> '29:max_peak_no': 14,
pyelectro >>> '29:maximum': 61.370853,
pyelectro >>> '29:mean_spike_frequency': 14.458903347792235,
pyelectro >>> '29:min_interspike_time': 25.59999999999994,
pyelectro >>> '29:min_peak_no': 13,
pyelectro >>> '29:minimum': -61.187748,
pyelectro >>> '29:peak_decay_exponent': -0.15153571047754788,
pyelectro >>> '29:peak_linear_gradient': -0.007862338819211844,
pyelectro >>> '29:spike_broadening': 0.5870269764023183,
pyelectro >>> '29:spike_frequency_adaptation': -0.016055090374903703,
pyelectro >>> '29:spike_width_adaptation': 0.00754720476623759,
pyelectro >>> '29:trough_decay_exponent': -0.05167436400749407,
pyelectro >>> '29:trough_phase_adaptation': 0.006928896725677521}

Going over acquisition # 30
pyelectro >>> { '30:average_last_1percent': -60.68766657511393,
pyelectro >>> '30:average_maximum': 48.13843,
pyelectro >>> '30:average_minimum': -29.626032,
pyelectro >>> '30:first_spike_time': 92.80000000000001,
pyelectro >>> '30:interspike_time_covar': 0.34247923269889735,
pyelectro >>> '30:max_interspike_time': 103.89999999999998,
pyelectro >>> '30:max_peak_no': 15,
pyelectro >>> '30:maximum': 61.767582,
pyelectro >>> '30:mean_spike_frequency': 15.688032272523532,
pyelectro >>> '30:min_interspike_time': 24.59999999999994,
pyelectro >>> '30:min_peak_no': 14,
pyelectro >>> '30:minimum': -61.248783,
pyelectro >>> '30:peak_decay_exponent': -0.19194537406993217,
pyelectro >>> '30:peak_linear_gradient': -0.006994401512067068,
pyelectro >>> '30:spike_broadening': 0.5770492253613585,
pyelectro >>> '30:spike_frequency_adaptation': -0.01538364588730742,
pyelectro >>> '30:spike_width_adaptation': 0.0069293286774622966,
pyelectro >>> '30:trough_decay_exponent': -0.04326627973106321,
pyelectro >>> '30:trough_phase_adaptation': 0.0064812799362231775}

Going over acquisition # 31
pyelectro >>> { '31:average_last_1percent': -60.63212458292643,
pyelectro >>> '31:average_maximum': 48.024498,
pyelectro >>> '31:average_minimum': -29.024399,
pyelectro >>> '31:first_spike_time': 92.4,
pyelectro >>> '31:interspike_time_covar': 0.406847478416553,
pyelectro >>> '31:max_interspike_time': 133.5999999999999,
pyelectro >>> '31:max_peak_no': 15,
pyelectro >>> '31:maximum': 61.889652,
pyelectro >>> '31:mean_spike_frequency': 14.704337779644996,
pyelectro >>> '31:min_interspike_time': 29.5,
pyelectro >>> '31:min_peak_no': 14,
pyelectro >>> '31:minimum': -61.30982,
pyelectro >>> '31:peak_decay_exponent': -0.1671016453568311,
pyelectro >>> '31:peak_linear_gradient': -0.0086990196695813,
pyelectro >>> '31:spike_broadening': 0.5569432887124698,
pyelectro >>> '31:spike_frequency_adaptation': -0.014767300558908368,
pyelectro >>> '31:spike_width_adaptation': 0.006743383637276833,
pyelectro >>> '31:trough_decay_exponent': -0.04051025499900451,

```

(continues on next page)

(continued from previous page)

```

pyelectro >>>      '31:trough_phase_adaptation': 0.006526251236739548}
Going over acquisition # 32
pyelectro >>> {   '32:average_last_1percent': -59.76054255167643,
pyelectro >>>     '32:average_maximum': 47.896324,
pyelectro >>>     '32:average_minimum': -27.88653,
pyelectro >>>     '32:first_spike_time': 91.30000000000001,
pyelectro >>>     '32:interspike_time_covar': 0.34354448310799324,
pyelectro >>>     '32:max_interspike_time': 106.60000000000002,
pyelectro >>>     '32:max_peak_no': 15,
pyelectro >>>     '32:maximum': 62.469486,
pyelectro >>>     '32:mean_spike_frequency': 15.222355115798628,
pyelectro >>>     '32:min_interspike_time': 30.700000000000003,
pyelectro >>>     '32:min_peak_no': 14,
pyelectro >>>     '32:minimum': -60.42481,
pyelectro >>>     '32:peak_decay_exponent': -0.2052962133940038,
pyelectro >>>     '32:peak_linear_gradient': -0.00818069814788547,
pyelectro >>>     '32:spike_broadening': 0.5505132639070699,
pyelectro >>>     '32:spike_frequency_adaptation': -0.0127147931736426,
pyelectro >>>     '32:spike_width_adaptation': 0.006794945084249822,
pyelectro >>>     '32:trough_decay_exponent': -0.045165955567810896,
pyelectro >>>     '32:trough_phase_adaptation': 0.00648520248429949}
Going over acquisition # 33
pyelectro >>> {   '33:average_last_1percent': -59.76237360636393,
pyelectro >>>     '33:average_maximum': 47.544212,
pyelectro >>>     '33:average_minimum': -27.22403,
pyelectro >>>     '33:first_spike_time': 91.4,
pyelectro >>>     '33:interspike_time_covar': 0.9530683477414146,
pyelectro >>>     '33:max_interspike_time': 322.6,
pyelectro >>>     '33:max_peak_no': 14,
pyelectro >>>     '33:maximum': 62.28638,
pyelectro >>>     '33:mean_spike_frequency': 13.151239251390995,
pyelectro >>>     '33:min_interspike_time': 29.0,
pyelectro >>>     '33:min_peak_no': 13,
pyelectro >>>     '33:minimum': -60.42481,
pyelectro >>>     '33:peak_decay_exponent': -0.21944646857580366,
pyelectro >>>     '33:peak_linear_gradient': -0.00986060329457358,
pyelectro >>>     '33:spike_broadening': 0.5524597059029492,
pyelectro >>>     '33:spike_frequency_adaptation': -0.01666625273183203,
pyelectro >>>     '33:spike_width_adaptation': 0.006743589954469429,
pyelectro >>>     '33:trough_decay_exponent': -0.03685620725832345,
pyelectro >>>     '33:trough_phase_adaptation': -0.012225503917922204}

```

We now have the following information:

- `analysis_results`: the results of the analysis by PyElectro; we need these to set the target values for our fitting
- `currents`: the value of stimulation current for each sweep we've chosen; we need this for our models
- `memb_vals`: the time series of the membrane potentials and recordings times; we'll use this to plot the membrane potentials later to compare our fitted model against

9.2 Running the optimisation

To run the optimisation, we want to choose which of the 33 time series we want to fit our model against. Ideally, we would want to fit our model to all of them. Here, however, for simplicity and to keep the computation time in check, we only pick two of the 33 sweeps. (As an exercise, you can change the list to see how that affects your fitting.)

```
sweeps_to_tune_against = [16, 21]
report = tune_izh_model(sweeps_to_tune_against, analysis_results, currents)
```

The Neurotune optimiser uses the evolutionary computation method provided by the [Inspyred](#) package. In short:

- the evolutionary algorithm starts with a population of models, each with a random value for a set of parameters constrained by a max/min value we have supplied
- it then calculates a fitness value for each model by comparing the features generated by the model to the target features that we provide
- in each generation, it finds the fittest models (parents)
- it mutates these to generate the next generation of models (offspring)
- it replaces the least fit models with fittest of the new individuals

The idea is that by calculating the fittest parents and offspring, it will find the candidate models that fit the provided target data best. You can read more about evolutionary computation online (e.g. [Wikipedia](#)). More information on model fitting in computational neuroscience can also be found in the literature. For example, see this review [[PBM04](#), [RGF+11](#)].

Here, we follow the following steps:

- we set up a template NeuroML model that will be passed to the optimiser
- we list the parameters we want to fit, and provide the extents of their state spaces
- we list the target features that the optimiser will use to calculate fitness, and set their weights
- finally, we use the `run_optimisation` function to run the optimisation

The `tune_izh_model` function shown below is the main workhorse function that does our fitting:

```
def tune_izh_model(acq_list: List, metrics_from_data: Dict, currents: Dict) -> Dict:
    """Tune networks model against the data.

    Here we generate a network with the necessary number of Izhikevich cells,
    one for each current stimulus, and tune them against the experimental data.

    :param acq_list: list of indices of acquisitions/sweeps to tune against
    :type acq_list: list
    :param metrics_from_data: dictionary with the sweep number as index, and
        the dictionary containing metrics generated from the analysis
    :type metrics_from_data: dict
    :param currents: dictionary with sweep number as index and stimulus current
        value
    """

    # length of simulation of the cells---should match the length of the
    # experiment
    sim_time = 1500.0
    # Create a NeuroML template network simulation file that we will use for
    # the tuning
    template_doc = NeuroMLDocument(id="IzhTuneNet")
```

(continues on next page)

(continued from previous page)

```

# Add an Izhikevich cell with some parameters to the document
template_doc.izhikevich2007_cells.append(
    Izhikevich2007Cell(
        id="Izh2007",
        C="100pF",
        v0="-60mV",
        k="0.7nS_per_mV",
        vr="-60mV",
        vt="-40mV",
        vpeak="35mV",
        a="0.03per_ms",
        b="-2ns",
        c="-50.0mV",
        d="100pA",
    )
)
template_doc.networks.append(Network(id="Network0"))
# Add a cell for each acquisition list
popsize = len(acq_list)
template_doc.networks[0].populations.append(
    Population(id="Pop0", component="Izh2007", size=popsize)
)

# Add a current source for each cell, matching the currents that
# were used in the experimental study.
counter = 0
for acq in acq_list:
    template_doc.pulse_generators.append(
        PulseGenerator(
            id="Stim{}".format(counter),
            delay="80ms",
            duration="1000ms",
            amplitude="{}pA".format(currents[acq]),
        )
    )
    template_doc.networks[0].explicit_inputs.append(
        ExplicitInput(
            target="Pop0[{}]".format(counter), input="Stim{}".format(counter)
        )
    )
    counter = counter + 1

# Print a summary
print(template_doc.summary())

# Write to a neuroml file and validate it.
reference = "TuneIzhFergusonPyr3"
template_filename = "{}.net.nml".format(reference)
write_neuroml2_file(template_doc, template_filename, validate=True)

# Now for the tuning bits

# format is type:id/variable:id/units
# supported types: cell/channel/izhikevich2007cell
# supported variables:
# - channel: vShift

```

(continues on next page)

(continued from previous page)

```

# - cell: channelDensity, vShift_channelDensity, channelDensityNernst,
#   erev_id, erev_ion, specificCapacitance, resistivity
# - izhikevich2007Cell: all available attributes

# we want to tune these parameters within these ranges
# param: (min, max)
parameters = {
    "izhikevich2007Cell:Izh2007/C/pF": (100, 300),
    "izhikevich2007Cell:Izh2007/k/nS_per_mV": (0.01, 2),
    "izhikevich2007Cell:Izh2007/vr/mV": (-70, -50),
    "izhikevich2007Cell:Izh2007/vt/mV": (-60, 0),
    "izhikevich2007Cell:Izh2007/vpeak/mV": (35, 70),
    "izhikevich2007Cell:Izh2007/a/per_ms": (0.001, 0.4),
    "izhikevich2007Cell:Izh2007/b/nS": (-10, 10),
    "izhikevich2007Cell:Izh2007/c/mV": (-65, -10),
    "izhikevich2007Cell:Izh2007/d/pA": (50, 500),
} # type: Dict[str, Tuple[float, float]]

# Set up our target data and so on
ctr = 0
target_data = {}
weights = {}
for acq in acq_list:
    # data to fit to:
    # format: path/to/variable:metric
    # metric from pyelectro, for example:
    # https://pyelectro.readthedocs.io/en/latest/pyelectro.html?highlight=mean_
    # spike_frequency#pyelectro.analysis.mean_spike_frequency
    mean_spike_frequency = "Pop0[{}]/v:mean_spike_frequency".format(ctr)
    average_last_1percent = "Pop0[{}]/v:average_last_1percent".format(ctr)
    first_spike_time = "Pop0[{}]/v:first_spike_time".format(ctr)

    # each metric can have an associated weight
    weights[mean_spike_frequency] = 1
    weights[average_last_1percent] = 1
    weights[first_spike_time] = 1

    # value of the target data from our data set
    target_data[mean_spike_frequency] = metrics_from_data[acq][
        "{}:mean_spike_frequency".format(acq)
    ]
    target_data[average_last_1percent] = metrics_from_data[acq][
        "{}:average_last_1percent".format(acq)
    ]
    target_data[first_spike_time] = metrics_from_data[acq][
        "{}:first_spike_time".format(acq)
    ]

    # only add these if the experimental data includes them
    # these are only generated for traces with spikes
    if "{}:average_maximum".format(acq) in metrics_from_data[acq]:
        average_maximum = "Pop0[{}]/v:average_maximum".format(ctr)
        weights[average_maximum] = 1
        target_data[average_maximum] = metrics_from_data[acq][
            "{}:average_maximum".format(acq)
        ]

```

(continues on next page)

(continued from previous page)

```

if "{}:average_minimum".format(acq) in metrics_from_data[acq]:
    average_minimum = "Pop0[{}]/v:average_minimum".format(ctr)
    weights[average_minimum] = 1
    target_data[average_minimum] = metrics_from_data[acq][
        "{}:average_minimum".format(acq)
    ]

    ctr = ctr + 1

# simulator to use
simulator = "jNeuroML"

return run_optimisation(
    # Prefix for new files
    prefix="TuneIzh",
    # Name of the NeuroML template file
    neuroml_file=template_filename,
    # Name of the network
    target="Network0",
    # Parameters to be fitted
    parameters=list(parameters.keys()),
    # Our max and min constraints
    min_constraints=[v[0] for v in parameters.values()],
    max_constraints=[v[1] for v in parameters.values()],
    # Weights we set for parameters
    weights=weights,
    # The experimental metrics to fit to
    target_data=target_data,
    # Simulation time
    sim_time=sim_time,
    # EC parameters
    population_size=100,
    max_evaluations=500,
    num_selected=30,
    num_offspring=50,
    mutation_rate=0.9,
    num_elites=3,
    # Seed value
    seed=12345,
    # Simulator
    simulator=simulator,
    dt=0.025,
    show_plot_already='--nogui' not in sys.argv,
    save_to_file="fitted_izhikevich_fitness.png",
    save_to_file_scatter="fitted_izhikevich_scatter.png",
    save_to_file_hist="fitted_izhikevich_hist.png",
    save_to_file_output="fitted_izhikevich_output.png",
    num_parallel_evaluations=4,
)

```

Let us walk through the different sections of this function.

9.2.1 Writing a template model

In this example, we want to fit the parameters of an *Izhikevich cell* to our data such that simulating the cell then gives us membrane potentials similar to those observed in the experiment. Following the *Izhikevich network example*, we set up a template network with one Izhikevich cell for each experimental recording that we want to fit. For each of these cells, we provide a current stimulus matching the current used in the current clamp experiments that we obtained our recordings from:

```
# length of simulation of the cells---should match the length of the
# experiment
sim_time = 1500.0
# Create a NeuroML template network simulation file that we will use for
# the tuning
template_doc = NeuroMLDocument(id="IzhTuneNet")
# Add an Izhikevich cell with some parameters to the document
template_doc.izhikevich2007_cells.append(
    Izhikevich2007Cell(
        id="Izh2007",
        C="100pF",
        v0="-60mV",
        k="0.7nS_per_mV",
        vr="-60mV",
        vt="-40mV",
        vpeak="35mV",
        a="0.03per_ms",
        b="-2ns",
        c="-50.0mV",
        d="100pA",
    )
)
template_doc.networks.append(Network(id="Network0"))
# Add a cell for each acquisition list
popsize = len(acq_list)
template_doc.networks[0].populations.append(
    Population(id="Pop0", component="Izh2007", size=popsize)
)

# Add a current source for each cell, matching the currents that
# were used in the experimental study.
counter = 0
for acq in acq_list:
    template_doc.pulse_generators.append(
        PulseGenerator(
            id="Stim{}".format(counter),
            delay="80ms",
            duration="1000ms",
            amplitude="{{$}}pA".format(currents[acq]),
        )
    )
template_doc.networks[0].explicit_inputs.append(
    ExplicitInput(
        target="Pop0[{}].format(counter), input="Stim{}".format(counter)
    )
)
counter = counter + 1

# Print a summary
```

(continues on next page)

(continued from previous page)

```

print(template_doc.summary())

# Write to a neuroml file and validate it.
reference = "TuneIzhFergusonPyr3"
template_filename = "{}.net.nml".format(reference)
write_neuroml2_file(template_doc, template_filename, validate=True)

```

The resultant network template model for our two chosen recordings is shown below:

```

<neuroml xmlns="http://www.neuroml.org/schema/neuroml2" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.neuroml.org/schema/neuroml2 https://raw.github.com/NeuroML/NeuroML2/development/Schemas/NeuroML2/NeuroML_v2.2.xsd" id="IzhTuneNet">
    <izhikevich2007Cell id="Izh2007" C="100pF" v0="-60mV" k="0.7nS_per_mV" vr="-60mV" vt="-40mV" vpeak="35mV" a="0.03per_ms" b="-2nS" c="-50.0mV" d="100pA"/>
        <pulseGenerator id="Stim0" delay="80ms" duration="1000ms" amplitude="152.0pA"/>
        <pulseGenerator id="Stim1" delay="80ms" duration="1000ms" amplitude="202.0pA"/>
    <network id="Network0">
        <population id="Pop0" component="Izh2007" size="2"/>
        <explicitInput target="Pop0[0]" input="Stim0"/>
        <explicitInput target="Pop0[1]" input="Stim1"/>
    </network>
</neuroml>

```

Please note that the initial parameters of the Izhikevich Cell do not matter here because the optimiser will modify these to run the candidate simulations.

9.2.2 Setting up the optimisation parameters

The next step is to set the features/metrics that we want to fit:

The `parameters` dictionary contains the specifications of the parameters that we wish to fit, along with their minimum and maximum permitted values.

```

# we want to tune these parameters within these ranges
# param: (min, max)
parameters = {
    "izhikevich2007Cell:Izh2007/C/pF": (100, 300),
    "izhikevich2007Cell:Izh2007/k/nS_per_mV": (0.01, 2),
    "izhikevich2007Cell:Izh2007/vr/mV": (-70, -50),
    "izhikevich2007Cell:Izh2007/vt/mV": (-60, 0),
    "izhikevich2007Cell:Izh2007/vpeak/mV": (35, 70),
    "izhikevich2007Cell:Izh2007/a/per_ms": (0.001, 0.4),
    "izhikevich2007Cell:Izh2007/b/nS": (-10, 10),
    "izhikevich2007Cell:Izh2007/c/mV": (-65, -10),
    "izhikevich2007Cell:Izh2007/d/pA": (50, 500),
}
# type: Dict[str, Tuple[float, float]]

```

The format of the parameter specification is: `ComponentType:ComponentID/VariableName[:VariableID]/Units`. So, for example, to fit the Capacitance of the Izhikevich cell, our parameter specification string is: `izhikevich2007Cell:Izh2007/C/pF`.

All NeuroML `Cell` and `Channel` ComponentTypes can be fitted using the NeuroMLTuner.

Next, we specify the target data that we want to fit against.

```

# Set up our target data and so on
ctr = 0
target_data = {}
weights = {}
for acq in acq_list:
    # data to fit to:
    # format: path/to/variable:metric
    # metric from pyelectro, for example:
    # https://pyelectro.readthedocs.io/en/latest/pyelectro.html?highlight=mean_
    # spike_frequency#pyelectro.analysis.mean_spike_frequency
    mean_spike_frequency = "Pop0[{}]/v:mean_spike_frequency".format(ctr)
    average_last_1percent = "Pop0[{}]/v:average_last_1percent".format(ctr)
    first_spike_time = "Pop0[{}]/v:first_spike_time".format(ctr)

    # each metric can have an associated weight
    weights[mean_spike_frequency] = 1
    weights[average_last_1percent] = 1
    weights[first_spike_time] = 1

    # value of the target data from our data set
    target_data[mean_spike_frequency] = metrics_from_data[acq][
        "{}:mean_spike_frequency".format(acq)]
    ]
    target_data[average_last_1percent] = metrics_from_data[acq][
        "{}:average_last_1percent".format(acq)]
    ]
    target_data[first_spike_time] = metrics_from_data[acq][
        "{}:first_spike_time".format(acq)]
    ]

    # only add these if the experimental data includes them
    # these are only generated for traces with spikes
    if "{}:average_maximum".format(acq) in metrics_from_data[acq]:
        average_maximum = "Pop0[{}]/v:average_maximum".format(ctr)
        weights[average_maximum] = 1
        target_data[average_maximum] = metrics_from_data[acq][
            "{}:average_maximum".format(acq)]
        ]
    if "{}:average_minimum".format(acq) in metrics_from_data[acq]:
        average_minimum = "Pop0[{}]/v:average_minimum".format(ctr)
        weights[average_minimum] = 1
        target_data[average_minimum] = metrics_from_data[acq][
            "{}:average_minimum".format(acq)]
        ]

    ctr = ctr + 1

```

As we have set up a cell for each recording that we want to fit to, we must also set the target value for each cell. We pick four features from a subset of features that PyElectro provided us with:

- mean_spike_frequency
- average_last_1percent
- average_maximum
- average_minimum

The last two can only be calculated for membrane potential data that includes spikes. Since a few of the experimental

recordings to not show any spikes, these two metrics will not be calculated for them. So, we only add them for the corresponding cell only if they are present in the features for the chosen recording.

The format for the `target_data` is similar to that of the `parameters`. The keys of the `target_data` dictionary are the specifications for the metrics. The format for these is: `path/to/variable:pyelectro metric`. You can learn more about constructing paths in NeuroML [here](#). The value for each key is the corresponding metric that was calculated for us by PyElectro (in `analysis_results`). The for loop will set the `target_data` to this (printed by `pyNeuroML` when we run the script):

```
target_data = {
    'Pop0[0]/v:mean_spike_frequency': 7.033585370142431,
    'Pop0[0]/v:average_last_1percent': -60.84635798136393,
    'Pop0[0]/v:average_maximum': 58.73414,
    'Pop0[0]/v:average_minimum': -43.800358,
    'Pop0[1]/v:mean_spike_frequency': 10.8837614279495,
    'Pop0[1]/v:average_last_1percent': -60.380863189697266,
    'Pop0[1]/v:average_maximum': 54.52382,
    'Pop0[1]/v:average_minimum': -39.78882
}
```

Similarly, we also set up the weights for each target metric in the `weights` variable:

```
weights = {
    'Pop0[0]/v:mean_spike_frequency': 1,
    'Pop0[0]/v:average_last_1percent': 1,
    'Pop0[0]/v:average_maximum': 1,
    'Pop0[0]/v:average_minimum': 1,
    'Pop0[1]/v:mean_spike_frequency': 1,
    'Pop0[1]/v:average_last_1percent': 1,
    'Pop0[1]/v:average_maximum': 1,
    'Pop0[1]/v:average_minimum': 1
}
```

For simplicity, we set the weights for all as 1 here.

9.2.3 Calling the optimisation function

The last step is to call our `run_optimisation` function with the various parameters that we have set up. Here, for simplicity, we use the `jNeuroML` simulator. For multi-compartmental models, however, we will need to use the `jNeuroML_NEURON` simulator (since `jNeuroML` only supports single compartment simulations). A number of arguments to the function are specific to evolutionary computation, and their discussion is beyond the scope of this tutorial.

```
# simulator to use
simulator = "jNeuroML"

return run_optimisation(
    # Prefix for new files
    prefix="TuneIzh",
    # Name of the NeuroML template file
    neuroml_file=template_filename,
    # Name of the network
    target="Network0",
    # Parameters to be fitted
    parameters=list(parameters.keys()),
    # Our max and min constraints
```

(continues on next page)

(continued from previous page)

```

min_constraints=[v[0] for v in parameters.values()],
max_constraints=[v[1] for v in parameters.values()],
# Weights we set for parameters
weights=weights,
# The experimental metrics to fit to
target_data=target_data,
# Simulation time
sim_time=sim_time,
# EC parameters
population_size=100,
max_evaluations=500,
num_selected=30,
num_offspring=50,
mutation_rate=0.9,
num_elites=3,
# Seed value
seed=12345,
# Simulator
simulator=simulator,
dt=0.025,
show_plot_already='--nogui' not in sys.argv,
save_to_file="fitted_izhikevich_fitness.png",
save_to_file_scatter="fitted_izhikevich_scatter.png",
save_to_file_hist="fitted_izhikevich_hist.png",
save_to_file_output="fitted_izhikevich_output.png",
num_parallel_evaluations=4,
)

```

The `run_optimisation` function will print out the optimisation report, and also return it so that it can be stored in a variable for further use. The terminal output is shown below:

```

Ran 500 evaluations (pop: 100) in 582.205449 seconds (9.703424 mins total; 1.164411s per eval)

----- Best candidate -----
{
    'Pop0[0]/v:average_last_1percent': -59.276969863333285,
    'Pop0[0]/v:average_maximum': 47.35760225,
    'Pop0[0]/v:average_minimum': -53.95061271428572,
    'Pop0[0]/v:first_spike_time': 170.1,
    'Pop0[0]/v:interspike_time_covar': 0.1330373936860586,
    'Pop0[0]/v:max_interspike_time': 190.57499999999982,
    'Pop0[0]/v:max_peak_no': 8,
    'Pop0[0]/v:maximum': 47.427714,
    'Pop0[0]/v:mean_spike_frequency': 6.957040276293886,
    'Pop0[0]/v:min_interspike_time': 135.25000000000003,
    'Pop0[0]/v:min_peak_no': 7,
    'Pop0[0]/v:minimum': -68.13577,
    'Pop0[0]/v:peak_decay_exponent': 0.0003379360943630205,
    'Pop0[0]/v:peak_linear_gradient': -3.270149536895308e-05,
    'Pop0[0]/v:spike_broadening': 0.982357731987536,
    'Pop0[0]/v:spike_frequency_adaptation': -0.016935379943933133,
    'Pop0[0]/v:spike_width_adaptation': 0.011971808793771004,
    'Pop0[0]/v:trough_decay_exponent': -0.0008421760726029059,
    'Pop0[0]/v:trough_phase_adaptation': -0.014231837120099502,
    'Pop0[1]/v:average_last_1percent': -59.28251401166662,
    'Pop0[1]/v:average_maximum': 47.242452454545464,
}

```

(continues on next page)

(continued from previous page)

```
'Pop0[1]/v:average_minimum': -48.287914,
'Pop0[1]/v:first_spike_time': 146.7,
'Pop0[1]/v:interspike_time_covar': 0.01075626702836981,
'Pop0[1]/v:max_interspike_time': 91.67499999999998,
'Pop0[1]/v:max_peak_no': 11,
'Pop0[1]/v:maximum': 47.423363,
'Pop0[1]/v:mean_spike_frequency': 10.973033769511423,
'Pop0[1]/v:min_interspike_time': 88.20000000000002,
'Pop0[1]/v:min_peak_no': 10,
'Pop0[1]/v:minimum': -62.58064000000001,
'Pop0[1]/v:peak_decay_exponent': 0.0008036004162568405,
'Pop0[1]/v:peak_linear_gradient': -0.00012436953066659044,
'Pop0[1]/v:spike_broadening': 0.9877761288704633,
'Pop0[1]/v:spike_frequency_adaptation': 0.0064956079899488595,
'Pop0[1]/v:spike_width_adaptation': 0.008982392557695507,
'Pop0[1]/v:trough_decay_exponent': -0.004658690933014975,
'Pop0[1]/v:trough_phase_adaptation': 0.009514671770845617}
```

TARGETS:

```
{ 'Pop0[0]/v:average_last_1percent': -60.84635798136393,
'Pop0[0]/v:average_maximum': 58.73414,
'Pop0[0]/v:average_minimum': -43.800358,
'Pop0[0]/v:mean_spike_frequency': 7.033585370142431,
'Pop0[1]/v:average_last_1percent': -60.380863189697266,
'Pop0[1]/v:average_maximum': 54.52382,
'Pop0[1]/v:average_minimum': -39.78882,
'Pop0[1]/v:mean_spike_frequency': 10.8837614279495}
```

TUNED VALUES:

```
{ 'Pop0[0]/v:average_last_1percent': -59.276969863333285,
'Pop0[0]/v:average_maximum': 47.35760225,
'Pop0[0]/v:average_minimum': -53.95061271428572,
'Pop0[0]/v:mean_spike_frequency': 6.957040276293886,
'Pop0[1]/v:average_last_1percent': -59.28251401166662,
'Pop0[1]/v:average_maximum': 47.2424524545464,
'Pop0[1]/v:average_minimum': -48.287914,
'Pop0[1]/v:mean_spike_frequency': 10.973033769511423}
```

FITNESS: 0.003633

```
FITTEST: { 'izhikevich2007Cell:Izh2007/C/pF': 240.6982897890555,
'izhikevich2007Cell:Izh2007/a/per_ms': 0.03863507615280202,
'izhikevich2007Cell:Izh2007/b/nS': 2.0112449831346746,
'izhikevich2007Cell:Izh2007/c/mV': -43.069939785498356,
'izhikevich2007Cell:Izh2007/d/pA': 212.50982499591083,
'izhikevich2007Cell:Izh2007/k/nS_per_mV': 0.24113869560362797,
'izhikevich2007Cell:Izh2007/vpeak/mV': 47.44063356996336,
'izhikevich2007Cell:Izh2007/vr/mV': -59.283747806929135,
'izhikevich2007Cell:Izh2007/vt/mV': -48.9131459978619}
```

It will also generate a number of plots (shown below):

- showing the evolution of the parameters being fitted, with indications of the fitness value: larger circles mean more fitness
- the change in the overall fitness value as the population evolves
- distributions of the values of the parameters being fitted, with indications of the fitness value: darker lines mean

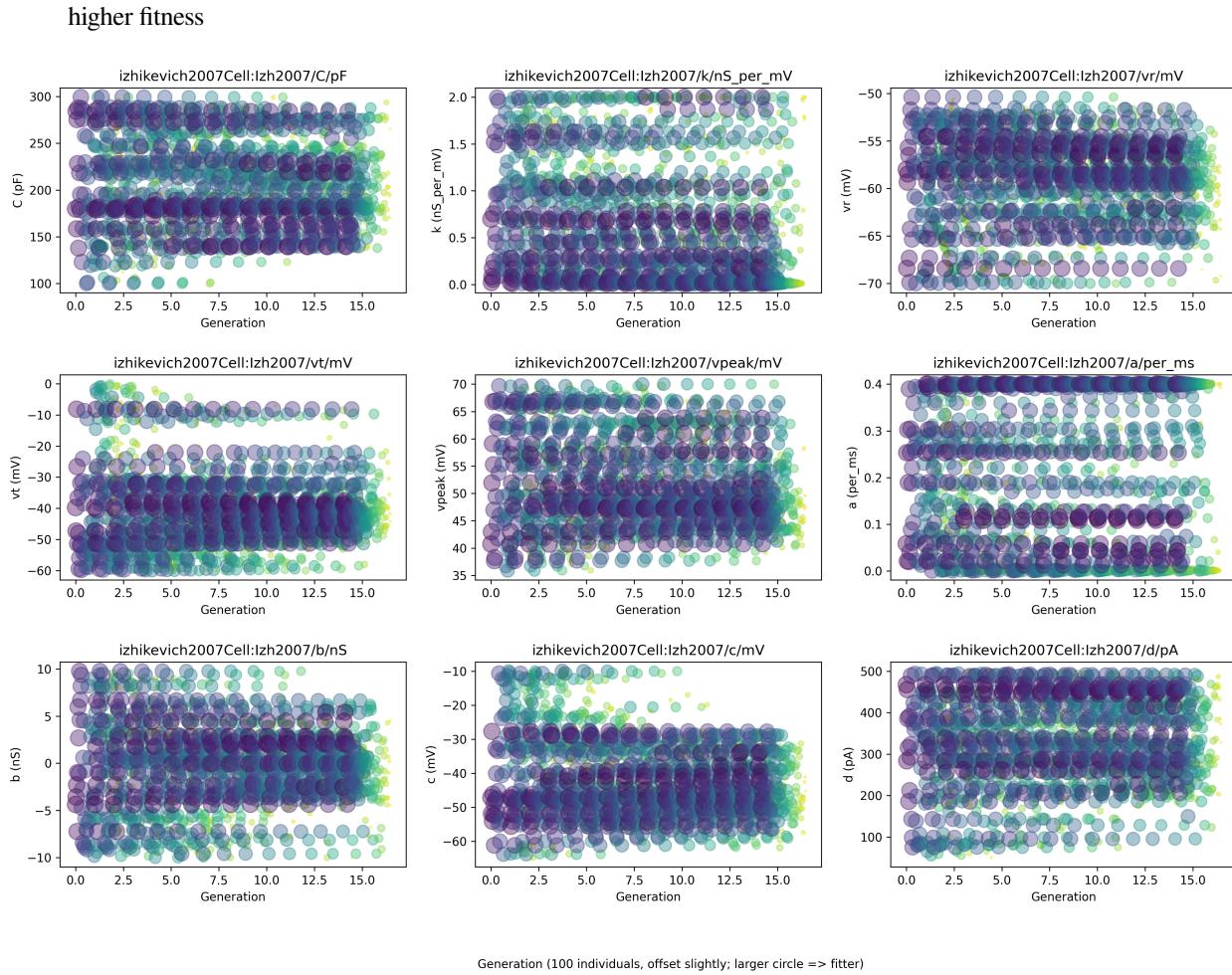


Fig. 9.4: The figure shows the values of various parameters throughout the evolution, with larger circles having higher values of fitness.

9.3 Viewing results

The tuner also generates a plot with the membrane potential of a cell using the fitted parameter values (shown on the top of the page). Here, to document how the fitted parameters are to be extracted from the output of the `run_optimisation` function, we also construct a model to use the fitted parameters ourselves and plot the membrane potential to compare it against the experimental data.

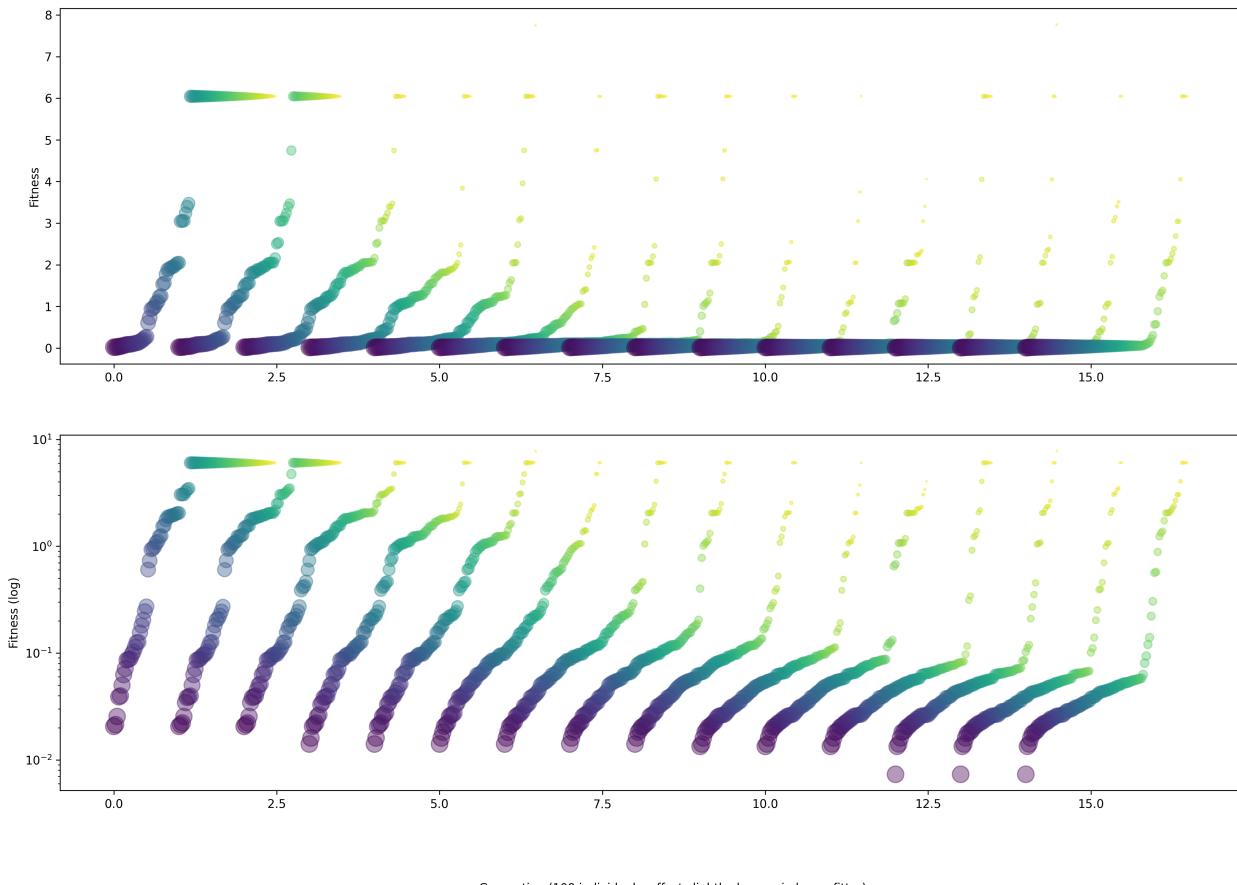


Fig. 9.5: The figure shows the trend of the fitness throughout the evolution.

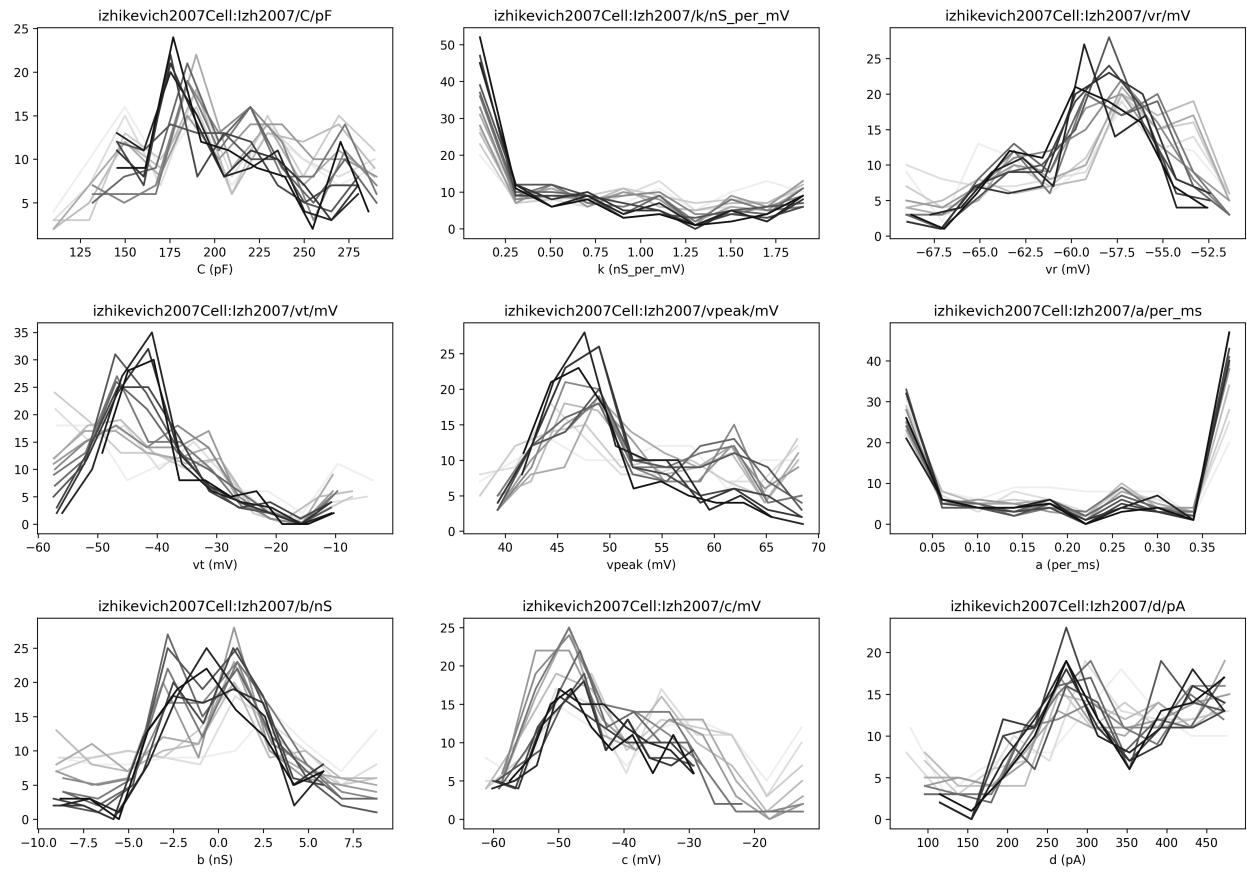


Fig. 9.6: The figure shows the distribution of values that for each parameter throughout the evolution. Darker lines have higher fitness values.

9.3.1 Extracting results and running a fitted model

This is done in the `run_fitted_cell_simulation` function:

```
def run_fitted_cell_simulation(
    sweeps_to_tune_against: List, tuning_report: Dict, simulation_id: str
) -> None:
    """Run a simulation with the values obtained from the fitting

    :param tuning_report: tuning report from the optimiser
    :type tuning_report: Dict
    :param simulation_id: text id of simulation
    :type simulation_id: str

    """
    # get the fittest variables
    fittest_vars = tuning_report["fittest vars"]
    C = str(fittest_vars["izhikevich2007Cell:Izh2007/C/pF"]) + "pF"
    k = str(fittest_vars["izhikevich2007Cell:Izh2007/k/nS_per_mV"]) + "nS_per_mV"
    vr = str(fittest_vars["izhikevich2007Cell:Izh2007/vr/mV"]) + "mV"
    vt = str(fittest_vars["izhikevich2007Cell:Izh2007/vt/mV"]) + "mV"
    vpeak = str(fittest_vars["izhikevich2007Cell:Izh2007/vpeak/mV"]) + "mV"
    a = str(fittest_vars["izhikevich2007Cell:Izh2007/a/per_ms"]) + "per_ms"
    b = str(fittest_vars["izhikevich2007Cell:Izh2007/b/nS"]) + "ns"
    c = str(fittest_vars["izhikevich2007Cell:Izh2007/c/mV"]) + "mV"
    d = str(fittest_vars["izhikevich2007Cell:Izh2007/d/pA"]) + "pA"

    # Create a simulation using our obtained parameters.
    # Note that the tuner generates a graph with the fitted values already, but
    # we want to keep a copy of our fitted cell also, so we'll create a NeuroML
    # Document ourselves also.
    sim_time = 1500.0
    simulation_doc = NeuroMLDocument(id="FittedNet")
    # Add an Izhikevich cell with some parameters to the document
    simulation_doc.izhikevich2007_cells.append(
        Izhikevich2007Cell(
            id="Izh2007",
            C=C,
            v0="-60mV",
            k=k,
            vr=vr,
            vt=vt,
            vpeak=vpeak,
            a=a,
            b=b,
            c=c,
            d=d,
        )
    )
    simulation_doc.networks.append(Network(id="Network0"))
    # Add a cell for each acquisition list
    popsize = len(sweeps_to_tune_against)
    simulation_doc.networks[0].populations.append(
        Population(id="Pop0", component="Izh2007", size=popsize)
    )

    # Add a current source for each cell, matching the currents that
```

(continues on next page)

(continued from previous page)

```

# were used in the experimental study.
counter = 0
for acq in sweeps_to_tune_against:
    simulation_doc.pulse_generators.append(
        PulseGenerator(
            id="Stim{}".format(counter),
            delay="80ms",
            duration="1000ms",
            amplitude="{{$}pA".format(currents[acq]),
        )
    )
    simulation_doc.networks[0].explicit_inputs.append(
        ExplicitInput(
            target="Pop0[{}]" .format(counter), input="Stim{}".format(counter)
        )
    )
    counter = counter + 1

# Print a summary
print(simulation_doc.summary())

# Write to a neuroml file and validate it.
reference = "FittedIzhFergusonPyr3"
simulation_filename = "{}.net.nml".format(reference)
write_neuroml2_file(simulation_doc, simulation_filename, validate=True)

simulation = LEMSSimulation(
    sim_id=simulation_id,
    duration=sim_time,
    dt=0.1,
    target="Network0",
    simulation_seed=54321,
)
simulation.include_neuroml2_file(simulation_filename)
simulation.create_output_file("output0", "{}.v.dat".format(simulation_id))
counter = 0
for acq in sweeps_to_tune_against:
    simulation.add_column_to_output_file(
        "output0", "Pop0[{}]" .format(counter), "Pop0[{}]/v".format(counter)
    )
    counter = counter + 1
simulation_file = simulation.save_to_file()
# simulate
run_lems_with_jneuroml(simulation_file, max_memory="2G", nogui=True, plot=False)

```

First, we extract the fitted parameters from the dictionary returned by the `run_optimisation` function. Then, we use these parameters to set up a simple NeuroML network and run a test simulation, recording the values of membrane potentials generated by the cells. Please note that the current stimulus to the cells in this test model must also match the values that were used in the experiment, and so also in the fitting.

9.3.2 Plotting model generated and experimentally recorded membrane potentials

Finally, in the `plot_sim_data` function, we plot the membrane potentials from our fitted cells and the experimental data to see visually inspect the results of our fitting:

```
def plot_sim_data(
    sweeps_to_tune_against: List, simulation_id: str, memb_pots: Dict
) -> None:
    """Plot data from our fitted simulation

    :param simulation_id: string id of simulation
    :type simulation_id: str
    """

    # Plot
    data_array = np.loadtxt("%s.v.dat" % simulation_id)

    # construct data for plotting
    counter = 0
    time_vals_list = []
    sim_v_list = []
    data_v_list = []
    data_t_list = []
    stim_vals = []
    for acq in sweeps_to_tune_against:
        stim_vals.append("{}pA".format(currents[acq]))

        # remains the same for all columns
        time_vals_list.append(data_array[:, 0] * 1000.0)
        sim_v_list.append(data_array[:, counter + 1] * 1000.0)

        data_v_list.append(memb_pots[acq][1])
        data_t_list.append(memb_pots[acq][0])

        counter = counter + 1

    # Model membrane potential plot
    generate_plot(
        xvalues=time_vals_list,
        yvalues=sim_v_list,
        labels=stim_vals,
        title="Membrane potential (model)",
        show_plot_already=False,
        save_figure_to="%s-model-v.png" % simulation_id,
        xaxis="time (ms)",
        yaxis="membrane potential (mV)",
    )
    # data membrane potential plot
    generate_plot(
        xvalues=data_t_list,
        yvalues=data_v_list,
        labels=stim_vals,
        title="Membrane potential (exp)",
        show_plot_already=False,
        save_figure_to="%s-exp-v.png" % simulation_id,
        xaxis="time (ms)",
        yaxis="membrane potential (mV)",
    )
```

(continues on next page)

(continued from previous page)

This generates the following figures:

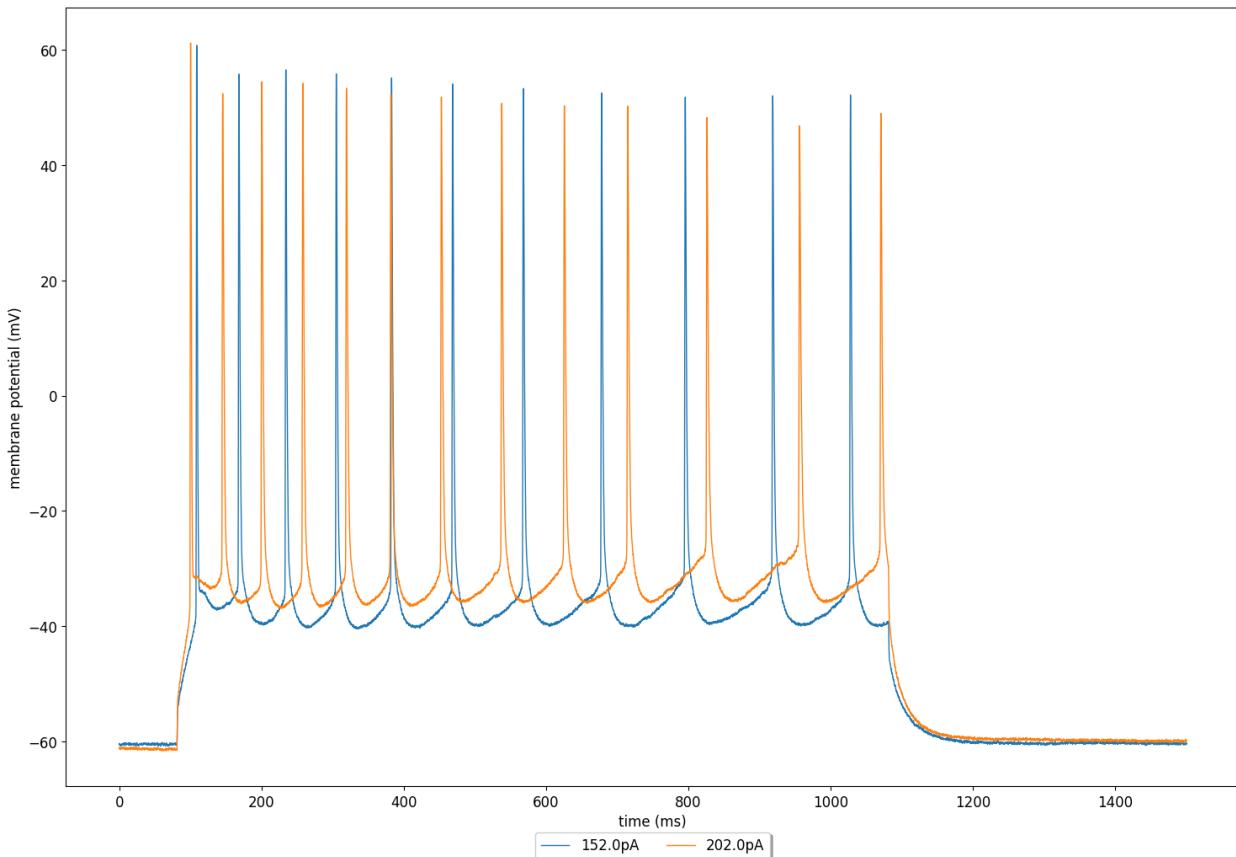


Fig. 9.7: Membrane potential from the experimental data.

We can clearly see the similarity between our fitted model and the experimental data. A number of tweaks can be made to improve the fitting. For example, pyNeuroML also provides a two staged optimisation function: `run_2stage_optimisation` that allows users to optimise sets of parameters in two different stages. The graphs also show ranges of parameters that provide fits, so users can also hand-tune their models further as required.

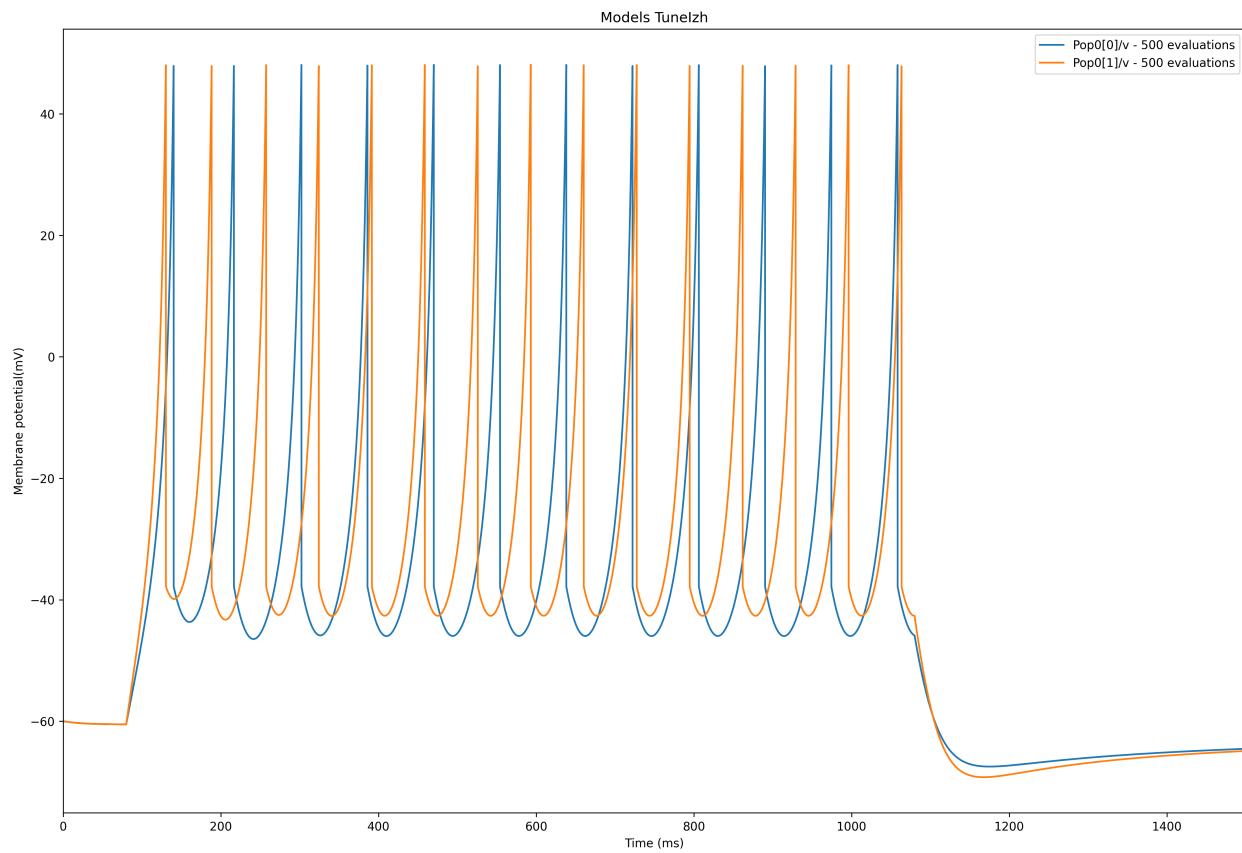


Fig. 9.8: Membrane potential obtained from the model with highest fitness.

CHAPTER
TEN

SCHEMA/SPECIFICATION

NeuroML v2.2 is the current stable release of the language, and is described below.

For an overview of the various releases of the language see: [A brief history of NeuroML](#).

We've briefly seen the XML representation of NeuroML models and simulations in the [Getting Started](#) tutorials. Here, we dive a little deeper into the underlying details of NeuroML.

XML itself does not define a set of standard tags: any tags may be used as long as the resultant document is [well-formed](#). Therefore, NeuroML defines a standard set of XML elements (the tags and attributes which specify the model and parameters, e.g. `<iafCell id="iaf" leakReversal="-60mV" ...>`) that may be used in NeuroML documents: the NeuroML [XML Schema Definition](#). This is referred to as the NeuroML *schema* or the NeuroML *specification*.

As the wiki page says:

XSD (XML Schema Definition), a recommendation of the World Wide Web Consortium (W3C), specifies how to formally describe the elements in an Extensible Markup Language (XML) document. It can be used by programmers to verify each piece of item content in a document, to assure it adheres to the description of the element it is placed in.

This gives us an idea of the advantages of using an XML based system. All NeuroML models must use these pre-defined tags/components—this is what we check for when we [validate NeuroML models](#). A valid NeuroML model is said to adhere to the NeuroML schema.

Purpose of the NeuroML specification/schema.

The NeuroML schema/specification defines the structure of a valid NeuroML document. The [core NeuroML tools](#) adhere to this specification and can read/write/interpret the language correctly.

In the next section, we learn more about the NeuroML 2 schema, and see how the dynamics of the NeuroML 2 entities are defined in LEMS.

10.1 NeuroML v2

The current stable version of NeuroML is v2.1, and can the schema for this be seen [here](#). The following figure, taken from Cannon et al. 2014 ([[CGC+14](#)]) shows some of the elements defined in NeuroML version 2 (note: these core elements haven't changed since that publication).

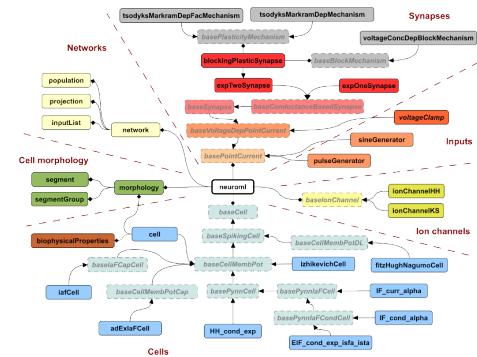


Fig. 10.1: Elements defined in the NeuroML schema, version 2.

You can see the complete definitions of NeuroML 2 entities in the following pages. You can also search this documentation for specific entities that you may be using in your NeuroML models.

Examples of files using the NeuroML 2 schema, and some of the elements they use are:

Example file	NeuroML elements used
A simple cell with a morphology & segments arranged into groups	<cell>, <morphology>, <segment>, <segmentGroup>
A cell specifying biophysical properties (channel densities, passive electrical properties, etc.)	<membraneProperties>, <intracellularProperties>, <channelDensity>
A simple HH Na ⁺ channel	<ionChannelHH>, <gateHHrates>, <HHExpLinearRate>
Some of the simplified spiking neuron models which are supported	<iafCell>, <zihikevich2007Cell>, <adExIaFCell>, <fitzHughNagumoCell>
Synapse models	<alphaSynapse>, <expTwoSynapse>, <blockingPlasticSynapse>, <doubleSynapse>
A network of cells positioned in 3D and synaptically connected	<network>, <population>, <projection>, <connection>, <inputList>

NeuroML files containing the XML representation of the model can be [validated](#) to ensure all of the correct tags/attributes are present.

But how do we know how the model is actually meant to use the specified attributes in an element? The schema only says that leakReversal, thresh, etc. are allowed attributes on iafCell, but how are these used to calculate the membrane potential? The answer lies in another, lower-level language, called LEMS (Low Entropy Model Specification).

10.1.1 Defining dynamics in LEMS

While valid NeuroML entities are contained in the schema, their underlying mathematical structure and composition rules must also be defined. For this, NeuroML version 2 makes use of [LEMS](#) (Low Entropy Language Specification).

LEMS

For an in-depth guide to LEMS, please see the research paper: [LEMS: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2](#). Documentation on the structure of the LEMS language can be found [here](#).

LEMS is an XML based language originally developed by Robert Cannon for specifying generic models of hybrid dynamical systems. Models defined in LEMS can also be simulated directly through a native interpreter.

- **ComponentType** elements define the behaviour of a specific type of model and specify **Parameters**, **StateVariables**, and their **Dynamics** and **Structure** can be defined as templates for model elements (e.g. HH ion channels, abstract cells, etc.). The notion of a **ComponentType** is thus similar to that of a **class** in object oriented programming.
- **Components** are instances of these types, with specific values of **Parameters** (e.g. HH squid axon Na⁺ channel, I&F cell with threshold -60mV, etc.). **Components** play the same role as **objects** in object oriented programming.

On the left side of the figure, examples are shown of the (truncated) XML representations of:

- (blue) a *network* containing two *populations* of *integrate-and-fire cells* connected by a single *projection* between them;
- (green) a *spiking neuron model* as described by Izhikevich (2003);
- (yellow) a *conductance based synapse* with a single exponential decay waveform.

On the right, the definition of the structure and dynamics of these elements in the LEMS language is shown. Each element has a corresponding **ComponentType** definition, describing the parameters (as well as their dimensions, not shown) and the dynamics in terms of state variables and their derivatives, any derived variables, and the behaviour when certain conditions are met or events are received (for example, the emission of a spike after a given threshold is crossed).

NeuroML 2 Component Type definitions in LEMS

The standard set of **ComponentType** definitions for the core NeuroML2 elements are contained in a curated set of files (below) though users are *free to define their own ComponentTypes to extend the scope of the language*.

- *Dimensions/units allowed* (source in LEMS)
- *Cell models* (source in LEMS)
- *Network elements* (source in LEMS)
- *Ion channels* (source in LEMS)
- *Synapse models* (source in LEMS)
- *Mapping of PyNN cells & synapses* (source in LEMS)

Here, for example, the *izhikevich2007Cell* is defined in the NeuroML schema as having the following internal attributes:

```
<xs:complexType name="Izhikevich2007Cell">
  <xs:complexContent>
    <xs:extension base="BaseCellMembPotCap">
      <xs:attribute name="v0" type="Nml2Quantity_voltage" use="required"/>
    
```

(continues on next page)

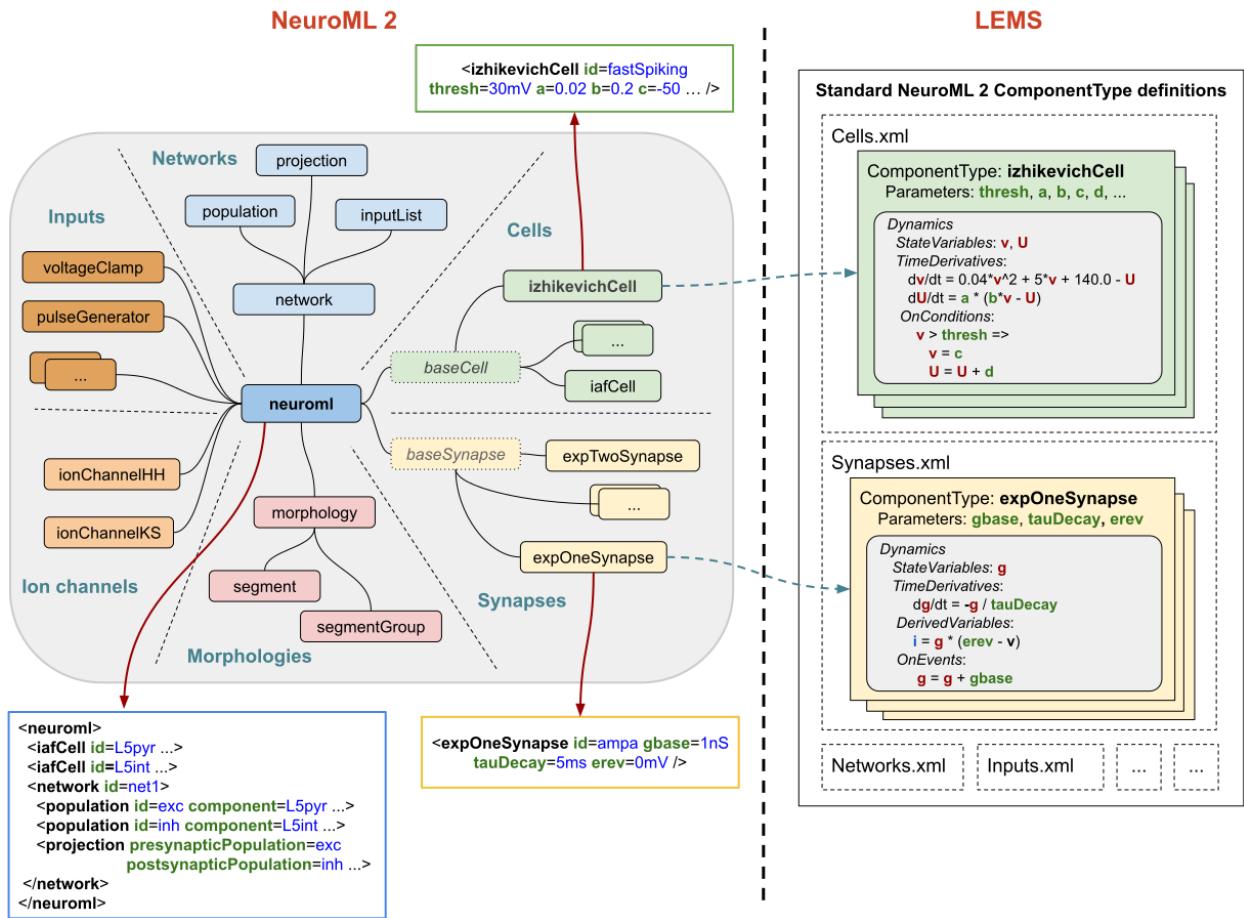


Fig. 10.2: This image (from Blundell et al. 2018 ([BBC+18])) shows the usage of LEMS **ComponentTypes** and **Components** in NeuroML. Elements in NeuroML v2 are **Components** which have a corresponding structural and mathematical definition in LEMS **ComponentTypes**.

(continued from previous page)

```

<xs:attribute name="k" type="Nml2Quantity_conductancePerVoltage" use=
↳ "required"/>
    <xs:attribute name="vr" type="Nml2Quantity_voltage" use="required"/>
    <xs:attribute name="vt" type="Nml2Quantity_voltage" use="required"/>
    <xs:attribute name="vpeak" type="Nml2Quantity_voltage" use="required"/>
    <xs:attribute name="a" type="Nml2Quantity_pertime" use="required"/>
    <xs:attribute name="b" type="Nml2Quantity_conductance" use="required"/>
    <xs:attribute name="c" type="Nml2Quantity_voltage" use="required"/>
    <xs:attribute name="d" type="Nml2Quantity_current" use="required"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
```

Correspondingly, its **ComponentType** dynamics are defined in the LEMS file, Cells.xml. (Note: you do not need to read the XML LEMS definitions, you can see this information in a well formatted form [here in the documentation itself](#))

```

<ComponentType name="izhikevich2007Cell"
    extends="baseCellMembPotCap"
    description="Cell based on the modified Izhikevich model in Izhikevich 2007, ↳
    Dynamical systems in neuroscience, MIT Press">

    <Parameter name="v0" dimension="voltage"/>

    <!--
    Defined in baseCellMembPotCap:
    <Parameter name="C" dimension="capacitance"/>
    -->
    <Parameter name="k" dimension="conductance_per_voltage"/>

    <Parameter name="vr" dimension="voltage"/>
    <Parameter name="vt" dimension="voltage"/>
    <Parameter name="vpeak" dimension="voltage"/>

    <Parameter name="a" dimension="per_time"/>
    <Parameter name="b" dimension="conductance"/>
    <Parameter name="c" dimension="voltage"/>
    <Parameter name="d" dimension="current"/>

    <Attachments name="synapses" type="basePointCurrent"/>

    <Exposure name="u" dimension="current"/>

    <Dynamics>

        <StateVariable name="v" dimension="voltage" exposure="v"/>
        <StateVariable name="u" dimension="current" exposure="u"/>

        <DerivedVariable name="iSyn" dimension="current" exposure="iSyn" select=
        ↳ "synapses[*]/i" reduce="add" />

        <DerivedVariable name="iMemb" dimension="current" exposure="iMemb" value="k *_
        ↳ (v-vr) * (v-vt) + iSyn - u"/>

        <TimeDerivative variable="v" value="iMemb / C"/>
        <TimeDerivative variable="u" value="a * (b * (v-vr) - u)"/>
```

(continues on next page)

(continued from previous page)

```

<OnStart>
    <StateAssignment variable="v" value="v0"/>
    <StateAssignment variable="u" value="0"/>
</OnStart>

<OnCondition test="v .gt. vpeak">
    <StateAssignment variable="v" value="c"/>
    <StateAssignment variable="u" value="u + d"/>
    <EventOut port="spike"/>
</OnCondition>

</Dynamics>

</ComponentType>

```

We can define **Components** of the *izhikevich2007Cell* **ComponentType** with the parameters we need. For example, the *izhikevich2007Cell* neuron model can exhibit different spiking behaviours, so we can define a regular spiking **Component**, or another **Component** that exhibits bursting.

```

<izhikevich2007Cell id="iz2007RS" v0 = "-60mV" C="100 pF" k = "0.7 nS_per_mV"
                     vr = "-60 mV" vt = "-40 mV" vpeak = "35 mV"
                     a = "0.03 per_ms" b = "-2 nS" c = "-50 mV" d = "100 pA"/>

```

Once these **Components** are defined in the NeuroML document, we can use **Instances** of them to create populations and networks, and so on.

You don't have to write in XML...

A quick reminder that while XML files can be edited in a standard text editor, you generally don't have to create/update them by hand. *This guide* goes through the steps of creating an example using the *izhikevich2007Cell* model in Python using *libNeuroML* and *pyNeuroML*

Using LEMS to specify the core of NeuroML version 2 has the following significant advantages:

- NeuroML 2 XML files can be used standalone by applications (exported/imported) in the same way as NeuroML v1.x, without reference to the LEMS definitions, easing the transition for v1.x compliant applications
- Any NeuroML 2 **ComponentType** can be extended and will be usable/translatable by any application (e.g. jLEMS) which understands LEMS

The first point above means that a parsing application does not necessarily have to natively read the LEMS type definition for, e.g. an *izhikevich2007Cell* element: it just has to map the NeuroML element parameters onto its own object model implementing that entity. Ideally, the behaviour should be the same – which could be ascertained by testing against the reference LEMS interpreter implementation ([jLEMS](#)).

The second point above means that if an application does support LEMS, it can automatically parse (and generate code for) a wide range of NeuroML 2 cells, channels and synapses, including any new **ComponentType** derived from these, without having to natively know anything about channels, cell models, etc.

jnml and pynml handle both LEMS and NeuroML 2.

jNeuroML and *pynml* handle both LEMS and NeuroML 2. They bundle jLEMS together with the LEMS definitions for NeuroML 2 ComponentTypes, and can simulate any LEMS model as well as many NeuroML 2 models.

10.1.2 NeuroMLCoreDimensions

Original ComponentType definitions: [NeuroMLCoreDimensions.xml](#). Schema against which NeuroML based on these should be valid: [NeuroML_v2.2.xsd](#). Generated on 07/06/22 from [this](#) commit. Please file any issues or questions at the issue tracker [here](#).

Dimensions

area

Dimensions L^2

Units

- Defined unit: [cm²](#)
- Defined unit: [m²](#)
- Defined unit: [um²](#)

capacitance

Dimensions $M^{-1} L^{-2} T^4 I^2$

Units

- Defined unit: [F](#)
- Defined unit: [nF](#)
- Defined unit: [pF](#)
- Defined unit: [uF](#)

charge

Dimensions $T^1 I^1$

Units

- Defined unit: [C](#)
- Defined unit: [e](#)

charge_per_mole

Dimensions $T^1 I^1 N^{-1}$

Units

- Defined unit: [C_per_mol](#)
- Defined unit: [nA_ms_per_amol](#)

concentration

Dimensions $L^{-3} N^1$

Units

- Defined unit: *M*
- Defined unit: *mM*
- Defined unit: *mol_per_cm3*
- Defined unit: *mol_per_m3*

conductance

Dimensions $M^{-1} L^{-2} T^3 I^2$

Units

- Defined unit: *S*
- Defined unit: *mS*
- Defined unit: *nS*
- Defined unit: *pS*
- Defined unit: *uS*

conductanceDensity

Dimensions $M^{-1} L^{-4} T^3 I^2$

Units

- Defined unit: *S_per_cm2*
- Defined unit: *S_per_m2*
- Defined unit: *mS_per_cm2*

conductance_per_voltage

Dimensions $M^{-2} L^{-4} T^6 I^3$

Units

- Defined unit: *S_per_V*
- Defined unit: *nS_per_mV*

currentDimensions I¹

Units

- Defined unit: *A*
- Defined unit: *nA*
- Defined unit: *pA*
- Defined unit: *uA*

currentDensityDimensions L⁻² I¹

Units

- Defined unit: *A_per_m2*
- Defined unit: *mA_per_cm2*
- Defined unit: *uA_per_cm2*

idealGasConstantDimsDimensions M¹ L² T⁻² K⁻¹ N⁻¹

Units

- Defined unit: *J_per_K_per_mol*

lengthDimensions L¹

Units

- Defined unit: *cm*
- Defined unit: *m*
- Defined unit: *um*

per_timeDimensions T⁻¹

Units

- Defined unit: *Hz*
- Defined unit: *per_hour*
- Defined unit: *per_min*
- Defined unit: *per_ms*

- Defined unit: *per_s*

per_voltage

Dimensions $M^{-1} L^{-2} T^3 I^1$

Units

- Defined unit: *per_V*
- Defined unit: *per_mV*

permeability

Dimensions $L^1 T^{-1}$

Units

- Defined unit: *cm_per_ms*
- Defined unit: *cm_per_s*
- Defined unit: *m_per_s*
- Defined unit: *um_per_ms*

resistance

Dimensions $M^1 L^2 T^{-3} I^{-2}$

Units

- Defined unit: *Mohm*
- Defined unit: *kohm*
- Defined unit: *ohm*

resistivity

Dimensions $M^2 L^2 T^{-3} I^{-2}$

Units

- Defined unit: *kohm_cm*
- Defined unit: *ohm_cm*
- Defined unit: *ohm_m*

rho_factorDimensions L⁻¹ T⁻¹ I¹ N¹

Units

- Defined unit: *mol_per_cm_per_uA_per_ms*
- Defined unit: *mol_per_m_per_A_per_s*

specificCapacitanceDimensions M⁻¹ L⁻⁴ T⁴ I²

Units

- Defined unit: *F_per_m2*
- Defined unit: *uF_per_cm2*

substanceDimensions N¹

Units

- Defined unit: *mol*

temperatureDimensions K¹

Units

- Defined unit: *K*
- Defined unit: *degC*

timeDimensions T¹

Units

- Defined unit: *hour*
- Defined unit: *min*
- Defined unit: *ms*
- Defined unit: *s*

voltage

Dimensions $M^1 L^2 T^{-3} I^{-1}$

Units

- Defined unit: V
- Defined unit: mV

volume

Dimensions L^3

Units

- Defined unit: cm^3
- Defined unit: $litre$
- Defined unit: m^3
- Defined unit: um^3

Units

A

Summary

- Dimension: $current$
- Power of 10: 0

Conversions

- $1 A = 1.00e+09 nA$
- $1 A = 1.00e+12 pA$
- $1 A = 1.00e+06 uA$

A_per_m2

Summary

- Dimension: $currentDensity$
- Power of 10: 0

Conversions

- $1 A_per_m2 = 0.1 mA_per_cm2$
- $1 A_per_m2 = 100 uA_per_cm2$

C

Summary

- Dimension: *charge*
- Power of 10: 0

Conversions

- $1 \text{ C} = 6.24 \times 10^{18} e$

C_per_mol

Summary

- Dimension: *charge_per_mole*
- Power of 10: 0

Conversions

- $1 \text{ C_per_mol} = 1 \times 10^{-6} nA_ms_per_amol$

F

Summary

- Dimension: *capacitance*
- Power of 10: 0

Conversions

- $1 \text{ F} = 1.00 \times 10^9 nF$
- $1 \text{ F} = 1.00 \times 10^{12} pF$
- $1 \text{ F} = 1.00 \times 10^6 uF$

F_per_m2

Summary

- Dimension: *specificCapacitance*
- Power of 10: 0

Conversions

- $1 \text{ F_per_m2} = 100 \text{ uF_per_cm2}$

Hz

Summary

- Dimension: *per_time*
- Power of 10: 0

Conversions

- 1 Hz = 3600 *per_hour*
- 1 Hz = 60 *per_min*
- 1 Hz = 0.001 *per_ms*
- 1 Hz = 1 *per_s*

J_per_K_per_mol

Summary

- Dimension: *idealGasConstantDims*
- Power of 10: 0

K

Summary

- Dimension: *temperature*
- Power of 10: 0

Conversions

- 1 K = -272.15 *degC*

M

Summary

- Dimension: *concentration*
- Power of 10: 3

Conversions

- 1 M = 1000 *mM*
- 1 M = 0.001 *mol_per_cm3*
- 1 M = 1000 *mol_per_m3*

Mohm

Summary

- Dimension: *resistance*
- Power of 10: 6

Conversions

- 1 Mohm = 1000 *kohm*
- 1 Mohm = 1.00e+06 *ohm*

S

Summary

- Dimension: *conductance*
- Power of 10: 0

Conversions

- 1 S = 1000 *mS*
- 1 S = 1.00e+09 *nS*
- 1 S = 1.00e+12 *pS*
- 1 S = 1.00e+06 *uS*

S_per_V

Summary

- Dimension: *conductance_per_voltage*
- Power of 10: 0

Conversions

- 1 S_per_V = 1.00e+06 *nS_per_mV*

S_per_cm2

Summary

- Dimension: *conductanceDensity*
- Power of 10: 4

Conversions

- 1 S_per_cm2 = 10000 *S_per_m2*
- 1 S_per_cm2 = 1000 *mS_per_cm2*

S_per_m2

Summary

- Dimension: *conductanceDensity*
- Power of 10: 0

Conversions

- $1 \text{ S_per_m}^2 = 0.0001 \text{ S_per_cm}^2$
- $1 \text{ S_per_m}^2 = 0.1 \text{ mS_per_cm}^2$

V

Summary

- Dimension: *voltage*
- Power of 10: 0

Conversions

- $1 \text{ V} = 1000 \text{ mV}$

cm

Summary

- Dimension: *length*
- Power of 10: -2

Conversions

- $1 \text{ cm} = 0.01 \text{ m}$
- $1 \text{ cm} = 10000 \text{ um}$

cm2

Summary

- Dimension: *area*
- Power of 10: -4

Conversions

- $1 \text{ cm}^2 = 0.0001 \text{ m}^2$
- $1 \text{ cm}^2 = 1.00e+08 \text{ um}^2$

cm3

Summary

- Dimension: *volume*
- Power of 10: -6

Conversions

- $1 \text{ cm}^3 = 0.001 \text{ litre}$
- $1 \text{ cm}^3 = 1\text{e-}06 \text{ m}^3$
- $1 \text{ cm}^3 = 1.00\text{e+}12 \text{ um}^3$

cm_per_ms

Summary

- Dimension: *permeability*
- Power of 10: 1

Conversions

- $1 \text{ cm_per_ms} = 1000 \text{ cm_per_s}$
- $1 \text{ cm_per_ms} = 10 \text{ m_per_s}$
- $1 \text{ cm_per_ms} = 10000 \text{ um_per_ms}$

cm_per_s

Summary

- Dimension: *permeability*
- Power of 10: -2

Conversions

- $1 \text{ cm_per_s} = 0.001 \text{ cm_per_ms}$
- $1 \text{ cm_per_s} = 0.01 \text{ m_per_s}$
- $1 \text{ cm_per_s} = 10 \text{ um_per_ms}$

degC

Summary

- Dimension: *temperature*
- Power of 10: 0
- Offset: 273.15

Conversions

- $1 \text{ degC} = 274.15 \text{ K}$

e

Summary

- Dimension: *charge*
- Power of 10: 0
- Scale: 1.602176634e-19

Conversions

- 1 e = 1.6022e-19 C

hour

Summary

- Dimension: *time*
- Power of 10: 0
- Scale: 3600.0

Conversions

- 1 hour = 60 min
- 1 hour = 3.60e+06 ms
- 1 hour = 3600 s

kohm

Summary

- Dimension: *resistance*
- Power of 10: 3

Conversions

- 1 kohm = 0.001 Mohm
- 1 kohm = 1000 ohm

kohm_cm

Summary

- Dimension: *resistivity*
- Power of 10: 1

Conversions

- 1 kohm_cm = 1000 ohm_cm
- 1 kohm_cm = 10 ohm_m

litre

Summary

- Dimension: *volume*
- Power of 10: -3

Conversions

- 1 litre = 1000 *cm*³
- 1 litre = 0.001 *m*³
- 1 litre = 1.00e+15 *um*³

m

Summary

- Dimension: *length*
- Power of 10: 0

Conversions

- 1 m = 100 *cm*
- 1 m = 1.00e+06 *um*

m2

Summary

- Dimension: *area*
- Power of 10: 0

Conversions

- 1 m² = 10000 *cm*²
- 1 m² = 1.00e+12 *um*²

m3

Summary

- Dimension: *volume*
- Power of 10: 0

Conversions

- 1 m³ = 1.00e+06 *cm*³
- 1 m³ = 1000 *litre*
- 1 m³ = 1.00e+18 *um*³

mA_per_cm2

Summary

- Dimension: *currentDensity*
- Power of 10: 1

Conversions

- $1 \text{ mA_per_cm}^2 = 10 \text{ A_per_m}^2$
- $1 \text{ mA_per_cm}^2 = 1000 \text{ uA_per_cm}^2$

mM

Summary

- Dimension: *concentration*
- Power of 10: 0

Conversions

- $1 \text{ mM} = 0.001 \text{ M}$
- $1 \text{ mM} = 1\text{e-}06 \text{ mol_per_cm}^3$
- $1 \text{ mM} = 1 \text{ mol_per_m}^3$

mS

Summary

- Dimension: *conductance*
- Power of 10: -3

Conversions

- $1 \text{ mS} = 0.001 \text{ S}$
- $1 \text{ mS} = 1.00\text{e+}06 \text{ nS}$
- $1 \text{ mS} = 1.00\text{e+}09 \text{ pS}$
- $1 \text{ mS} = 1000 \text{ uS}$

mS_per_cm2

Summary

- Dimension: *conductanceDensity*
- Power of 10: 1

Conversions

- $1 \text{ mS_per_cm}^2 = 0.001 \text{ S_per_cm}^2$
- $1 \text{ mS_per_cm}^2 = 10 \text{ S_per_m}^2$

mV

Summary

- Dimension: *voltage*
- Power of 10: -3

Conversions

- $1 \text{ mV} = 0.001 \text{ } V$

m_per_s

Summary

- Dimension: *permeability*
- Power of 10: 0

Conversions

- $1 \text{ m_per_s} = 0.1 \text{ cm_per_ms}$
- $1 \text{ m_per_s} = 100 \text{ cm_per_s}$
- $1 \text{ m_per_s} = 1000 \text{ um_per_ms}$

min

Summary

- Dimension: *time*
- Power of 10: 0
- Scale: 60.0

Conversions

- $1 \text{ min} = 0.016667 \text{ hour}$
- $1 \text{ min} = 6.00\text{e+04} \text{ ms}$
- $1 \text{ min} = 60 \text{ s}$

mol

Summary

- Dimension: *substance*
- Power of 10: 0

mol_per_cm3

Summary

- Dimension: *concentration*
- Power of 10: 6

Conversions

- $1 \text{ mol_per_cm}^3 = 1000 \text{ M}$
- $1 \text{ mol_per_cm}^3 = 1.00\text{e+}06 \text{ mM}$
- $1 \text{ mol_per_cm}^3 = 1.00\text{e+}06 \text{ mol_per_m}^3$

mol_per_cm_per_uA_per_ms

Summary

- Dimension: *rho_factor*
- Power of 10: 11

Conversions

- $1 \text{ mol_per_cm_per_uA_per_ms} = 1.00\text{e+}11 \text{ mol_per_m_per_A_per_s}$

mol_per_m3

Summary

- Dimension: *concentration*
- Power of 10: 0

Conversions

- $1 \text{ mol_per_m}^3 = 0.001 \text{ M}$
- $1 \text{ mol_per_m}^3 = 1 \text{ mM}$
- $1 \text{ mol_per_m}^3 = 1\text{e-}06 \text{ mol_per_cm}^3$

mol_per_m_per_A_per_s

Summary

- Dimension: *rho_factor*
- Power of 10: 0

Conversions

- $1 \text{ mol_per_m_per_A_per_s} = 1\text{e-}11 \text{ mol_per_cm_per_uA_per_ms}$

ms

Summary

- Dimension: *time*
- Power of 10: -3

Conversions

- $1 \text{ ms} = 2.7778 \times 10^{-7} \text{ hour}$
- $1 \text{ ms} = 1.6667 \times 10^{-5} \text{ min}$
- $1 \text{ ms} = 0.001 \text{ s}$

nA

Summary

- Dimension: *current*
- Power of 10: -9

Conversions

- $1 \text{ nA} = 1 \times 10^{-9} \text{ A}$
- $1 \text{ nA} = 1000 \text{ pA}$
- $1 \text{ nA} = 0.001 \text{ uA}$

nA_ms_per_amol

Summary

- Dimension: *charge_per_mole*
- Power of 10: 6

Conversions

- $1 \text{ nA_ms_per_amol} = 1.00 \times 10^6 \text{ C_per_mol}$

nF

Summary

- Dimension: *capacitance*
- Power of 10: -9

Conversions

- $1 \text{ nF} = 1 \times 10^{-9} \text{ F}$
- $1 \text{ nF} = 1000 \text{ pF}$
- $1 \text{ nF} = 0.001 \text{ uF}$

nS

Summary

- Dimension: *conductance*
- Power of 10: -9

Conversions

- $1 \text{ nS} = 1\text{e-}09 \text{ S}$
- $1 \text{ nS} = 1\text{e-}06 \text{ mS}$
- $1 \text{ nS} = 1000 \text{ pS}$
- $1 \text{ nS} = 0.001 \text{ uS}$

nS_per_mV

Summary

- Dimension: *conductance_per_voltage*
- Power of 10: -6

Conversions

- $1 \text{ nS_per_mV} = 1\text{e-}06 \text{ S_per_V}$

ohm

Summary

- Dimension: *resistance*
- Power of 10: 0

Conversions

- $1 \text{ ohm} = 1\text{e-}06 \text{ Mohm}$
- $1 \text{ ohm} = 0.001 \text{ kohm}$

ohm_cm

Summary

- Dimension: *resistivity*
- Power of 10: -2

Conversions

- $1 \text{ ohm_cm} = 0.001 \text{ kohm_cm}$
- $1 \text{ ohm_cm} = 0.01 \text{ ohm_m}$

ohm_m

Summary

- Dimension: *resistivity*
- Power of 10: 0

Conversions

- $1 \text{ ohm_m} = 0.1 \text{ kohm_cm}$
- $1 \text{ ohm_m} = 100 \text{ ohm_cm}$

pA

Summary

- Dimension: *current*
- Power of 10: -12

Conversions

- $1 \text{ pA} = 1\text{e-}12 \text{ A}$
- $1 \text{ pA} = 0.001 \text{ nA}$
- $1 \text{ pA} = 1\text{e-}06 \text{ uA}$

pF

Summary

- Dimension: *capacitance*
- Power of 10: -12

Conversions

- $1 \text{ pF} = 1\text{e-}12 \text{ F}$
- $1 \text{ pF} = 0.001 \text{ nF}$
- $1 \text{ pF} = 1\text{e-}06 \text{ uF}$

pS

Summary

- Dimension: *conductance*
- Power of 10: -12

Conversions

- $1 \text{ pS} = 1\text{e-}12 \text{ S}$
- $1 \text{ pS} = 1\text{e-}09 \text{ mS}$
- $1 \text{ pS} = 0.001 \text{ nS}$

- 1 pS = 1e-06 *uS*

per_V

Summary

- Dimension: *per_voltage*
- Power of 10: 0

Conversions

- 1 per_V = 0.001 *per_mV*

per_hour

Summary

- Dimension: *per_time*
- Power of 10: 0
- Scale: 0.00027777777778

Conversions

- 1 per_hour = 0.00027778 *Hz*
- 1 per_hour = 0.016667 *per_min*
- 1 per_hour = 2.7778e-07 *per_ms*
- 1 per_hour = 0.00027778 *per_s*

per_mV

Summary

- Dimension: *per_voltage*
- Power of 10: 3

Conversions

- 1 per_mV = 1000 *per_V*

per_min

Summary

- Dimension: *per_time*
- Power of 10: 0
- Scale: 0.0166666667

Conversions

- 1 per_min = 0.016667 *Hz*
- 1 per_min = 60 *per_hour*

- 1 per_min = 1.6667e-05 *per_ms*
- 1 per_min = 0.016667 *per_s*

per_ms

Summary

- Dimension: *per_time*
- Power of 10: 3

Conversions

- 1 per_ms = 1000 *Hz*
- 1 per_ms = 3.60e+06 *per_hour*
- 1 per_ms = 6.00e+04 *per_min*
- 1 per_ms = 1000 *per_s*

per_s

Summary

- Dimension: *per_time*
- Power of 10: 0

Conversions

- 1 per_s = 1 *Hz*
- 1 per_s = 3600 *per_hour*
- 1 per_s = 60 *per_min*
- 1 per_s = 0.001 *per_ms*

s

Summary

- Dimension: *time*
- Power of 10: 0

Conversions

- 1 s = 0.00027778 *hour*
- 1 s = 0.016667 *min*
- 1 s = 1000 *ms*

uA

Summary

- Dimension: *current*
- Power of 10: -6

Conversions

- $1 \text{ uA} = 1\text{e-}06 \text{ A}$
- $1 \text{ uA} = 1000 \text{ nA}$
- $1 \text{ uA} = 1.00\text{e+}06 \text{ pA}$

uA_per_cm2

Summary

- Dimension: *currentDensity*
- Power of 10: -2

Conversions

- $1 \text{ uA_per_cm2} = 0.01 \text{ A_per_m2}$
- $1 \text{ uA_per_cm2} = 0.001 \text{ mA_per_cm2}$

uF

Summary

- Dimension: *capacitance*
- Power of 10: -6

Conversions

- $1 \text{ uF} = 1\text{e-}06 \text{ F}$
- $1 \text{ uF} = 1000 \text{ nF}$
- $1 \text{ uF} = 1.00\text{e+}06 \text{ pF}$

uF_per_cm2

Summary

- Dimension: *specificCapacitance*
- Power of 10: -2

Conversions

- $1 \text{ uF_per_cm2} = 0.01 \text{ F_per_m2}$

uS

Summary

- Dimension: *conductance*
- Power of 10: -6

Conversions

- $1 \text{ uS} = 1\text{e-}06 \text{ S}$
- $1 \text{ uS} = 0.001 \text{ mS}$
- $1 \text{ uS} = 1000 \text{ nS}$
- $1 \text{ uS} = 1.00\text{e+}06 \text{ pS}$

um

Summary

- Dimension: *length*
- Power of 10: -6

Conversions

- $1 \text{ um} = 0.0001 \text{ cm}$
- $1 \text{ um} = 1\text{e-}06 \text{ m}$

um2

Summary

- Dimension: *area*
- Power of 10: -12

Conversions

- $1 \text{ um}^2 = 1\text{e-}08 \text{ cm}^2$
- $1 \text{ um}^2 = 1\text{e-}12 \text{ m}^2$

um3

Summary

- Dimension: *volume*
- Power of 10: -18

Conversions

- $1 \text{ um}^3 = 1\text{e-}12 \text{ cm}^3$
- $1 \text{ um}^3 = 1\text{e-}15 \text{ litre}$
- $1 \text{ um}^3 = 1\text{e-}18 \text{ m}^3$

um_per_ms

Summary

- Dimension: *permeability*
- Power of 10: -3

Conversions

- 1 um_per_ms = 0.0001 cm_per_ms
- 1 um_per_ms = 0.1 cm_per_s
- 1 um_per_ms = 0.001 m_per_s

10.1.3 NeuroMLCoreCompTypes

Original ComponentType definitions: [NeuroMLCoreCompTypes.xml](#). Schema against which NeuroML based on these should be valid: [NeuroML_v2.2.xsd](#). Generated on 07/06/22 from [this commit](#). Please file any issues or questions at the issue tracker [here](#).

notes

Human readable notes/description for a Component.

Usage: XML

```
<notes>A Simple Spiking cell for testing purposes</notes>
```

```
<notes>Multicompartmental cell</notes>
```

```
<notes>Leak conductance</notes>
```

annotation

A structured annotation containing metadata, specifically RDF or *property* elements.

Child list

<code>rdf:RDF</code>	<code>rdf_RDF</code>
----------------------	----------------------

Children list

prop- erty	<i>property</i>
-----------------------	-----------------

Usage: Python

Go to the [libNeuroML documentation](#)

```
from neuroml import Annotation

variable = Annotation(anytypeobjs_=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<annotation>
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"_
    xmlns:bqbiol="http://biomodels.net/biology-qualifiers/">
    <rdf:Description rdf:about="HippoCA1Cell">
      <bqbiol:is>
        <rdf:Bag>

          <rdf:li rdf:resource="urn:miriam:neurondb:258"/>
        </rdf:Bag>
      </bqbiol:is>
    </rdf:Description>
  </rdf:RDF>
</annotation>
```

property

A property (a **tag** and **value** pair), which can be on any *baseStandalone* either as a direct child, or within an *annotation*. Generally something which helps the visual display or facilitates simulation of a Component, but is not a core physiological property. Common examples include: **numberInternalDivisions**, equivalent of nseg in NEURON; **radius**, for a radius to use in graphical displays for abstract cells (i.e. without defined morphologies); **color**, the color to use for a *population* or *populationList* of cells; **recommended_dt_ms**, the recommended timestep to use for simulating a *network*, **recommended_duration_ms** the recommended duration to use when running a *network*.

Text fields

tag	Name of the property
value	Value of the property

Usage: Python

Go to the [libNeuroML documentation](#)

```
from neuroml import Property

variable = Property(tag=None, value=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<property tag="numberInternalDivisions" value="9"/>
```

baseStandalone

Base type of any Component which can have *notes*, *annotation*, or a *property* list.

Child list

<i>notes</i>		<i>notes</i>
<i>annotation</i>		<i>annotation</i>

Children list

prop- erty		<i>property</i>
-----------------------	--	-----------------

rdf_RDF

Structured block in an *annotation* based on RDF. See https://github.com/OpenSourceBrain/OSB_API/blob/master/python/examples/grancelllayer.xml.

Text fields

xmlns:rdf	
------------------	--

Child list

rdf:Description	<i>rdf_Description</i>
------------------------	------------------------

rdf_Description

Structured block in an *annotation* based on RDF.

Text fields

rdf:about	
------------------	--

Child list

bqbiol:encodes	<i>bqbiol_encodes</i>
bqbiol:hasPart	<i>bqbiol_hasPart</i>
bqbiol:hasProperty	<i>bqbiol_hasProperty</i>
bqbiol:hasVersion	<i>bqbiol_hasVersion</i>
bqbiol:is	<i>bqbiol_is</i>
bqbiol:isDescribedBy	<i>bqbiol_isDescribedBy</i>
bqbiol:isEncodedBy	<i>bqbiol_isEncodedBy</i>
bqbiol:isHomologTo	<i>bqbiol_isHomologTo</i>
bqbiol:isPartOf	<i>bqbiol_isPartOf</i>
bqbiol:isPropertyOf	<i>bqbiol_isPropertyOf</i>
bqbiol:isVersionOf	<i>bqbiol_isVersionOf</i>
bqbiol:occursIn	<i>bqbiol_occursIn</i>
bqbiol:hasTaxon	<i>bqbiol_hasTaxon</i>
bq-model:is	<i>bqmmodel_is</i>
bq-model:isDescribedBy	<i>bq-model_isDescribedBy</i>
bq-model:isDerivedFrom	<i>bq-model_isDerivedFrom</i>

baseBqbiol

Structured block in an *annotation* based on RDF.

Child list

rdf:Bag	<i>rdf_Bag</i>
----------------	----------------

bqbiol_encodes

extends [baseBqbiol](#)

See <http://co.mbine.org/standards/qualifiers>.

{tab-set}

bqbiol_hasPart

extends [baseBqbiol](#)

See <http://co.mbine.org/standards/qualifiers>.

{tab-set}

bqbiol_hasProperty

extends [baseBqbiol](#)

See <http://co.mbine.org/standards/qualifiers>.

{tab-set}

bqbiol_hasVersion

extends [baseBqbiol](#)

See <http://co.mbine.org/standards/qualifiers>.

{tab-set}

bqbiol_is

extends [baseBqbiol](#)

See <http://co.mbine.org/standards/qualifiers>.

{tab-set}

bqbiol_isDescribedBy

extends baseBqbiol

See <http://co.mbine.org/standards/qualifiers>.

{tab-set}

bqbiol_isEncodedBy

extends baseBqbiol

See <http://co.mbine.org/standards/qualifiers>.

{tab-set}

bqbiol_isHomologTo

extends baseBqbiol

See <http://co.mbine.org/standards/qualifiers>.

{tab-set}

bqbiol_isPartOf

extends baseBqbiol

See <http://co.mbine.org/standards/qualifiers>.

{tab-set}

bqbiol_isPropertyOf

extends baseBqbiol

See <http://co.mbine.org/standards/qualifiers>.

{tab-set}

bqbiol_isVersionOf

extends baseBqbiol

See <http://co.mbine.org/standards/qualifiers>.

Text fields

<code>xmlns:bqbiol</code>

bqbiol_occursIn

extends `baseBqbiol`

See <http://co.mbine.org/standards/qualifiers>.

{tab-set}

bqbiol_hasTaxon

extends `baseBqbiol`

See <http://co.mbine.org/standards/qualifiers>.

{tab-set}

bqmodel_is

extends `baseBqbiol`

See <http://co.mbine.org/standards/qualifiers>.

{tab-set}

bqmodel_isDescribedBy

extends `baseBqbiol`

See <http://co.mbine.org/standards/qualifiers>.

Text fields

<code>xmlns:bqmodel</code>

bqmodel_isDerivedFrom

extends `baseBqbiol`

See <http://co.mbine.org/standards/qualifiers>.

{tab-set}

rdf_Bag

Structured block in an *annotation* based on RDF.

Children list

rdf:li		schema:rdf:li
---------------	--	---------------

rdf_li

Structured block in an *annotation* based on RDF.

Text fields

rdf:resource

point3DWithDiam

Base type for ComponentTypes which specify an (**x**, **y**, **z**) coordinate along with a **diameter**. Note: no dimension used in the attributes for these coordinates! These are assumed to have dimension micrometer (10^{-6} m). This is due to micrometers being the default option for the majority of neuronal morphology formats, and dimensions are omitted here to facilitate reading and writing of morphologies in NeuroML.

Parameters

diameter	Diameter of the ppoint. Note: no dimension used, see description of <i>point3DWithDiam</i> for details.	Dimensionless
x	x coordinate of the point. Note: no dimension used, see description of <i>point3DWithDiam</i> for details.	Dimensionless
y	y coordinate of the ppoint. Note: no dimension used, see description of <i>point3DWithDiam</i> for details.	Dimensionless
z	z coordinate of the ppoint. Note: no dimension used, see description of <i>point3DWithDiam</i> for details.	Dimensionless

Derived parameters

radius	A dimensional quantity given by half the _diameter.	<i>length</i>
xLength	A version of _x with dimension length.	<i>length</i>
yLength	A version of _y with dimension length.	<i>length</i>
zLength	A version of _z with dimension length.	<i>length</i>

Constants

MICRON = 1um	length
--------------	--------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Point3DWithDiam

variable = Point3DWithDiam(x=None, y=None, z=None, diameter=None, gds_collector_=None,
                           **kwargs_)
```

10.1.4 Cells

Defines both abstract cell models (e.g. [izhikevichCell](#), adaptive exponential integrate and fire cell, [adExIaFCell](#)), point conductance based cell models ([pointCellCondBased](#), [pointCellCondBasedCa](#)) and cells models ([cell](#)) which specify the [morphology](#) (containing [segments](#)) and [biophysicalProperties](#) separately.

Original ComponentType definitions: [Cells.xml](#). Schema against which NeuroML based on these should be valid: [NeuroML_v2.2.xsd](#). Generated on 07/06/22 from [this](#) commit. Please file any issues or questions at the issue tracker [here](#).

baseCell

extends [baseStandalone](#)

Base type of any cell (e.g. point neuron like [izhikevich2007Cell](#), or a morphologically detailed [cell](#) with [segments](#)) which can be used in a [population](#).

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BaseCell

variable = BaseCell(neuro_lex_id=None, id=None, metaid=None, notes=None,
                     properties=None, annotation=None, extensiontype_=None, gds_collector_=None,
                     **kwargs_)
```

baseSpikingCellextends [baseCell](#)Base type of any cell which can emit **spike** events.**Event Ports**

spike	Spike event	Direction: out
--------------	-------------	----------------

baseCellMembPotextends [baseSpikingCell](#)Any spiking cell which has a membrane potential **v** with units of voltage (as opposed to a dimensionless membrane potential used in [baseCellMembPotDL](#)).**Exposures**

v	Membrane potential	<i>voltage</i>
----------	--------------------	----------------

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
--------------	---	----------------

baseCellMembPotDLextends [baseSpikingCell](#)Any spiking cell which has a dimensioness membrane potential, **V** (as opposed to a membrane potential units of voltage, [baseCellMembPot](#)).**Exposures**

V	Membrane potential	Dimensionless
----------	--------------------	---------------

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
--------------	---	----------------

baseChannelPopulationextends [baseVoltageDepPointCurrent](#)Base type for any current produced by a population of channels, all of which are of type **ionChannel**.**Component References**

ion- Chan- nel		baseIonChannel
-------------------------------	--	--------------------------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	current
----------	--	-------------------------

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepPointCurrent</i>)	voltage
----------	---	-------------------------

channelPopulationextends [baseChannelPopulation](#)Population of a **number** of ohmic ion channels. These each produce a conductance **channelg** across a reversal potential **erev**, giving a total current **i**. Note that active membrane currents are more frequently specified as a density over an area of the [cell](#) using [channelDensity](#).**Parameters**

erev	The reversal potential of the current produced	voltage
number	The number of channels present. This will be multiplied by the time varying conductance of the individual ion channel (which extends baseIonChannel) to produce the total conductance	Dimensionless

Text fields

ion	Which ion flows through the channel. Note: ideally this needs to be a property of ionChannel only, but it's here as it makes it easier to select channelPopulations transmitting specific ions.
------------	---

Constants

vShift = 0mV	<i>voltage</i>
---------------------	----------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
----------	--	----------------

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepPointCurrent</i>)	<i>voltage</i>
----------	---	----------------

Dynamics

Structure CHILD INSTANCE: **ionChannel**

Derived Variables **channelg** = ionChannel->g

geff = channelg * number

i = geff * (erev - v) (exposed as **i**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelPopulation

variable = ChannelPopulation(neuro_lex_id=None, id=None, ion_channel=None, ↴
    ↴number=None, erev=None, segment_groups='all', segments=None, ion=None, variable_
    ↴parameters=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<channelPopulation id="naChansDend" ionChannel="NaConductance" segment="2" number=
→"120000" erev="50mV" ion="na"/>
```

channelPopulationNernstextends [baseChannelPopulation](#)

Population of a **number** of channels with a time varying reversal potential **erev** determined by Nernst equation. Note: hard coded for Ca only!

Parameters

number	The number of channels present. This will be multiplied by the time varying conductance of the individual ion channel (which extends <i>baseIonChannel</i>) to produce the total conductance	Dimensionless
---------------	---	---------------

Text fields

ion	Which ion flows through the channel. Note: ideally this needs to be a property of ionChannel only, but it's here as it makes it easier to select channelPopulations transmitting specific ions.
------------	---

Constants

R = 8.3144621 J_per_K_per_mol		<i>idealGasConstant-Dims</i>
zCa = 2		Dimensionless
F = 96485.3 C_per_mol		<i>charge_per_mole</i>
vShift = 0mV		<i>voltage</i>

Exposures

erev	The reversal potential of the current produced, calculated from _caConcExt and _caConc	<i>voltage</i>
i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>

Requirements

caConc	The internal Ca ²⁺ concentration, as calculated/exposed by the parent Component	<i>concentration</i>
caConcExt	The external Ca ²⁺ concentration, as calculated/exposed by the parent Component	<i>concentration</i>
temperature	The temperature to use in the calculation of _erev. Note this is generally exposed by a <i>networkWithTemperature</i> .	<i>temperature</i>
v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepPointCurrent</i>)	<i>voltage</i>

Dynamics

Structure CHILD INSTANCE: **ionChannel**

Derived Variables **singleChannelConductance** = ionChannel->g

totalConductance = singleChannelConductance * number

erev = (R * temperature / (zCa * F)) * log(caConcExt / caConc) (exposed as **erev**)

i = totalConductance * (erev - v) (exposed as **i**)

baseChannelDensity

Base type for a current of density **iDensity** distributed on an area of a *cell*, flowing through the specified **ionChannel**. Instances of this (normally *channelDensity*) are specified in the *membraneProperties* of the *cell*.

Component References

ion- Chan- nel		<i>baseIonChannel</i>
-------------------------------	--	-----------------------

Exposures

iDen- sity		<i>currentDensity</i>
-----------------------	--	-----------------------

Requirements

v		<i>voltage</i>
----------	--	----------------

baseChannelDensityCondextends [baseChannelDensity](#)

Base type for distributed conductances on an area of a cell producing a (not necessarily ohmic) current.

Parameters

cond-Density		<i>conductanceDensity</i>
---------------------	--	---------------------------

Exposures

gDen-sity		<i>conductanceDensity</i>
iDen-sity	(from baseChannelDensity)	<i>currentDensity</i>

Requirements

v	(from baseChannelDensity)	<i>voltage</i>
----------	--	----------------

variableParameter

Specifies a **parameter** (e.g. condDensity) which can vary its value across a **segmentGroup**. The value is calculated from **value** attribute of the *inhomogeneousValue* subelement. This element is normally a child of *channelDensityNonUniform*, *channelDensityNonUniformNernst* or *channelDensityNonUniformGHK* and is used to calculate the value of the conductance, etc. which will vary on different parts of the cell. The **segmentGroup** specified here needs to define an *inhomogeneousParameter* (referenced from **inhomogeneousParameter** in the *inhomogeneousValue*), which calculates a **variable** (e.g. p) varying across the cell (e.g. based on the path length from soma), which is then used in the **value** attribute of the *inhomogeneousValue* (so for example condDensity = f(p)).

Text fields

parameter	
segment-Group	

Child list

inhomogeneousValue		<i>inhomogeneousValue</i>
---------------------------	--	---------------------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import VariableParameter

variable = VariableParameter(parameter=None, segment_groups=None, inhomogeneous_
    ↴value=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<variableParameter parameter="condDensity" segmentGroup="dendrite_group">
    <inhomogeneousValue inhomogeneousParameter="dendrite_group_x1"
    ↴" value="5e-7 * exp(-p/200)"/>
</variableParameter>
```

inhomogeneousValue

Specifies the **value** of an **inhomogeneousParameter**. For usage see *variableParameter*.

Text fields

inhomogeneousParameter	
value	

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import InhomogeneousValue

variable = InhomogeneousValue(inhomogeneous_parameters=None, value=None, gds_
    ↴collector_=None, **kwargs_)
```

Usage: XML

```
<inhomogeneousValue inhomogeneousParameter="dendrite_group_x1" value="5e-7 * exp(-p/  
-200) "/>
```

channelDensityNonUniform

extends [baseChannelDensity](#)

Specifies a time varying ohmic conductance density, which is distributed on a region of the **cell**. The conductance density of the channel is not uniform, but is set using the [variableParameter](#). Note, there is no dynamical description of this in LEMS yet, as this type only makes sense for multicompartmental cells. A [ComponentType](#) for this needs to be present to enable export of NeuroML 2 multicompartmental cells via LEMS/jNeuroML to NEURON.

Parameters

erev	The reversal potential of the current produced	<i>voltage</i>
-------------	--	----------------

Text fields

segment-Group	
ion	Which ion flows through the channel. Note: ideally this needs to be a property of ionChannel only, but it's here as it makes it easier to select channelPopulations transmitting specific ions.

Child list

vari-ablePa-rameter		<i>variableParameter</i>
----------------------------	--	--------------------------

Constants

ZERO_CURR_DENS = 0 A_per_m2		<i>currentDensity</i>
---------------------------------------	--	-----------------------

Exposures

iDen-sity	(from baseChannelDensity)	currentDensity
-----------	---------------------------	----------------

Requirements

v	(from baseChannelDensity)	voltage
---	---------------------------	---------

Dynamics

Structure CHILD INSTANCE: **ionChannel**

Derived Variables **iDensity** = ZERO_CURR_DENS (exposed as **iDensity**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelDensityNonUniform

variable = ChannelDensityNonUniform(neuro_lex_id=None, id=None, ion_channel=None, ↴
                                     erev=None, ion=None, variable_parameters=None, gds_collector=None, **kwargs_)
```

Usage: XML

```
<channelDensityNonUniform id="nonuniform_na_chans" ionChannel="NaConductance" erev=
    ↴"50mV" ion="na">
    <variableParameter parameter="condDensity" segmentGroup="dendrite_
    ↴group">
        <inhomogeneousValue inhomogeneousParameter="dendrite_group_x1
    ↴" value="5e-7 * exp(-p/200)"/>
    </variableParameter>
</channelDensityNonUniform>
```

channelDensityNonUniformNernst

extends **baseChannelDensity**

Specifies a time varying conductance density, which is distributed on a region of the **cell**, and whose reversal potential is calculated from the Nernst equation. Hard coded for Ca only!. The conductance density of the channel is not uniform, but is set using the **variableParameter**. Note, there is no dynamical description of this in LEMS yet, as this type only makes sense for multicompartmental cells. A ComponentType for this needs to be present to enable export of NeuroML 2 multicompartmental cells via LEMS/jNeuroML to NEURON.

Text fields

segment-Group	
ion	Which ion flows through the channel. Note: ideally this needs to be a property of ionChannel only, but it's here as it makes it easier to select channelPopulations transmitting specific ions.

Child list

variableParameter		<i>variableParameter</i>
--------------------------	--	--------------------------

Constants

ZERO_CURR_DENS = 0		<i>currentDensity</i>
---------------------------	--	-----------------------

Exposures

iDensity	(from baseChannelDensity)	<i>currentDensity</i>
-----------------	---------------------------	-----------------------

Requirements

v	(from baseChannelDensity)	<i>voltage</i>
----------	---------------------------	----------------

Dynamics**Structure CHILD INSTANCE: ionChannel****Derived Variables** **iDensity** = ZERO_CURR_DENS (exposed as **iDensity**)**Usage: Python***Go to the libNeuroML documentation*

```
from neuroml import ChannelDensityNonUniformNernst

variable = ChannelDensityNonUniformNernst(neuro_lex_id=None, id=None, ion_
                                         _channel=None, ion=None, variable_parameters=None, gds_collector_=None, **kwargs_)
```

channelDensityNonUniformGHK

extends [baseChannelDensity](#)

Specifies a time varying conductance density, which is distributed on a region of the **cell**, and whose current is calculated from the Goldman-Hodgkin-Katz equation. Hard coded for Ca only!. The conductance density of the channel is not uniform, but is set using the [*variableParameter*](#). Note, there is no dynamical description of this in LEMS yet, as this type only makes sense for multicompartmental cells. A ComponentType for this needs to be present to enable export of NeuroML 2 multicompartmental cells via LEMS/jNeuroML to NEURON.

Text fields

segment-Group	
ion	Which ion flows through the channel. Note: ideally this needs to be a property of ionChannel only, but it's here as it makes it easier to select channelPopulations transmitting specific ions.

Child list

variableParameter		<i>variableParameter</i>
--------------------------	--	--------------------------

Constants

ZERO_CURR_DENS = 0 A_per_m2		<i>currentDensity</i>
---------------------------------------	--	-----------------------

Exposures

iDensity	(from baseChannelDensity)	<i>currentDensity</i>
-----------------	--	-----------------------

Requirements

v	(from baseChannelDensity)	<i>voltage</i>
----------	--	----------------

Dynamics

Structure CHILD INSTANCE: **ionChannel**

Derived Variables **iDensity** = ZERO_CURR_DENS (exposed as **iDensity**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelDensityNonUniformGHK

variable = ChannelDensityNonUniformGHK(neuro_lex_id=None, id=None, ion_channel=None, _  
    ↪ion=None, variable_parameters=None, gds_collector_=None, **kwargs_)
```

channelDensity

extends **baseChannelDensityCond**

Specifies a time varying ohmic conductance density, **gDensity**, which is distributed on an area of the **cell** (specified in *membraneProperties*) with fixed reversal potential **erev** producing a current density **iDensity**.

Parameters

cond-Density	(from baseChannelDensityCond)	<i>conductanceDensity</i>
erev	The reversal potential of the current produced	<i>voltage</i>

Text fields

segment-Group	Which <i>segmentGroup</i> the channelDensity is placed on. If this is missing, it implies it is placed on all <i>_segment_s</i> of the <i>cell</i>
ion	Which ion flows through the channel. Note: ideally this needs to be a property of ionChannel only, but it's here as it makes it easier to select channelPopulations transmitting specific ions.

Constants

vShift = 0mV		<i>voltage</i>
---------------------	--	----------------

Exposures

gDensity	(from baseChannelDensityCond)	<i>conductanceDensity</i>
iDensity	(from baseChannelDensity)	<i>currentDensity</i>

Requirements

v	(from baseChannelDensity)	<i>voltage</i>
----------	---------------------------	----------------

Dynamics

Structure CHILD INSTANCE: **ionChannel**

Derived Variables **channelf** = ionChannel->fopen

gDensity = condDensity * channelf (exposed as **gDensity**)

iDensity = gDensity * (erev - v) (exposed as **iDensity**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelDensity

variable = 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, extenstiontype_=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<channelDensity id="pasChans" ionChannel="pas" condDensity="3.0 S_per_m2" erev="-70mV"
" ion="non_specific"/>
```

```
<channelDensity id="naChansSoma" ionChannel="NaConductance" segmentGroup="soma_group"_
condDensity="120.0 mS_per_cm2" erev="50mV" ion="na"/>
```

```
<channelDensity id="naChans" ionChannel="HH_Na" segmentGroup="soma_group" condDensity=
"120.0 mS_per_cm2" ion="na" erev="50mV"/>
```

channelDensityVShift

extends *channelDensity*

Same as *channelDensity*, 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.

Parameters

cond-Density	(from <i>baseChannelDensityCond</i>)	<i>conductanceDensity</i>
erev	The reversal potential of the current produced (from <i>channelDensity</i>)	<i>voltage</i>
vShift		<i>voltage</i>

Text fields

segment-Group	Which <i>segmentGroup</i> the channelDensity is placed on. If this is missing, it implies it is placed on all <i>_segment_s</i> of the <i>cell</i>
ion	Which ion flows through the channel. Note: ideally this needs to be a property of <i>ionChannel</i> only, but it's here as it makes it easier to select <i>channelPopulations</i> transmitting specific ions.

Exposures

gDen-sity	(from <i>baseChannelDensityCond</i>)	<i>conductanceDensity</i>
iDen-sity	(from <i>baseChannelDensity</i>)	<i>currentDensity</i>

Requirements

v	(from <i>baseChannelDensity</i>)	<i>voltage</i>
----------	-----------------------------------	----------------

Usage: Python

Go to the *libNeuroML* documentation

```
from neuroml import ChannelDensityVShift

variable = 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, gds_collector_=None, **kwargs_)
```

channelDensityNernst

extends `baseChannelDensityCond`

Specifies a time varying conductance density, **gDensity**, which is distributed on an area of the **cell**, producing a current density **iDensity** and whose reversal potential is calculated from the Nernst equation. Hard coded for Ca only! See <https://github.com/OpenSourceBrain/ghk-nernst>.

Parameters

cond-Density	(from <code>baseChannelDensityCond</code>)	<i>conductanceDensity</i>
---------------------	---	---------------------------

Text fields

segment-Group	Which <i>segmentGroup</i> the channelDensityNernst is placed on. If this is missing, it implies it is placed on all _segment_s of the <i>cell</i>
ion	Which ion flows through the channel. Note: ideally this needs to be a property of ionChannel only, but it's here as it makes it easier to select channelPopulations transmitting specific ions.

Constants

R = 8.3144621 J_per_K_per_mol		<i>idealGasConstant-Dims</i>
zCa = 2		Dimensionless
F = 96485.3 C_per_mol		<i>charge_per_mole</i>

Exposures

erev	The reversal potential of the current produced, calculated from caConcExt and caConc	<i>voltage</i>
gDensity	(from <code>baseChannelDensityCond</code>)	<i>conductanceDensity</i>
iDensity	(from <code>baseChannelDensity</code>)	<i>currentDensity</i>

Requirements

caConc		<i>concentration</i>
caConcExt		<i>concentration</i>
temperature		<i>temperature</i>
v	(from <code>baseChannelDensity</code>)	<i>voltage</i>

Dynamics

Structure CHILD INSTANCE: **ionChannel**

Derived Variables **channelf** = ionChannel->fopen

Conditional Derived Variables IF caConcExt > 0 THEN

gDensity = condDensity * channelf (exposed as **gDensity**)

IF caConcExt <= 0 THEN

gDensity = 0 (exposed as **gDensity**)

IF caConcExt > 0 THEN

erev = (R * temperature / (zCa * F)) * log(caConcExt / caConc) (exposed as **erev**)

IF caConcExt <= 0 THEN

erev = 0 (exposed as **erev**)

IF caConcExt > 0 THEN

iDensity = gDensity * (erev - v) (exposed as **iDensity**)

IF caConcExt <= 0 THEN

iDensity = 0 (exposed as **iDensity**)

Usage: Python

Go to the *libNeuroML* documentation

```
from neuroml import ChannelDensityNernst

variable = 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, gds_collector_=None, **kwargs_)
```

channelDensityNernstCa2

extends [baseChannelDensityCond](#)

This component is similar to the original component type *channelDensityNernst* but it is changed in order to have a reversal potential that depends on a second independent Ca++ pool (ca2). See <https://github.com/OpenSourceBrain/ghk-nernst>.

Parameters

cond-Density	(from baseChannelDensityCond)	<i>conductanceDensity</i>
---------------------	--	---------------------------

Text fields

segment-Group	Which <i>segmentGroup</i> the channelDensityNernstCa2 is placed on. If this is missing, it implies it is placed on all <i>_segment_s</i> of the <i>cell</i>
ion	Which ion flows through the channel. Note: ideally this needs to be a property of ionChannel only, but it's here as it makes it easier to select channelPopulations transmitting specific ions.

Constants

R = 8.3144621 J_per_K_per_mol		<i>idealGasConstant-Dims</i>
zCa = 2		Dimensionless
F = 96485.3 C_per_mol		<i>charge_per_mole</i>

Exposures

erev	The reversal potential of the current produced	<i>voltage</i>
gDen-sity	(from baseChannelDensityCond)	<i>conductanceDensity</i>
iDen-sity	(from baseChannelDensity)	<i>currentDensity</i>

Requirements

ca-Conc2		<i>concentration</i>
caConcExt2		<i>concentration</i>
temperature		<i>temperature</i>
v	(from baseChannelDensity)	<i>voltage</i>

Dynamics

Structure CHILD INSTANCE: **ionChannel**

Derived Variables **channelf** = ionChannel->fopen

Conditional Derived Variables IF caConcExt2 > 0 THEN

gDensity = condDensity * channelf (exposed as **gDensity**)

IF caConcExt2 <= 0 THEN

gDensity = 0 (exposed as **gDensity**)

IF caConcExt2 > 0 THEN

erev = (R * temperature / (zCa * F)) * log(caConcExt2 / caConc2) (exposed as **erev**)

```
IF caConcExt2 <= 0 THEN
    erev = 0    (exposed as erev)
IF caConcExt2 > 0 THEN
    iDensity = gDensity * (erev - v)    (exposed as iDensity)
IF caConcExt2 <= 0 THEN
    iDensity = 0    (exposed as iDensity)
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelDensityNernstCa2

variable = ChannelDensityNernstCa2(neuro_lex_id=None, id=None, ion_channel=None, cond_
    ↪density=None, segment_groups='all', segments=None, ion=None, variable_
    ↪parameters=None, gds_collector=None, **kwargs_)
```

channelDensityGHK

extends `baseChannelDensity`

Specifies a time varying conductance density, **gDensity**, which is distributed on an area of the cell, producing a current density **iDensity** and whose reversal potential is calculated from the Goldman Hodgkin Katz equation. Hard coded for Ca only! See <https://github.com/OpenSourceBrain/ghk-nernst>.

Parameters

permeability		<i>permeability</i>
---------------------	--	---------------------

Text fields

segment-Group	Which <i>segmentGroup</i> the channelDensityGHK is placed on. If this is missing, it implies it is placed on all _segment_s of the <i>cell</i>
ion	Which ion flows through the channel. Note: ideally this needs to be a property of ionChannel only, but it's here as it makes it easier to select channelPopulations transmitting specific ions.

Constants

R = 8.3144621 J_per_K_per_mol		<i>idealGasConstant-Dims</i>
zCa = 2		Dimensionless
F = 96485.3 C_per_mol		<i>charge_per_mole</i>

Exposures

iDensity	(from baseChannelDensity)	<i>currentDensity</i>
-----------------	---------------------------	-----------------------

Requirements

caConc		<i>concentration</i>
caConcExt		<i>concentration</i>
temperature		<i>temperature</i>
v	(from baseChannelDensity)	<i>voltage</i>

Dynamics

Structure CHILD INSTANCE: **ionChannel**

Derived Variables **K** = (**zCa** * **F**) / (**R** * **temperature**)

expKv = exp(-1 * **K** * **v**)

channelf = ionChannel->fopen

Conditional Derived Variables IF **caConcExt** > 0 THEN

iDensity = -1 * **channelf** * permeability * **zCa** * **F** * **K** * **v** * (**caConc** - (**caConcExt** * **expKv**)) / (1 - **expKv**)
(exposed as **iDensity**)

IF **caConcExt** <= 0 THEN

iDensity = 0 (exposed as **iDensity**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelDensityGHK

variable = ChannelDensityGHK(neuro_lex_id=None, id=None, ion_channel=None, _permeability=None, segment_groups='all', segments=None, ion=None, gds_collector_=None, **kwargs_)
```

channelDensityGHK2

extends `baseChannelDensityCond`

Time varying conductance density, **gDensity**, which is distributed on an area of the cell, producing a current density **iDensity**. Modified version of Jaffe et al. 1994 (used also in Lawrence et al. 2006). See <https://github.com/OpenSourceBrain/ghk-nernst>.

Parameters

cond-Density	(from <code>baseChannelDensityCond</code>)	<i>conductanceDensity</i>
---------------------	---	---------------------------

Text fields

segment-Group	Which <i>segmentGroup</i> the channelDensityGHK2 is placed on. If this is missing, it implies it is placed on all _segment_s of the <i>cell</i>
ion	Which ion flows through the channel. Note: ideally this needs to be a property of ionChannel only, but it's here as it makes it easier to select channelPopulations transmitting specific ions.

Constants

VOLT_SCALE = 1 mV		<i>voltage</i>
CONC_SCALE = 1 mM		<i>concentration</i>
TEMP_SCALE = 1 K		<i>temperature</i>

Exposures

gDensity	(from <code>baseChannelDensityCond</code>)	<i>conductanceDensity</i>
iDensity	(from <code>baseChannelDensity</code>)	<i>currentDensity</i>

Requirements

caConc		<i>concentration</i>
caConcExt		<i>concentration</i>
temperature		<i>temperature</i>
v	(from <code>baseChannelDensity</code>)	<i>voltage</i>

Dynamics

Structure CHILD INSTANCE: **ionChannel**

Derived Variables $V = v / \text{VOLT_SCALE}$

$\text{ca_conc_i} = \text{caConc} / \text{CONC_SCALE}$

$\text{ca_conc_ext} = \text{caConcExt} / \text{CONC_SCALE}$

$T = \text{temperature} / \text{TEMP_SCALE}$

$\text{channelf} = \text{ionChannel}->\text{fopen}$

$\text{gDensity} = \text{condDensity} * \text{channelf}$ (exposed as **gDensity**)

$\text{tmp} = (25 * T) / (293.15 * 2)$

Conditional Derived Variables IF $V/\text{tmp} = 0$. THEN

$\text{pOpen} = \text{tmp} * 1e-3 * (1 - ((\text{ca_conc_i}/\text{ca_conc_ext}) * \exp(V/\text{tmp}))) * (1 - (V/\text{tmp})/2)$

IF $V/\text{tmp} \neq 0$. THEN

$\text{pOpen} = \text{tmp} * 1e-3 * (1 - ((\text{ca_conc_i}/\text{ca_conc_ext}) * \exp(V/\text{tmp}))) * ((V/\text{tmp}) / (\exp(V/\text{tmp}) - 1))$

IF $\text{ca_conc_ext} > 0$ THEN

$\text{iDensity} = \text{gDensity} * \text{pOpen}$ (exposed as **iDensity**)

IF $\text{ca_conc_ext} \leq 0$ THEN

$\text{iDensity} = 0$ (exposed as **iDensity**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelDensityGHK2

variable = ChannelDensityGHK2(neuro_lex_id=None, id=None, ion_channel=None, cond_
                               density=None, segment_groups='all', segments=None, ion=None, gds_collector_=None,_
                               **kwargs)
```

pointCellCondBased

extends [baseCellMembPotCap](#)

Simple model of a conductance based cell, with no separate *morphology* element, just an absolute capacitance **C**, and a set of channel **populations**. Note: use of *cell* is generally preferable (and more widely supported), even for a single compartment cell.

Parameters

C	Total capacitance of the cell membrane (<i>from baseCellMembPotCap</i>)	<i>capacitance</i>
thresh	The voltage threshold above which the cell is considered to be _spiking	<i>voltage</i>
v0	The initial membrane potential of the cell	<i>voltage</i>

Children list

popula- tions		<i>baseChannelPopula-</i> <i>tion</i>
--------------------------------	--	--

Exposures

iMemb	Total current crossing the cell membrane (<i>from baseCellMembPotCap</i>)	<i>current</i>
iSyn	Total current due to synaptic inputs (<i>from baseCellMembPotCap</i>)	<i>current</i>
v	Membrane potential (<i>from baseCellMembPot</i>)	<i>voltage</i>

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
--------------	---	----------------

Attachments

synapses	<i>basePointCurrent</i>
-----------------	-------------------------

Dynamics

State Variables **v**: *voltage* (exposed as **v**)

spiking: Dimensionless

On Start **v** = **v0**

spiking = 0

On Conditions IF **v** > **thresh** AND **spiking** < 0.5 THEN

spiking = 1

EVENT OUT on port: **spike**

IF **v** < **thresh** THEN

spiking = 0

Derived Variables **iChannels** = **populations**[*]->i(reduce method: add)

iSyn = **synapses**[*]->i(reduce method: add) (exposed as **iSyn**)

iMemb = **iChannels** + **iSyn** (exposed as **iMemb**)

Time Derivatives $d v / dt = iMemb / C$

pointCellCondBasedCa

extends [baseCellMembPotCap](#)

TEMPORARY: Point cell with conductances and Ca concentration info. Not yet fully tested!!! TODO: Remove in favour of *cell*.

Parameters

C	Total capacitance of the cell membrane (<i>from baseCellMembPotCap</i>)	<i>capacitance</i>
thresh	The voltage threshold above which the cell is considered to be _spiking	<i>voltage</i>
v0	The initial membrane potential of the cell	<i>voltage</i>

Children list

popula-tions		<i>baseChannelPopula-tion</i>
concen-tration-Models		<i>concentrationModel</i>

Exposures

caConc		<i>concentration</i>
iCa		<i>current</i>
iMemb	Total current crossing the cell membrane (<i>from baseCellMembPotCap</i>)	<i>current</i>
iSyn	Total current due to synaptic inputs (<i>from baseCellMembPotCap</i>)	<i>current</i>
v	Membrane potential (<i>from baseCellMembPot</i>)	<i>voltage</i>

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
--------------	---	----------------

Attachments

synapses	<i>basePointCurrent</i>
-----------------	-------------------------

Dynamics

State Variables **v**: *voltage* (exposed as **v**)

spiking: Dimensionless

On Start **v** = **v0**

spiking = 0

On Conditions IF **v** > thresh AND **spiking** < 0.5 THEN

spiking = 1

EVENT OUT on port: **spike**

IF **v** < thresh THEN

spiking = 0

Derived Variables **iChannels** = populations[*]->i(reduce method: add)

iCa = populations[ion='ca']->i(reduce method: add) (exposed as **iCa**)

caConc = concentrationModels[species='ca']->concentration(reduce method: add) (exposed as **caConc**)

iSyn = synapses[*]->i(reduce method: add) (exposed as **iSyn**)

iMemb = **iChannels** + **iSyn** (exposed as **iMemb**)

Time Derivatives $d v / dt$ = **iMemb** / C

distal

extends *point3DWithDiam*

Point on a *segment* furthest from the soma. Should always be present in the description of a *segment*, unlike *proximal*.

Parameters

diameter	Diameter of the ppoint. Note: no dimension used, see description of <i>point3DWithDiam</i> for details. (<i>from point3DWithDiam</i>)	Dimensionless
x	x coordinate of the point. Note: no dimension used, see description of <i>point3DWithDiam</i> for details. (<i>from point3DWithDiam</i>)	Dimensionless
y	y coordinate of the ppoint. Note: no dimension used, see description of <i>point3DWithDiam</i> for details. (<i>from point3DWithDiam</i>)	Dimensionless
z	z coordinate of the ppoint. Note: no dimension used, see description of <i>point3DWithDiam</i> for details. (<i>from point3DWithDiam</i>)	Dimensionless

Derived parameters

radius	A dimensional quantity given by half the _diameter. (<i>from point3DWithDiam</i>)	<i>length</i>
xLength	A version of _x with dimension length. (<i>from point3DWithDiam</i>)	<i>length</i>
yLength	A version of _y with dimension length. (<i>from point3DWithDiam</i>)	<i>length</i>
zLength	A version of _z with dimension length. (<i>from point3DWithDiam</i>)	<i>length</i>

Usage: XML

```
<distal x="10" y="0" z="0" diameter="10"/>
```

```
<distal x="20" y="0" z="0" diameter="3"/>
```

```
<distal x="30" y="0" z="0" diameter="1"/>
```

proximal

extends *point3DWithDiam*

Point on a *segment* closest to the soma. Note, the proximal point can be omitted, and in this case is defined as being the point **fractionAlong** between the proximal and *distal* point of the *parent*, i.e. if **fractionAlong** = 1 (as it is by default) it will be the *distal* on the parent, or if **fractionAlong** = 0, it will be the proximal point. If between 0 and 1, it is the linear interpolation between the two points.

Parameters

diameter	Diameter of the ppoint. Note: no dimension used, see description of <i>point3DWithDiam</i> for details. (<i>from point3DWithDiam</i>)	Dimensionless
x	x coordinate of the point. Note: no dimension used, see description of <i>point3DWithDiam</i> for details. (<i>from point3DWithDiam</i>)	Dimensionless
y	y coordinate of the ppoint. Note: no dimension used, see description of <i>point3DWithDiam</i> for details. (<i>from point3DWithDiam</i>)	Dimensionless
z	z coordinate of the ppoint. Note: no dimension used, see description of <i>point3DWithDiam</i> for details. (<i>from point3DWithDiam</i>)	Dimensionless

Derived parameters

radius	A dimensional quantity given by half the _diameter. (<i>from point3DWithDiam</i>)	<i>length</i>
xLength	A version of _x with dimension length. (<i>from point3DWithDiam</i>)	<i>length</i>
yLength	A version of _y with dimension length. (<i>from point3DWithDiam</i>)	<i>length</i>
zLength	A version of _z with dimension length. (<i>from point3DWithDiam</i>)	<i>length</i>

Usage: XML

```
<proximal x="0" y="0" z="0" diameter="10"/>
```

```
<proximal x="25" y="0" z="0" diameter="0.1"/>
```

```
<proximal x="0" y="0" z="0" diameter="10"/>
```

parent

Specifies the *segment* which is this segment's parent. The **fractionAlong** specifies where it is connected, usually 1 (the default value), meaning the *distal* point of the parent, or 0, meaning the *proximal* point. If it is between these, a linear interpolation between the 2 points should be used.

Text fields

segment	The id of the parent segment
fractionA-long	The fraction along the the parent segment at which this segment is attached. For usage see <i>proximal</i>

Usage: XML

```
<parent segment="0"/>
```

```
<parent segment="1"/>
```

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

segment

A segment defines the smallest unit within a possibly branching structure (*morphology*), such as a dendrite or axon. Its **id** should be a nonnegative integer (usually soma/root = 0). Its end points are given by the *proximal* and *distal* points. The *proximal* point can be omitted, usually because it is the same as a point on the *parent* segment, see *proximal* for details. *parent* specifies the parent segment. The first segment of a *cell* (with no *parent*) usually represents the soma. The shape is normally a cylinder (radii of the *proximal* and *distal* equal, but positions different) or a conical frustum (radii and positions different). If the x, y, z positions of the *proximal* and *distal* are equal, the segment can be interpreted as a sphere, and in this case the radii of these points must be equal. NOTE: LEMS does not yet support multicompartmental modelling, so the Dynamics here is only appropriate for single compartment modelling.

Text fields

name	An optional name for the segment. Convenient for providing a suitable variable name for generated code, e.g. soma, dend0
-------------	--

Child list

parent		<i>parent</i>
distal		<i>distal</i>
proximal		<i>proximal</i>

Constants

LEN = 1m		<i>length</i>
-----------------	--	---------------

Exposures

length		<i>length</i>
radDist		<i>length</i>
surfaceArea		<i>area</i>

Dynamics

Derived Variables **radDist** = distal->radius (exposed as **radDist**)

dx = distal->xLength

dy = distal->yLength

dz = distal->zLength

px = proximal->xLength

py = proximal->yLength

pz = proximal->zLength

length = $\sqrt{((dx - px) * (dx - px) + (dy - py) * (dy - py) + (dz - pz) * (dz - pz)) / (LEN * LEN)} * LEN$ (exposed as **length**)

Conditional Derived Variables IF length = 0 * LEN THEN

surfaceArea = $4 * radDist * radDist * 3.14159265$ (exposed as **surfaceArea**)

IF length > 0 * LEN THEN

surfaceArea = $2 * radDist * 3.14159265 * length$ (exposed as **surfaceArea**)

Usage: Python

Go to the [libNeuroML documentation](#)

```
from neuroml import Segment

variable = Segment(neuro_lex_id=None, id=None, name=None, parent=None, proximal=None, ↴
    ↴distal=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<segment id="3" name="Spine1">
    <parent segment="2" fractionAlong="0.5"/>
    <proximal x="25" y="0" z="0" diameter="0.1"/>
    <distal x="25" y="0.2" z="0" diameter="0.1"/>
</segment>
```

```
<segment id="0" name="Soma">
    <proximal x="0" y="0" z="0" diameter="10"/>
    <distal x="10" y="0" z="0" diameter="10"/>
</segment>
```

```
<segment id="1" name="Dendrite1">
    <parent segment="0"/>
    <distal x="20" y="0" z="0" diameter="3"/>
</segment>
```

segmentGroup

A method to describe a group of *segments* in a *morphology*, e.g. `soma_group`, `dendrite_group`, `axon_group`. While a name is useful to describe the group, the **neuroLexId** attribute can be used to explicitly specify the meaning of the group, e.g. `sao1044911821` for ‘Neuronal Cell Body’, `sao1211023249` for ‘Dendrite’. The *segments* in this group can be specified as: a list of individual *member* segments; a *path*, all of the segments along which should be included; a *subTree* of the *cell* to include; other *segmentGroups* to *include* (so all segments from those get included here). An *inhomogeneousParameter* can be defined on the region of the cell specified by this group (see *variableParameter* for usage).

Text fields

neu-roLexId	An id string for pointing to an entry in the NeuroLex ontology. Use of this attribute is a shorthand for a full RDF based reference to the MIRIAM Resource urn:miriam:neurolex, with an bqbiol:is qualifier.
--------------------	--

Child list

notes		<i>notes</i>
annotation		<i>annotation</i>

Children list

prop- erty		<i>property</i>
mem- bers		<i>member</i>
paths		<i>path</i>
sub- Trees		<i>subTree</i>
in- cludes		<i>include</i>
inhomoge- neous- Param- eter		<i>inhomogeneousParameter</i>

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SegmentGroup

variable = 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, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<segmentGroup id="dendrite_group" neuroLexId="sao1211023249">
    <member segment="1"/>
    <member segment="2"/>
    <member segment="3"/>
</segmentGroup>
```

```
<segmentGroup id="soma_group" neuroLexId="sao1044911821">
    <member segment="0"/>
</segmentGroup>
```

```
<segmentGroup id="spines" neuroLexId="sao1145756102">
    <member segment="3"/>
</segmentGroup>
```

member

A single identified **segment** which is part of the *segmentGroup*.

Text fields

segment	
---------	--

Usage: Python

Go to the *libNeuroML* documentation

```
from neuroml import Member

variable = Member(segments=None, gds_collector_=None, **kwargs_)
```

Usage: XML

<member segment="0"/>

<member segment="1"/>

<member segment="2"/>

from

In a *path* or *subTree*, specifies which **segment** (inclusive) from which to calculate the *segmentGroup*.

Text fields

segment	
---------	--

Usage: XML

<from segment="1"/>

<from segment="1"/>

to

In a *path*, specifies which **segment** (inclusive) up to which to calculate the *segmentGroup*.

Text fields

segment	
----------------	--

Usage: XML

<code><to segment="2" /></code>

include

Include all members of another *segmentGroup* in this group.

Text fields

href	TODO: fix this!!! This is needed here, since include is used to include external nml files!!
segment-Group	

Usage: Python

Go to the libNeuroML documentation

<pre>from neuroml import Include variable = Include(segment_groups=None, gds_collector_=None, **kwargs_)</pre>

Usage: XML

<code><include href="NML2_SingleCompHHCell.nml" /></code>

<code><include href="NML2_SimpleIonChannel.nml" /></code>

<code><include href="NML2_SimpleIonChannel.nml" /></code>

path

Include all the *segments* between those specified by *from* and *to*, inclusive.

Child list

from		<i>from</i>
to		<i>to</i>

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Path

variable = Path(from_=None, to=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<path>
    <from segment="1"/>
    <to segment="2"/>
</path>
```

subTree

Include all the *segments* distal to that specified by *from* in the *segmentGroup*.

Child list

from		<i>from</i>
-------------	--	-------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SubTree

variable = SubTree(from_=None, to=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<subTree>
    <from segment="1"/>
</subTree>
```

inhomogeneousParameter

An inhomogeneous parameter specified across the *segmentGroup* (see *variableParameter* for usage).

Text fields

variable	
metric	

Child list

proximal		<i>proximalProperties</i>
distal		<i>distalProperties</i>

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import InhomogeneousParameter

variable = InhomogeneousParameter(neuro_lex_id=None, id=None, variable=None,_
                                   metric=None, proximal=None, distal=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<inhomogeneousParameter id="dendrite_group_x2" variable="r" metric="Path Length from_
    ↪root">
    <proximal translationStart="0"/>
    <distal normalizationEnd="1"/>
</inhomogeneousParameter>
```

```
<inhomogeneousParameter id="dendrite_group_x1" variable="p" metric="Path Length from_
    ↪root"/>
```

proximalProperties

What to do at the proximal point when creating an inhomogeneous parameter.

Text fields

transla- tionStart	
-----------------------	--

distalProperties

What to do at the distal point when creating an inhomogeneous parameter.

Text fields

normal- izatio- nEnd	
----------------------------	--

morphology

The collection of *segments* which specify the 3D structure of the cell, along with a number of *segmentGroups*.

Children list

seg- ments		<i>segment</i>
seg- ment- Groups		<i>segmentGroup</i>

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Morphology

variable = Morphology(neuro_lex_id=None, id=None, metaid=None, notes=None,
                     properties=None, annotation=None, segments=None, segment_groups=None, gds_collector_=
                     None, **kwargs_)
```

Usage: XML

```

<morphology id="SpikingCell_morphology">
    <segment id="0" name="Soma">

        <proximal x="0" y="0" z="0" diameter="10"/>
        <distal x="10" y="0" z="0" diameter="10"/>
    </segment>
    <segment id="1" name="Dendrite1">
        <parent segment="0"/>

        <distal x="20" y="0" z="0" diameter="3"/>
    </segment>
    <segment id="2" name="Dendrite2">
        <parent segment="1"/>
        <distal x="30" y="0" z="0" diameter="1"/>
    </segment>
    <segment id="3" name="Spine1">
        <parent segment="2" fractionAlong="0.5"/>
        <proximal x="25" y="0" z="0" diameter="0.1"/>
        <distal x="25" y="0.2" z="0" diameter="0.1"/>
    </segment>

    <segmentGroup id="soma_group" neuroLexId="sao1044911821">
        <member segment="0"/>
    </segmentGroup>
    <segmentGroup id="dendrite_group" neuroLexId="sao1211023249">
        <member segment="1"/>
        <member segment="2"/>
        <member segment="3"/>
    </segmentGroup>
    <segmentGroup id="spines" neuroLexId="sao1145756102">
        <member segment="3"/>
    </segmentGroup>
</morphology>
```

```

<morphology id="NeuroMorpho_PyrCell123">
    <segment id="0" name="Soma">

        <proximal x="0" y="0" z="0" diameter="10"/>
        <distal x="10" y="0" z="0" diameter="10"/>
    </segment>

</morphology>
```

```

<morphology id="SimpleCell_Morphology">

    <segment id="0" name="Soma">

        <proximal x="0" y="0" z="0" diameter="10"/>
        <distal x="10" y="0" z="0" diameter="10"/>
    </segment>
    <segment id="1" name="MainDendrite1">
        <parent segment="0"/>

        <proximal x="10" y="0" z="0" diameter="3"/>
    </segment>
```

(continues on next page)

(continued from previous page)

```
<distal x="20" y="0" z="0" diameter="3"/>
</segment>
<segment id="2" name="MainDendrite2">
    <parent segment="1"/>

    <distal x="30" y="0" z="0" diameter="1"/>
</segment>

<segmentGroup id="soma_group" neuroLexId="sao1044911821">
    <member segment="0"/>
</segmentGroup>
<segmentGroup id="dendrite_group" neuroLexId="sao1211023249">
    <member segment="1"/>
    <member segment="2"/>

    <inhomogeneousParameter id="dendrite_group_x1" variable="p" metric=
    "Path Length from root"/>

    <inhomogeneousParameter id="dendrite_group_x2" variable="r" metric=
    "Path Length from root">
        <proximal translationStart="0"/>
        <distal normalizationEnd="1"/>
    </inhomogeneousParameter>

    </segmentGroup>
</morphology>
```

specificCapacitance

Capacitance per unit area.

Parameters

value	specificCapacitance
-------	---------------------

Text fields

segment-Group	
----------------------	--

Exposures

spec-Cap		<i>specificCapacitance</i>
-----------------	--	----------------------------

Dynamics

Derived Variables **specCap** = value (exposed as **specCap**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SpecificCapacitance

variable = SpecificCapacitance(value=None, segment_groups='all', gds_collector_=None, _  
    **kwargs_)
```

Usage: XML

```
<specificCapacitance segmentGroup="soma_group" value="1.0 uF_per_cm2"/>
```

```
<specificCapacitance segmentGroup="dendrite_group" value="2.0 uF_per_cm2"/>
```

```
<specificCapacitance segmentGroup="soma_group" value="1.0 uF_per_cm2"/>
```

initMembPotential

Explicitly set initial membrane potential for the cell.

Parameters

value		<i>voltage</i>
--------------	--	----------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import InitMembPotential

variable = InitMembPotential(value=None, segment_groups='all', gds_collector=None,_
                             **kwargs)
```

Usage: XML

```
<initMembPotential value="-65mV"/>
```

```
<initMembPotential value="-65mV"/>
```

spikeThresh

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.

Parameters

value	<i>voltage</i>
--------------	----------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SpikeThresh

variable = SpikeThresh(value=None, segment_groups='all', gds_collector=None,_
                           **kwargs)
```

Usage: XML

```
<spikeThresh value="-20mV"/>
```

```
<spikeThresh value="-20mV"/>
```

membraneProperties

Properties specific to the membrane, such as the **populations** of channels, **channelDensities**, **specificCapacitance**, etc.

Child list

init-MembPotential		<i>initMembPotential</i>
spikeThresh		<i>spikeThresh</i>

Children list

specificCapacitances		<i>specificCapacitance</i>
populations		<i>baseChannelPopulation</i>
channelDensities		<i>baseChannelDensity</i>

Exposures

iCa		<i>current</i>
totChan-Current		<i>current</i>
tot-Spec-Cap		<i>specificCapacitance</i>

Requirements

surfaceArea		<i>area</i>
--------------------	--	-------------

Dynamics

Derived Variables

- totSpecCap** = specificCapacitances[*]->specCap(reduce method: add) (exposed as **totSpecCap**)
- totChanPopCurrent** = populations[*]->i(reduce method: add)
- totChanDensCurrentDensity** = channelDensities[*]->iDensity(reduce method: add)
- totChanCurrent** = totChanPopCurrent + (totChanDensCurrentDensity * surfaceArea) (exposed as **totChanCurrent**)
- totChanPopCurrentCa** = populations[ion='ca']->i(reduce method: add)
- totChanDensCurrentDensityCa** = channelDensities[ion='ca']->iDensity(reduce method: add)
- iCa** = totChanPopCurrentCa + (totChanDensCurrentDensityCa * surfaceArea) (exposed as **iCa**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import MembraneProperties

variable = 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_
    ↪thresholds=None, specific_capacitances=None, init_memb_potentials=None, extensiontype_
    ↪=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<membraneProperties>
    <channelPopulation id="naChansDend" ionChannel="NaConductance"_
    ↪segment="2" number="120000" erev="50mV" ion="na"/>
        <channelDensity id="pasChans" ionChannel="pas" condDensity="3.0 S_per_
        ↪m2" erev="-70mV" ion="non_specific"/>
            <channelDensity id="naChansSoma" ionChannel="NaConductance"_
            ↪segmentGroup="soma_group" condDensity="120.0 mS_per_cm2" erev="50mV" ion="na"/>
                <specificCapacitance segmentGroup="soma_group" value="1.0 uF_per_cm2"/_
                ↪>
                <specificCapacitance segmentGroup="dendrite_group" value="2.0 uF_per_
                ↪cm2"/>
</membraneProperties>
```

```
<membraneProperties>

    <channelDensity id="naChans" ionChannel="HH_Na" segmentGroup="soma_group"_
    ↪condDensity="120.0 mS_per_cm2" ion="na" erev="50mV"/>
        <!-- Ions present inside the cell. Note: a fixed reversal potential is_
        ↪specified here
            <reversalPotential species="na" value="50mV"/>
            <reversalPotential species="k" value="-77mV"/>-->
</membraneProperties>
```

```

<membraneProperties>
    <channelDensityNonUniform id="nonuniform_na_chans" ionChannel=
        ↵ "NaConductance" erev="50mV" ion="na">
        <variableParameter parameter="condDensity" segmentGroup="dendrite_
        ↵ group">
            <inhomogeneousValue inhomogeneousParameter="dendrite_group_x1
        ↵ " value="5e-7 * exp(-p/200)"/>
        </variableParameter>
    </channelDensityNonUniform>
    <specificCapacitance segmentGroup="soma_group" value="1.0 uF_per_cm2"/
    ↵
</membraneProperties>

```

membraneProperties2CaPools

extends *membraneProperties*

Variant of membraneProperties with 2 independent Ca pools.

Child list

init-Mem-bPotential		<i>initMembPotential</i>
spikeThresh		<i>spikeThresh</i>

Children list

specific-Capacitances		<i>specificCapacitance</i>
populations		<i>baseChannelPopulation</i>
channelDensities		<i>baseChannelDensity</i>

Exposures

iCa	(from <i>membraneProperties</i>)	<i>current</i>
iCa2		<i>current</i>
totChan-Current	(from <i>membraneProperties</i>)	<i>current</i>
tot-Spec-Cap	(from <i>membraneProperties</i>)	<i>specificCapacitance</i>

Requirements

sur-faceArea		<i>area</i>
sur-faceArea	(from <i>membraneProperties</i>)	<i>area</i>

Dynamics

Derived Variables **totSpecCap** = specificCapacitances[*]->specCap(reduce method: add) (exposed as **totSpecCap**)

totChanPopCurrent = populations[*]->i(reduce method: add)

totChanDensCurrentDensity = channelDensities[*]->iDensity(reduce method: add)

totChanCurrent = totChanPopCurrent + (totChanDensCurrentDensity * surfaceArea) (exposed as **totChanCurrent**)

totChanPopCurrentCa = populations[ion='ca']->i(reduce method: add)

totChanDensCurrentDensityCa = channelDensities[ion='ca']->iDensity(reduce method: add)

iCa = totChanPopCurrentCa + (totChanDensCurrentDensityCa * surfaceArea) (exposed as **iCa**)

totChanPopCurrentCa2 = populations[ion='ca2']->i(reduce method: add)

totChanDensCurrentDensityCa2 = channelDensities[ion='ca2']->iDensity(reduce method: add)

iCa2 = totChanPopCurrentCa2 + (totChanDensCurrentDensityCa2 * surfaceArea) (exposed as **iCa2**)

Usage: Python

Go to the *libNeuroML* documentation

```
from neuroml import MembraneProperties2CaPools

variable = 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, gds_collector_=None, **kwargs_)
```

biophysicalProperties

The biophysical properties of the *cell*, including the *membraneProperties* and the *intracellularProperties*.

Child list

membraneProperties		<i>membraneProperties</i>
intracellularProperties		<i>intracellularProperties</i>

Exposures

totSpecCap		<i>specificCapacitance</i>
------------	--	----------------------------

Dynamics

Derived Variables **totSpecCap** = membraneProperties->totSpecCap (exposed as **totSpecCap**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BiophysicalProperties

variable = BiophysicalProperties(neuro_lex_id=None, id=None, metaid=None, notes=None,
                                 properties=None, annotation=None, membrane_properties=None, intracellular_
                                 properties=None, extracellular_properties=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<biophysicalProperties id="bio_cell">
    <membraneProperties>
        <channelPopulation id="naChansDend" ionChannel="NaConductance"_
        ↵segment="2" number="120000" erev="50mV" ion="na"/>
            <channelDensity id="pasChans" ionChannel="pas" condDensity="3.0 S_per_
            ↵m2" erev="-70mV" ion="non_specific"/>
                <channelDensity id="naChansSoma" ionChannel="NaConductance"_
                ↵segmentGroup="soma_group" condDensity="120.0 mS_per_cm2" erev="50mV" ion="na"/>
                    <specificCapacitance segmentGroup="soma_group" value="1.0 uF_per_cm2"/_
                    ↵>
                    <specificCapacitance segmentGroup="dendrite_group" value="2.0 uF_per_
                    ↵cm2"/>
    </membraneProperties>
    <intracellularProperties>
        <resistivity value="0.1 kohm_cm"/>
```

(continues on next page)

(continued from previous page)

```
</intracellularProperties>
</biophysicalProperties>
```

```
<biophysicalProperties id="PyrCellChanDist">
    <membraneProperties>

        <channelDensity id="naChans" ionChannel="HH_Na" segmentGroup="soma_group"-
        <condDensity value="120.0 mS_per_cm2" ion="na" erev="50mV"/>
            <!-- Ions present inside the cell. Note: a fixed reversal potential is-
            <!-- specified here
            <reversalPotential species="na" value="50mV"/>
            <reversalPotential species="k" value="-77mV"/>-->

    </membraneProperties>
    <intracellularProperties>
        <resistivity value="0.1 kohm_cm"/>
        <!-- REMOVED UNTIL WE CHECK HOW THE USAGE OF LEMS IMPACTS THIS... --
        <biochemistry reactionScheme="InternalCaDynamics"/> Ref to earlier-
        <pathway -->
    </intracellularProperties>
</biophysicalProperties>
```

```
<biophysicalProperties id="biophys">
    <membraneProperties>
        <channelDensityNonUniform id="nonuniform_na_chans" ionChannel=
        <!--NaConductance" erev="50mV" ion="na">
            <variableParameter parameter="condDensity" segmentGroup="dendrite_-
            <!--group">
                <inhomogeneousValue inhomogeneousParameter="dendrite_group_x1-
                <!--" value="5e-7 * exp(-p/200)" />
                </variableParameter>
            </channelDensityNonUniform>
            <specificCapacitance segmentGroup="soma_group" value="1.0 uF_per_cm2"/-
            <!-->
        </membraneProperties>
        <intracellularProperties>
            <resistivity value="0.1 kohm_cm"/>
        </intracellularProperties>
    </biophysicalProperties>
```

biophysicalProperties2CaPools

The biophysical properties of the *cell*, including the *membraneProperties2CaPools* and the *intracellularProperties2CaPools* for a cell with two Ca pools.

Child list

mem- brane- Proper- ties2CaPools		<i>membraneProper- ties2CaPools</i>
intra- cellu- larProp- er- ties2CaPools		<i>intracellularProper- ties2CaPools</i>

Exposures

tot- Spec- Cap		<i>specificCapacitance</i>
-------------------------------	--	----------------------------

Dynamics

Derived Variables **totSpecCap** = *membraneProperties2CaPools->totSpecCap* (exposed as **totSpecCap**)

Usage: Python

Go to the *libNeuroML* documentation

```
from neuroml import BiophysicalProperties2CaPools

variable = 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, gds_ ↵
collector=None, **kwargs_)
```

intracellularProperties

Biophysical properties related to the intracellular space within the *cell*, such as the *resistivity* and the list of ionic *species* present. **caConc** and **caConcExt** are explicitly exposed here to facilitate accessing these values from other Components, even though **caConcExt** is clearly not an intracellular property.

Children list

resistivity		<i>resistivity</i>
speciesList		<i>species</i>

Exposures

caConc		<i>concentration</i>
caConcExt		<i>concentration</i>

Dynamics

Derived Variables **caConc** = speciesList[ion='ca']->concentration(reduce method: add) (exposed as **caConc**)
caConcExt = speciesList[ion='ca']->extConcentration(reduce method: add) (exposed as **caConcExt**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IntracellularProperties

variable = IntracellularProperties(species=None, resistivities=None, extensiontype_=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<intracellularProperties>
    <resistivity value="0.1 kohm_cm"/>
</intracellularProperties>
```

```
<intracellularProperties>
    <resistivity value="0.1 kohm_cm"/>
    <!-- REMOVED UNTIL WE CHECK HOW THE USAGE OF LEMS IMPACTS THIS...
        <biochemistry reactionScheme="InternalCaDynamics"/> Ref to earlier...
    <pathway -->
</intracellularProperties>
```

```
<intracellularProperties>
    <resistivity value="0.1 kohm_cm"/>
</intracellularProperties>
```

intracellularProperties2CaPools

extends *intracellularProperties*

Variant of intracellularProperties with 2 independent Ca pools.

Children list

<i>species-List</i>		<i>species</i>
<i>resistivity</i>		<i>resistivity</i>

Exposures

caConc	(from <i>intracellularProperties</i>)	<i>concentration</i>
ca-Conc2		<i>concentration</i>
caConcExt	(from <i>intracellularProperties</i>)	<i>concentration</i>
caConcExt2		<i>concentration</i>

Dynamics

Derived Variables **caConc2** = speciesList[ion='ca2']->concentration(reduce method: add) (exposed as **caConc2**)

caConcExt2 = speciesList[ion='ca2']->extConcentration(reduce method: add) (exposed as **caConcExt2**)

caConc = speciesList[ion='ca']->concentration(reduce method: add) (exposed as **caConc**)

caConcExt = speciesList[ion='ca']->extConcentration(reduce method: add) (exposed as **caConcExt**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IntracellularProperties2CaPools

variable = IntracellularProperties2CaPools(species=None, resistivities=None, gds_
                                          collector_=None, **kwargs_)
```

resistivity

The resistivity, or specific axial resistance, of the cytoplasm.

Parameters

value	<i>resistivity</i>
--------------	--------------------

Text fields

segment-Group	
----------------------	--

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Resistivity

variable = Resistivity(value=None, segment_groups='all', gds_collector=None,   
    **kwargs)
```

Usage: XML

```
<resistivity value="0.1 kohm_cm"/>
```

```
<resistivity value="0.1 kohm_cm"/>
```

```
<resistivity value="0.1 kohm_cm"/>
```

concentrationModel

Base for any model of an **ion** concentration which changes with time. Internal (**concentration**) and external (**extConcentration**) values for the concentration of the ion are given.

Text fields

ion	
------------	--

Exposures

concentration		<i>concentration</i>
extConcentration		<i>concentration</i>

Requirements

initialConcentration		<i>concentration</i>
initialExtConcentration		<i>concentration</i>
surfaceArea		<i>area</i>

Dynamics

State Variables **concentration**: *concentration* (exposed as **concentration**)

extConcentration: *concentration* (exposed as **extConcentration**)

On Start **concentration** = initialConcentration

extConcentration = initialExtConcentration

decayingPoolConcentrationModel

extends *concentrationModel*

Model of an intracellular buffering mechanism for **ion** (currently hard Coded to be calcium, due to requirement for **iCa**) which has a baseline level **restingConc** and tends to this value with time course **decayConstant**. The ion is assumed to occupy a shell inside the membrane of thickness **shellThickness..**

Parameters

decayConstant		<i>time</i>
restingConc		<i>concentration</i>
shellThickness		<i>length</i>

Text fields

ion	
-----	--

Constants

Faraday = 96485.3C_per_mol		charge_per_mole
AREA_SCALE = 1m ²		area
LENGTH_SCALE = 1m		length

Exposures

concentration	(from concentrationModel)	concentration
extConcentration	(from concentrationModel)	concentration

Requirements

iCa		current
initial-Concentration	(from concentrationModel)	concentration
initialExtConcentration	(from concentrationModel)	concentration
surfaceArea	(from concentrationModel)	area

Dynamics

State Variables concentration: *concentration* (exposed as **concentration**)

extConcentration: *concentration* (exposed as **extConcentration**)

On Start concentration = initialConcentration

extConcentration = initialExtConcentration

On Conditions IF concentration < 0 THEN

concentration = 0

Derived Variables effectiveRadius = LENGTH_SCALE * sqrt(surfaceArea/(AREA_SCALE * (4 * 3.14159)))

innerRadius = effectiveRadius - shellThickness

shellVolume = (4 * (effectiveRadius * effectiveRadius * effectiveRadius) * 3.14159 / 3) - (4 * (innerRadius * innerRadius * innerRadius) * 3.14159 / 3)

Time Derivatives $d \text{ concentration} / dt = i\text{Ca} / (2 * \text{Faraday} * \text{shellVolume}) - ((\text{concentration} - \text{restingConc}) / \text{decayConstant})$

Usage: Python

Go to the *libNeuroML* documentation

```
from neuroml import DecayingPoolConcentrationModel

variable = 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, gds_collector_=None,_
                                         **kwargs_)
```

fixedFactorConcentrationModel

extends *concentrationModel*

Model of buffering of concentration of an ion (currently hard coded to be calcium, due to requirement for **iCa**) which has a baseline level **restingConc** and tends to this value with time course **decayConstant**. A fixed factor **rho** is used to scale the incoming current *independently of the size of the compartment* to produce a concentration change.

Parameters

decay- Con- stant		<i>time</i>
resting- Conc		<i>concentration</i>
rho		<i>rho_factor</i>

Text fields

ion	
------------	--

Exposures

concen- tration	(from <i>concentrationModel</i>)	<i>concentration</i>
extCon- centra- tion	(from <i>concentrationModel</i>)	<i>concentration</i>

Requirements

iCa		<i>current concentration</i>
initial-Concentration	(from concentrationModel)	
initialExtConcentration	(from concentrationModel)	<i>concentration</i>
surfaceArea		<i>area</i>
surfaceArea	(from concentrationModel)	<i>area</i>

Dynamics

State Variables **concentration**: *concentration* (exposed as **concentration**)

extConcentration: *concentration* (exposed as **extConcentration**)

On Start **concentration** = initialConcentration

extConcentration = initialExtConcentration

On Conditions IF concentration < 0 THEN

concentration = 0

Time Derivatives d **concentration** /dt = (iCa/surfaceArea) * rho - ((concentration - restingConc) / decayConstant)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import FixedFactorConcentrationModel

variable = 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, gds_collector_=None, **kwargs_)
```

fixedFactorConcentrationModelTraub

extends *concentrationModel*

Model of buffering of concentration of an ion (currently hard coded to be calcium, due to requirement for **iCa**) which has a baseline level **restingConc** and tends to this value with time course 1 / **beta**. A fixed factor **phi** is used to scale the incoming current *independently of the size of the compartment* to produce a concentration change. Not recommended for use in models other than Traub et al. 2005!

Parameters

beta		<i>per_time</i>
phi		<i>rho_factor</i>
resting-Conc		<i>concentration</i>

Text fields

species	
----------------	--

Exposures

concen-tration	(from concentrationModel)	<i>concentration</i>
extConcen-tration	(from concentrationModel)	<i>concentration</i>

Requirements

iCa		<i>current</i>
initial-Concen-tration	(from concentrationModel)	<i>concentration</i>
ini-tialExtConcen-tration	(from concentrationModel)	<i>concentration</i>
sur-faceArea		<i>area</i>
sur-faceArea	(from concentrationModel)	<i>area</i>

Dynamics

State Variables **concentration**: *concentration* (exposed as **concentration**)

extConcentration: *concentration* (exposed as **extConcentration**)

On Start **concentration** = initialConcentration

extConcentration = initialExtConcentration

On Conditions IF concentration < 0 THEN

concentration = 0

Time Derivatives $d \text{ concentration} /dt = (\text{iCa}/\text{surfaceArea}) * 1e-9 * \phi - ((\text{concentration} - \text{restingConc}) * \beta)$

species

Description of a chemical species identified by **ion**, which has internal, **concentration**, and external, **extConcentration** values for its concentration.

Parameters

initial-Concentration		<i>concentration</i>
initialExtConcentration		<i>concentration</i>

Text fields

ion	
segment-Group	

Component References

concentration-Model		<i>concentrationModel</i>
---------------------	--	---------------------------

Exposures

concentration		<i>concentration</i>
extConcentration		<i>concentration</i>

Dynamics

Structure CHILD INSTANCE: **concentrationModel**

Derived Variables **concentration** = concentrationModel->concentration (exposed as **concentration**)

extConcentration = concentrationModel->extConcentration (exposed as **extConcentration**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Species

variable = Species(id=None, concentration_model=None, ion=None, initial_
    ↪concentration=None, initial_ext_concentration=None, segment_groups='all', gds_
    ↪collector_=None, **kwargs_)
```

cell

extends [baseCellMembPot](#)

Cell with *segments* specified in a *morphology* element along with details on its *biophysicalProperties*. NOTE: this can only be correctly simulated using jLEMS when there is a single segment in the cell, and *v* of this cell represents the membrane potential in that isopotential segment.

Text fields

neu- roLexId	
-------------------------------	--

Child list

mor- phology	Should only be used if morphology element is outside the cell. This points to the id of the morphology.	<i>morphology</i>
bio- phys- ical- Proper- ties	Should only be used if biophysicalProperties element is outside the cell. This points to the id of the biophysicalProperties	<i>biophysicalProperties</i>

Exposures

<code>caConc</code>		<i>concentration</i>
<code>caConcExt</code>		<i>concentration</i>
<code>iCa</code>		<i>current</i>
<code>iChannels</code>		<i>current</i>
<code>iSyn</code>		<i>current</i>
<code>spiking</code>		Dimensionless
<code>surfaceArea</code>		<i>area</i>
<code>totSpecCap</code>		<i>specificCapacitance</i>
<code>v</code>	Membrane potential (<i>from baseCellMembPot</i>)	<i>voltage</i>

Event Ports

<code>spike</code>	Spike event (<i>from baseSpikingCell</i>)	Direction: out
--------------------	---	----------------

Attachments

<code>synapses</code>	<i>basePointCurrent</i>
-----------------------	-------------------------

Dynamics

State Variables `v`: *voltage* (exposed as `v`)

`spiking`: Dimensionless (exposed as `spiking`)

On Start `spiking` = 0

`v` = `initMembPot`

On Conditions IF `v` > `thresh` AND `spiking` < 0.5 THEN

`spiking` = 1

EVENT OUT on port: `spike`

IF `v` < `thresh` THEN

`spiking` = 0

Derived Variables `initMembPot` = `biophysicalProperties->membraneProperties->initMembPotential->value`

`thresh` = `biophysicalProperties->membraneProperties->spikeThresh->value`

`surfaceArea` = `morphology->segments[*]->surfaceArea(reduce method: add)` (exposed as `surfaceArea`)

`totSpecCap` = `biophysicalProperties->totSpecCap` (exposed as `totSpecCap`)

`totCap` = `totSpecCap * surfaceArea`

iChannels = biophysicalProperties->membraneProperties->totChanCurrent (exposed as **iChannels**)

iSyn = synapses[*]->i(reduce method: add) (exposed as **iSyn**)

iCa = biophysicalProperties->membraneProperties->iCa (exposed as **iCa**)

caConc = biophysicalProperties->intracellularProperties->caConc (exposed as **caConc**)

caConcExt = biophysicalProperties->intracellularProperties->caConcExt (exposed as **caConcExt**)

Time Derivatives $d v / dt = (iChannels + iSyn) / totCap$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Cell

variable = 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, gds_collector_=
    None, **kwargs_)
```

Usage: XML

```
<cell id="SpikingCell" metaid="HippoCA1Cell">
    <notes>A Simple Spiking cell for testing purposes</notes>

    <annotation>
        <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"_
        xmlns:bqbiol="http://biomodels.net/biology-qualifiers/">
            <rdf:Description rdf:about="HippoCA1Cell">
                <bqbiol:is>
                    <rdf:Bag>

                        <rdf:li rdf:resource="urn:miriam:neurondb:258"/>
                    </rdf:Bag>
                </bqbiol:is>
            </rdf:Description>
        </rdf:RDF>
    </annotation>
    <morphology id="SpikingCell_morphology">
        <segment id="0" name="Soma">

            <proximal x="0" y="0" z="0" diameter="10"/>
            <distal x="10" y="0" z="0" diameter="10"/>
        </segment>
        <segment id="1" name="Dendrite1">
            <parent segment="0"/>

            <distal x="20" y="0" z="0" diameter="3"/>
        </segment>
        <segment id="2" name="Dendrite2">
            <parent segment="1"/>
            <distal x="30" y="0" z="0" diameter="1"/>
        </segment>
    </morphology>
</cell>
```

(continues on next page)

(continued from previous page)

```

<segment id="3" name="Spine1">
    <parent segment="2" fractionAlong="0.5"/>
    <proximal x="25" y="0" z="0" diameter="0.1"/>
    <distal x="25" y="0.2" z="0" diameter="0.1"/>
</segment>

<segmentGroup id="soma_group" neuroLexId="sao1044911821">
    <member segment="0"/>
</segmentGroup>
<segmentGroup id="dendrite_group" neuroLexId="sao1211023249">
    <member segment="1"/>
    <member segment="2"/>
    <member segment="3"/>
</segmentGroup>
<segmentGroup id="spines" neuroLexId="sao1145756102">
    <member segment="3"/>
</segmentGroup>
</morphology>
<biophysicalProperties id="bio_cell">
    <membraneProperties>
        <channelPopulation id="naChansDend" ionChannel="NaConductance"_
        ↵segment="2" number="120000" erev="50mV" ion="na"/>
        <channelDensity id="pasChans" ionChannel="pas" condDensity="3.0 S_per_-
        ↵m2" erev="-70mV" ion="non_specific"/>
        <channelDensity id="naChansSoma" ionChannel="NaConductance"_
        ↵segmentGroup="soma_group" condDensity="120.0 mS_per_cm2" erev="50mV" ion="na"/>
        <specificCapacitance segmentGroup="soma_group" value="1.0 uF_per_cm2"/
        ↵>
        <specificCapacitance segmentGroup="dendrite_group" value="2.0 uF_per_-
        ↵cm2"/>
    </membraneProperties>
    <intracellularProperties>
        <resistivity value="0.1 kohm_cm"/>
    </intracellularProperties>
</biophysicalProperties>
</cell>

```

```

<cell id="PyrCell" morphology="NeuroMorpho_PyrCell123" biophysicalProperties=
    ↵"PyrCellChanDist"/>

```

```

<cell id="SimpleCell">
    <morphology id="SimpleCell_Morphology">

        <segment id="0" name="Soma">
            <proximal x="0" y="0" z="0" diameter="10"/>
            <distal x="10" y="0" z="0" diameter="10"/>
        </segment>
        <segment id="1" name="MainDendrite1">
            <parent segment="0"/>
            <proximal x="10" y="0" z="0" diameter="3"/>
            <distal x="20" y="0" z="0" diameter="3"/>
        </segment>
        <segment id="2" name="MainDendrite2">

```

(continues on next page)

(continued from previous page)

```

<parent segment="1"/>

    <distal x="30" y="0" z="0" diameter="1"/>
</segment>

<segmentGroup id="soma_group" neuroLexId="sao1044911821">
    <member segment="0"/>
</segmentGroup>
<segmentGroup id="dendrite_group" neuroLexId="sao1211023249">
    <member segment="1"/>
    <member segment="2"/>

        <inhomogeneousParameter id="dendrite_group_x1" variable="p" metric=
        "Path Length from root"/>

        <inhomogeneousParameter id="dendrite_group_x2" variable="r" metric=
        "Path Length from root">
            <proximal translationStart="0"/>
            <distal normalizationEnd="1"/>
        </inhomogeneousParameter>

    </segmentGroup>
</morphology>

<biophysicalProperties id="biophys">
    <membraneProperties>
        <channelDensityNonUniform id="nonuniform_na_chans" ionChannel=
        "NaConductance" erev="50mV" ion="na">
            <variableParameter parameter="condDensity" segmentGroup="dendrite_
            group">
                <inhomogeneousValue inhomogeneousParameter="dendrite_group_x1"
                value="5e-7 * exp(-p/200)"/>
            </variableParameter>
        </channelDensityNonUniform>
        <specificCapacitance segmentGroup="soma_group" value="1.0 uF_per_cm2"/
        >
    </membraneProperties>
    <intracellularProperties>
        <resistivity value="0.1 kohm_cm"/>
    </intracellularProperties>
</biophysicalProperties>
</cell>

```

cell2CaPools

extends [cell](#)

Variant of cell with two independent Ca²⁺ pools. Cell with *segments* specified in a *morphology* element along with details on its *biophysicalProperties*. NOTE: this can only be correctly simulated using jLEMS when there is a single segment in the cell, and **v** of this cell represents the membrane potential in that isopotential segment.

Text fields

neu- roLexId	
-----------------	--

Child list

bio- phys- ical- Proper- ties2CaPools		<i>biophysicalProperties2CaPools</i>
---	--	--------------------------------------

Exposures

caConc	(from cell)	<i>concentration</i>
ca-Conc2		<i>concentration</i>
caCon- cExt	(from cell)	<i>concentration</i>
caCon- cExt2		<i>concentration</i>
iCa	(from cell)	<i>current</i>
iCa2		<i>current</i>
iChan- nels	(from cell)	<i>current</i>
iSyn	(from cell)	<i>current</i>
spiking	(from cell)	Dimensionless
sur- faceArea	(from cell)	<i>area</i>
tot- Spec- Cap	(from cell)	<i>specificCapacitance</i>
v	Membrane potential (from baseCellMembPot)	<i>voltage</i>

Event Ports

spike	Spike event (from baseSpikingCell)	Direction: out
-------	------------------------------------	----------------

Attachments

<code>synapses</code>	<code>basePointCurrent</code>
-----------------------	-------------------------------

Dynamics

State Variables `v`: *voltage* (exposed as `v`)

`spiking`: Dimensionless (exposed as `spiking`)

On Start `spiking` = 0

`v` = `initMembPot`

On Conditions IF `v` > `thresh` AND `spiking` < 0.5 THEN

`spiking` = 1

EVENT OUT on port: `spike`

IF `v` < `thresh` THEN

`spiking` = 0

Derived Variables `initMembPot` = `biophysicalProperties2CaPools->membraneProperties2CaPools->initMembPotential->value`

`thresh` = `biophysicalProperties2CaPools->membraneProperties2CaPools->spikeThresh->value`

`surfaceArea` = `morphology->segments[*]->surfaceArea(reduce method: add)` (exposed as `surfaceArea`)

`totSpecCap` = `biophysicalProperties2CaPools->totSpecCap` (exposed as `totSpecCap`)

`totCap` = `totSpecCap * surfaceArea`

`iChannels` = `biophysicalProperties2CaPools->membraneProperties2CaPools->totChanCurrent` (exposed as `iChannels`)

`iSyn` = `synapses[*]->i(reduce method: add)` (exposed as `iSyn`)

`iCa` = `biophysicalProperties2CaPools->membraneProperties2CaPools->iCa` (exposed as `iCa`)

`caConc` = `biophysicalProperties2CaPools->intracellularProperties2CaPools->caConc` (exposed as `caConc`)

`caConcExt` = `biophysicalProperties2CaPools->intracellularProperties2CaPools->caConcExt` (exposed as `caConcExt`)

`iCa2` = `biophysicalProperties2CaPools->membraneProperties2CaPools->iCa2` (exposed as `iCa2`)

`caConc2` = `biophysicalProperties2CaPools->intracellularProperties2CaPools->caConc2` (exposed as `caConc2`)

`caConcExt2` = `biophysicalProperties2CaPools->intracellularProperties2CaPools->caConcExt2` (exposed as `caConcExt2`)

Time Derivatives $d v / dt = (iChannels + iSyn) / totCap$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Cell2CaPools

variable = 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, gds_collector_=None, **kwargs_)
```

baseCellMembPotCap

extends baseCellMembPot

Any cell with a membrane potential **v** with voltage units and a membrane capacitance **C**. Also defines exposed value **iSyn** for current due to external synapses and **iMemb** for total transmembrane current (usually channel currents plus **iSyn**).

Parameters

C	Total capacitance of the cell membrane	<i>capacitance</i>
----------	--	--------------------

Exposures

iMemb	Total current crossing the cell membrane	<i>current</i>
iSyn	Total current due to synaptic inputs	<i>current</i>
v	Membrane potential (<i>from baseCellMembPot</i>)	<i>voltage</i>

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
--------------	---	----------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BaseCellMembPotCap

variable = BaseCellMembPotCap(neuro_lex_id=None, id=None, metaid=None, notes=None,_
    ↪properties=None, annotation=None, C=None, extensiontype_=None, gds_collector_=None,_
    ↪**kwargs_)
```

baselaf

extends [baseCellMembPot](#)

Base ComponentType for an integrate and fire cell which emits a spiking event at membrane potential **thresh** and and resets to **reset**.

Parameters

reset	The value the membrane potential is reset to on spiking	<i>voltage</i>
thresh	The membrane potential at which to emit a spiking event and reset voltage	<i>voltage</i>

Exposures

v	Membrane potential (<i>from baseCellMembPot</i>)	<i>voltage</i>
----------	--	----------------

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
--------------	---	----------------

iafTauCell

extends [baseIaf](#)

Integrate and fire cell which returns to its leak reversal potential of **leakReversal** with a time constant **tau**.

Parameters

leakReversal		<i>voltage</i>
reset	The value the membrane potential is reset to on spiking (<i>from baseIaf</i>)	<i>voltage</i>
tau		<i>time</i>
thresh	The membrane potential at which to emit a spiking event and reset voltage (<i>from baseIaf</i>)	<i>voltage</i>

Exposures

v	Membrane potential (<i>from baseCellMembPot</i>)	<i>voltage</i>
----------	--	----------------

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
--------------	---	----------------

Dynamics

State Variables v : *voltage* (exposed as v)

On Start $v = \text{leakReversal}$

On Conditions IF $v > \text{thresh}$ THEN

$v = \text{reset}$

EVENT OUT on port: **spike**

Time Derivatives $\frac{d v}{dt} = (\text{leakReversal} - v) / \tau$

Usage: Python

Go to the *libNeuroML* documentation

```
from neuroml import IafTauCell

variable = 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, gds_collector=None, **kwargs)
```

Usage: XML

```
<iafTauCell id="iafTau" leakReversal="-50mV" thresh="-55mV" reset="-70mV" tau="30ms"/>
```

iafTauRefCell

extends *iafTauCell*

Integrate and fire cell which returns to its leak reversal potential of **leakReversal** with a time course **tau**. It has a refractory period of **refract** after spiking.

Parameters

leakReversal	(<i>from iafTauCell</i>)	<i>voltage</i>
refract		<i>time</i>
reset	The value the membrane potential is reset to on spiking (<i>from baseIaf</i>)	<i>voltage</i>
tau	(<i>from iafTauCell</i>)	<i>time</i>
thresh	The membrane potential at which to emit a spiking event and reset voltage (<i>from baseIaf</i>)	<i>voltage</i>

Exposures

v	Membrane potential (<i>from baseCellMembPot</i>)	voltage
---	--	---------

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
-------	---	----------------

Dynamics

State Variables v: *voltage* (exposed as v)

lastSpikeTime: *time*

On Start v = leakReversal

Regime: refractory (initial) On Entry

lastSpikeTime = t

v = reset

On Conditions

IF t > lastSpikeTime + refract THEN

TRANSITION to REGIME integrating

Regime: integrating (initial) On Conditions

IF v > thresh THEN

EVENT OUT on port: spike

TRANSITION to REGIME refractory

Time Derivatives

d v /dt = (leakReversal - v) / tau

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IafTauRefCell

variable = 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, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<iafTauRefCell id="iafTauRef" leakReversal="-50mV" thresh="-55mV" reset="-70mV" tau=
↳ "30ms" refract="5ms"/>
```

baseIafCapCellextends [baseCellMembPotCap](#)Base Type for all Integrate and Fire cells with a capacitance **C**, threshold **thresh** and reset membrane potential **reset**.**Parameters**

C	Total capacitance of the cell membrane (<i>from baseCellMembPotCap</i>)	<i>capacitance</i>
reset		<i>voltage</i>
thresh		<i>voltage</i>

Exposures

iMemb	Total current crossing the cell membrane (<i>from baseCellMembPotCap</i>)	<i>current</i>
iSyn	Total current due to synaptic inputs (<i>from baseCellMembPotCap</i>)	<i>current</i>
v	Membrane potential (<i>from baseCellMembPot</i>)	<i>voltage</i>

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
--------------	---	----------------

iafCellextends [baseIafCapCell](#)Integrate and fire cell with capacitance **C**, **leakConductance** and **leakReversal**.**Parameters**

C	Total capacitance of the cell membrane (<i>from baseCellMembPotCap</i>)	<i>capacitance</i>
leak-Conduc-tance		<i>conductance</i>
leakRe-versal		<i>voltage</i>
reset	(<i>from baseIafCapCell</i>)	<i>voltage</i>
thresh	(<i>from baseIafCapCell</i>)	<i>voltage</i>

Exposures

iMemb	Total current crossing the cell membrane (<i>from baseCellMembPotCap</i>)	<i>current</i>
iSyn	Total current due to synaptic inputs (<i>from baseCellMembPotCap</i>)	<i>current</i>
v	Membrane potential (<i>from baseCellMembPot</i>)	<i>voltage</i>

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
--------------	---	----------------

Attachments

synapses	<i>basePointCurrent</i>
-----------------	-------------------------

Dynamics

State Variables **v**: *voltage* (exposed as **v**)

On Start **v** = leakReversal

On Conditions IF **v** > thresh THEN

v = reset

EVENT OUT on port: **spike**

Derived Variables **iSyn** = synapses[*]->i(reduce method: add) (exposed as **iSyn**)

iMemb = leakConductance * (leakReversal - **v**) + **iSyn** (exposed as **iMemb**)

Time Derivatives $d v / dt = iMemb / C$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IafCell

variable = 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, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<iafCell id="iaf" leakReversal="-50mV" thresh="-55mV" reset="-70mV" C="0.2nF" ↴
  leakConductance="0.01uS"/>
```

```
<iafCell id="iaf" leakConductance="0.2nS" leakReversal="-70mV" thresh="-55mV" reset="-
  ↴70mV" C="3.2pF"/>
```

```
<iafCell id="iaf" leakConductance="0.2nS" leakReversal="-70mV" thresh="-55mV" reset="-
  ↴70mV" C="3.2pF"/>
```

iafRefCellextends *iafCell*Integrate and fire cell with capacitance **C**, **leakConductance**, **leakReversal** and refractory period **refract**.**Parameters**

C	Total capacitance of the cell membrane (<i>from baseCellMembPotCap</i>)	<i>capacitance</i>
leak-Conduc-tance	(<i>from iafCell</i>)	<i>conductance</i>
leakRe-versal	(<i>from iafCell</i>)	<i>voltage</i>
refract		<i>time</i>
reset	(<i>from baseIafCapCell</i>)	<i>voltage</i>
thresh	(<i>from baseIafCapCell</i>)	<i>voltage</i>

Exposures

iMemb	Total current crossing the cell membrane (<i>from baseCellMembPotCap</i>)	<i>current</i>
iSyn	Total current due to synaptic inputs (<i>from baseCellMembPotCap</i>)	<i>current</i>
v	Membrane potential (<i>from baseCellMembPot</i>)	<i>voltage</i>

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
--------------	---	----------------

Attachments

synapses	<i>basePointCurrent</i>
----------	-------------------------

Dynamics

State Variables **v**: *voltage* (exposed as **v**)

lastSpikeTime: *time*

On Start **v** = leakReversal

Derived Variables **iSyn** = synapses[*]->i(reduce method: add) (exposed as **iSyn**)

iMemb = leakConductance * (leakReversal - v) + iSyn (exposed as **iMemb**)

Regime: refractory (initial) **On Entry**

lastSpikeTime = t

v = reset

On Conditions

IF t > lastSpikeTime + refract THEN

TRANSITION to REGIME **integrating**

Regime: integrating (initial) **On Conditions**

IF v > thresh THEN

EVENT OUT on port: **spike**

TRANSITION to REGIME **refractory**

Time Derivatives

$d v /dt = iMemb / C$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IafRefCell

variable = 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, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<iafRefCell id="iafRef" leakReversal="-50mV" thresh="-55mV" reset="-70mV" C="0.2nF" ↴  
 leakConductance="0.01uS" refract="5ms"/>
```

izhikevichCell

extends [baseCellMembPot](#)

Cell based on the 2003 model of Izhikevich, see <http://izhikevich.org/publications/spikes.htm>.

Parameters

a	Time scale of the recovery variable U	Dimensionless
b	Sensitivity of U to the subthreshold fluctuations of the membrane potential V	Dimensionless
c	After-spike reset value of V	Dimensionless
d	After-spike increase to U	Dimensionless
thresh	Spike threshold	<i>voltage</i>
v0	Initial membrane potential	<i>voltage</i>

Constants

MSEC = 1ms		<i>time</i>
MVOLT = 1mV		<i>voltage</i>

Exposures

U		Dimensionless
v	Membrane potential (<i>from baseCellMembPot</i>)	<i>voltage</i>

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
--------------	---	----------------

Attachments

synapses	<i>basePointCurrentDL</i>
-----------------	---------------------------

Dynamics

State Variables v : *voltage* (exposed as v)

U : Dimensionless (exposed as U)

On Start $v = v_0$

$U = v_0 * b / \text{MVOLT}$

On Conditions IF $v > \text{thresh}$ THEN

$v = c * \text{MVOLT}$

$U = U + d$

EVENT OUT on port: **spike**

Derived Variables $\text{ISyn} = \text{synapses}[*] \rightarrow I$ (reduce method: add)

Time Derivatives $d v / dt = (0.04 * v^2 / \text{MVOLT} + 5 * v + (140.0 - U + \text{ISyn}) * \text{MVOLT}) / \text{MSEC}$

$d U / dt = a * (b * v / \text{MVOLT} - U) / \text{MSEC}$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IzhikevichCell

variable = 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, gds_collector=None, **kwargs_)
```

Usage: XML

```
<izhikevichCell id="izBurst" v0="-70mV" thresh="30mV" a="0.02" b="0.2" c="-50.0" d="2
   "/>
```

izhikevich2007Cell

extends `baseCellMembPotCap`

Cell based on the modified Izhikevich model in Izhikevich 2007, *Dynamical systems in neuroscience*, MIT Press.

Parameters

C	Total capacitance of the cell membrane (<i>from baseCellMembPotCap</i>)	<i>capacitance</i>
a		<i>per_time</i>
b		<i>conductance</i>
c		<i>voltage</i>
d		<i>current</i>
k		<i>conductance_per_voltage</i>
v0		<i>voltage</i>
vpeak		<i>voltage</i>
vr		<i>voltage</i>
vt		<i>voltage</i>

Exposures

iMemb	Total current crossing the cell membrane (<i>from baseCellMembPotCap</i>)	<i>current</i>
iSyn	Total current due to synaptic inputs (<i>from baseCellMembPotCap</i>)	<i>current</i>
u		<i>current</i>
v	Membrane potential (<i>from baseCellMembPot</i>)	<i>voltage</i>

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
--------------	---	----------------

Attachments

synapses	<i>basePointCurrent</i>
-----------------	-------------------------

Dynamics

State Variables **v**: *voltage* (exposed as **v**)

u: *current* (exposed as **u**)

On Start **v** = **v0**

u = 0

On Conditions IF **v** > **vpeak** THEN

v = **c**

u = **u** + **d**

 EVENT OUT on port: **spike**

Derived Variables **iSyn** = **synapses**[*]->i(reduce method: add) (exposed as **iSyn**)

iMemb = **k** * (**v**-**vr**) * (**v**-**vt**) + **iSyn** - **u** (exposed as **iMemb**)

Time Derivatives $d v / dt = i_{Memb} / C$

$$d u / dt = a * (b * (v - v_r) - u)$$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Izhikevich2007Cell

variable = 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, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<izhikevich2007Cell id="iz2007RS" v0="-60mV" C="100 pF" k="0.7 nS_per_mV" vr="-60 mV"_
    <vt="-40 mV" vpeak="35 mV" a="0.03 per_ms" b="-2 nS" c="-50 mV" d="100 pA"/>
```

adExIaFCell

extends `baseCellMembPotCap`

Model based on Brette R and Gerstner W (2005) Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity. J Neurophysiol 94:3637-3642.

Parameters

C	Total capacitance of the cell membrane (<i>from baseCellMembPotCap</i>)	<i>capacitance</i>
EL		<i>voltage</i>
VT		<i>voltage</i>
a		<i>conductance</i>
b		<i>current</i>
delT		<i>voltage</i>
gL		<i>conductance</i>
refract		<i>time</i>
reset		<i>voltage</i>
tauw		<i>time</i>
thresh		<i>voltage</i>

Exposures

iMemb	Total current crossing the cell membrane (<i>from baseCellMembPotCap</i>)	<i>current</i>
iSyn	Total current due to synaptic inputs (<i>from baseCellMembPotCap</i>)	<i>current</i>
v	Membrane potential (<i>from baseCellMembPot</i>)	<i>voltage</i>
w		<i>current</i>

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
--------------	---	----------------

Attachments

synapses	<i>basePointCurrent</i>
-----------------	-------------------------

Dynamics

State Variables **v**: *voltage* (exposed as **v**)

w: *current* (exposed as **w**)

lastSpikeTime: *time*

On Start **v** = EL

w = 0

Derived Variables **iSyn** = synapses[*]->i(reduce method: add) (exposed as **iSyn**)

iMemb = -1 * gL * (v - EL) + gL * delT * exp((v - VT) / delT) - w + iSyn (exposed as **iMemb**)

Regime: refractory (initial) On Entry

lastSpikeTime = t

v = reset

w = w + b

On Conditions

IF t > lastSpikeTime + refract THEN

TRANSITION to REGIME integrating

Time Derivatives

$d w / dt = (a * (v - EL) - w) / tau_w$

Regime: integrating (initial) On Conditions

IF v > thresh THEN

EVENT OUT on port: **spike**

TRANSITION to REGIME refractory

Time Derivatives

$$\begin{aligned} \frac{d v}{dt} &= i_{Memb} / C \\ \frac{d w}{dt} &= (a * (v - E_L) - w) / \tau_{uw} \end{aligned}$$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import AdExIaFCell

variable = 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, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<adExIaFCell id="adExBurst" C="281pF" gL="30nS" EL="-70.6mV" reset="-48.5mV" VT="-50.4mV" thresh="-40.4mV" refract="0ms" delT="2mV" tauw="40ms" a="4nS" b="0.08nA"/>
```

fitzHughNagumoCell

extends `baseCellMembPotDL`

Simple dimensionless model of spiking cell from FitzHugh and Nagumo. Superseded by `fitzHughNagumo1969Cell` (See <https://github.com/NeuroML/NeuroML2/issues/42>).

Parameters

I		Dimensionless
---	--	---------------

Constants

SEC = 1s		time
----------	--	------

Exposures

V	Membrane potential (<i>from baseCellMembPotDL</i>)	Dimensionless
W		Dimensionless

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
-------	---	----------------

Dynamics

State Variables V : Dimensionless (exposed as \mathbf{V})

W : Dimensionless (exposed as \mathbf{W})

Time Derivatives $d \mathbf{V} / dt = ((V - ((V^3) / 3)) - W + I) / SEC$

$d \mathbf{W} / dt = (0.08 * (V + 0.7 - 0.8 * W)) / SEC$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import FitzHughNagumoCell

variable = FitzHughNagumoCell(neuro_lex_id=None, id=None, metaid=None, notes=None, properties=None, annotation=None, I=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<fitzHughNagumoCell id="fn1" I="0.8"/>
```

pinskyRinzelCA3Cell

extends `baseCellMembPot`

Reduced CA3 cell model from Pinsky and Rinzel 1994. See <https://github.com/OpenSourceBrain/PinskyRinzelModel>.

Parameters

alphac		Dimensionless
betac		Dimensionless
cm		<i>specificCapacitance</i>
eCa		<i>voltage</i>
eK		<i>voltage</i>
eL		<i>voltage</i>
eNa		<i>voltage</i>
gAmpa		<i>conductanceDensity</i>
gCa		<i>conductanceDensity</i>
gKC		<i>conductanceDensity</i>
gKahp		<i>conductanceDensity</i>
gKdr		<i>conductanceDensity</i>
gLd		<i>conductanceDensity</i>
gLs		<i>conductanceDensity</i>
gNa		<i>conductanceDensity</i>
gNmda		<i>conductanceDensity</i>
gc		<i>conductanceDensity</i>
iDend		<i>currentDensity</i>
iSoma		<i>currentDensity</i>
pp		Dimensionless
qd0		Dimensionless

Constants

MSEC = 1 ms		<i>time</i>
MVOLT = 1 mV		<i>voltage</i>
UAMP_PER_CM2 = 1 uA_per_cm2		<i>currentDensity</i>
Smax = 125.0		Dimensionless
Vsyn = 60.0 mV		<i>voltage</i>
betaqd = 0.001		Dimensionless

Exposures

Cad		Dimensionless
ICad		<i>currentDensity</i>
Si		Dimensionless
Vd		<i>voltage</i>
Vs		<i>voltage</i>
Wi		Dimensionless
cd		Dimensionless
hs		Dimensionless
ns		Dimensionless
qd		Dimensionless
sd		Dimensionless
v	Membrane potential (<i>from baseCellMembPot</i>)	<i>voltage</i>

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
--------------	---	----------------

Dynamics

State Variables **Vs**: *voltage* (exposed as **Vs**)

Vd: *voltage* (exposed as **Vd**)

Cad: Dimensionless (exposed as **Cad**)

hs: Dimensionless (exposed as **hs**)

ns: Dimensionless (exposed as **ns**)

sd: Dimensionless (exposed as **sd**)

cd: Dimensionless (exposed as **cd**)

qd: Dimensionless (exposed as **qd**)

Si: Dimensionless (exposed as **Si**)

Wi: Dimensionless (exposed as **Wi**)

Sisat: Dimensionless

On Start **Vs** = eL

Vd = eL

qd = qd0

Derived Variables **v** = **Vs** (exposed as **v**)

ICad = gCasdsd*(Vd-eCa) (exposed as **ICad**)

alphams_Vs = 0.32*(-46.9-Vs/MVOLT)/(exp((-46.9-Vs/MVOLT)/4.0)-1.0)

betams_Vs = 0.28*(Vs/MVOLT+19.9)/(exp((Vs/MVOLT+19.9)/5.0)-1.0)

Minfs_Vs = alphams_Vs/(alphams_Vs+betams_Vs)

alphans_Vs = 0.016*(-24.9-Vs/MVOLT)/(exp((-24.9-Vs/MVOLT)/5.0)-1.0)

betans_Vs = 0.25exp(-1.0-0.025Vs/MVOLT)

alphahs_Vs = 0.128*exp((-43.0-Vs/MVOLT)/18.0)

betahs_Vs = 4.0/(1.0+exp((-20.0-Vs/MVOLT)/5.0))

alphasd_Vd = 1.6/(1.0+exp(-0.072*(Vd/MVOLT-5.0)))

betasd_Vd = 0.02*(Vd/MVOLT+8.9)/(exp((Vd/MVOLT+8.9)/5.0)-1.0)

Iampa = gAmpaWi(Vd-Vsyn)

Inmda = gNmdaSisat(Vd-Vsyn)/(1.0+0.28exp(-0.062(Vd/MVOLT-60.0)))

Isyn = Iampa+Inmda

Conditional Derived Variables IF 0.00002*Cad > 0.01 THEN

alphaqd = 0.01

OTHERWISE

```

alphaqd = 0.00002*Cad
IF Cad/250 > 1 THEN
    chid = 1
ELSEWISE
    chid = Cad/250
IF Vd < -10*MVOLT THEN
    alphacd_Vd = exp((Vd/MVOLT+50.0)/11-(Vd/MVOLT+53.5)/27)/18.975
ELSEWISE
    alphacd_Vd = 2.0*exp((-53.5-Vd/MVOLT)/27.0)
IF Vd < -10*MVOLT THEN
    betacd_Vd = (2.0*exp((-53.5-Vd/MVOLT)/27.0)-alphacd_Vd)
ELSEWISE
    betacd_Vd = 0
IF Si > Smax THEN
    Sisat = Smax
ELSEWISE
    Sisat = Si

Time Derivatives d Vs /dt = (-gLs*(Vs-eL)-gNa*(Mins_Vs^2)*hs(Vs-eNa)-gKdrns(Vs-eK)+(gc/pp)*(Vd-Vs)+iSoma/pp) / cm
d Vd /dt = (iDend/(1.0-pp)-Isyn/(1.0-pp)-gLd*(Vd-eL)-ICad-gKahpqd(Vd-eK)-gKCcdchid*(Vd-eK)+(gc*(Vs-Vd))/(1.0-pp)) / cm
d Cad /dt = (-0.13*ICad/UAMP_PER_CM2-0.075*Cad) / MSEC
d hs /dt = (alphahs_Vs-(alphahs_Vs+betahs_Vs)*hs) / MSEC
d ns /dt = (alphans_Vs-(alphans_Vs+betans_Vs)*ns) / MSEC
d sd /dt = (alphasd_Vd-(alphasd_Vd+betasd_Vd)*sd) / MSEC
d cd /dt = (alphacd_Vd-(alphacd_Vd+betacd_Vd)*cd) / MSEC
d qd /dt = (alphaqd-(alphaqd+betaqd)*qd) / MSEC
d Si /dt = -Si/150.0
d Wi /dt = -Wi/2.0

```

Usage: Python

Go to the libNeuroML documentation

```

from neuroml import PinskyRinzelCA3Cell

variable = 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_
                                l_d=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, gds_collector=None, **kwargs_)

(continues on next page)

```

(continued from previous page)

Usage: XML

```
<pinskyRinzelCA3Cell id="pr2A" iSoma="0.75 uA_per_cm2" iDend="0 uA_per_cm2" gc="2.1_
↳ mS_per_cm2" qd0="0" gLs="0.1 mS_per_cm2" gLd="0.1 mS_per_cm2" gNa="30 mS_per_cm2"_
↳ gKdr="15 mS_per_cm2" gCa="10 mS_per_cm2" gKahp="0.8 mS_per_cm2" gKC="15 mS_per_cm2"_
↳ eNa="60 mV" eCa="80 mV" eK="-75 mV" eL="-60 mV" pp="0.5" cm="3 uF_per_cm2" alphac="2
↳ betac="0.1" gNmida="0 mS_per_cm2" gAmpa="0 mS_per_cm2"/>
```

10.1.5 Channels

Defines voltage (and concentration) gated ion channel models. Ion channels will generally extend *baseIonChannel*. The most commonly used voltage dependent gate will extend *baseGate*.

Original ComponentType definitions: [Channels.xml](#). Schema against which NeuroML based on these should be valid: [NeuroML_v2.2.xsd](#). Generated on 07/06/22 from [this commit](#). Please file any issues or questions at the [issue tracker](#) here.

baseVoltageDepRate

Base ComponentType for voltage dependent rate. Produces a time varying rate **r** which depends on **v..**

Exposures

r	<i>per_time</i>
----------	-----------------

Requirements

v	<i>voltage</i>
----------	----------------

baseVoltageConcDepRate

extends *baseVoltageDepRate*

Base ComponentType for voltage and concentration dependent rate. Produces a time varying rate **r** which depends on **v** and **caConc..**

Exposures

r	(from baseVoltageDepRate)	<i>per_time</i>
----------	---------------------------	-----------------

Requirements

caConc		<i>concentration</i>
v	(from baseVoltageDepRate)	<i>voltage</i>

baseHHRate

extends baseVoltageDepRate

Base ComponentType for rate which follow one of the typical forms for rate equations in the standard HH formalism, using the parameters **rate**, **midpoint** and **scale**.

Parameters

mid-point		<i>voltage</i>
rate		<i>per_time</i>
scale		<i>voltage</i>

Exposures

r	(from baseVoltageDepRate)	<i>per_time</i>
----------	---------------------------	-----------------

Requirements

v	(from baseVoltageDepRate)	<i>voltage</i>
----------	---------------------------	----------------

HHExpRate

extends baseHHRate

Exponential form for rate equation (Q: Should these be renamed hhExpRate, etc?).

Parameters

mid-point	(from baseHHRate)	voltage
rate	(from baseHHRate)	per_time
scale	(from baseHHRate)	voltage

Exposures

r	(from baseVoltageDepRate)	per_time
----------	---------------------------	----------

Requirements

v	(from baseVoltageDepRate)	voltage
----------	---------------------------	---------

Dynamics

Derived Variables $r = \text{rate} * \exp((v - \text{midpoint})/\text{scale})$ (exposed as **r**)

HHSigmoidRate

extends `baseHHRate`

Sigmoidal form for rate equation.

Parameters

mid-point	(from baseHHRate)	voltage
rate	(from baseHHRate)	per_time
scale	(from baseHHRate)	voltage

Exposures

r	(from baseVoltageDepRate)	per_time
----------	---------------------------	----------

Requirements

v	(from baseVoltageDepRate)	voltage
----------	---------------------------	---------

Dynamics

Derived Variables $r = \text{rate} / (1 + \exp(0 - (v - \text{midpoint})/\text{scale}))$ (exposed as **r**)

HHExpLinearRate

extends [baseHHRate](#)

Exponential linear form for rate equation. Linear for large positive **v**, exponentially decays for large negative **v**.

Parameters

mid-point	(from baseHHRate)	voltage
rate	(from baseHHRate)	per_time
scale	(from baseHHRate)	voltage

Exposures

r	(from baseVoltageDepRate)	per_time
----------	---------------------------	----------

Requirements

v	(from baseVoltageDepRate)	voltage
----------	---------------------------	---------

Dynamics

Derived Variables $x = (v - \text{midpoint}) / \text{scale}$

Conditional Derived Variables IF $x \neq 0$ THEN

$r = \text{rate} * x / (1 - \exp(0 - x))$ (exposed as **r**)

IF $x = 0$ THEN

$r = \text{rate}$ (exposed as **r**)

baseVoltageDepVariable

Base ComponentType for voltage dependent variable **x**, which depends on **v**. Can be used for inf/steady state of rate variable.

Exposures

x		Dimensionless
----------	--	---------------

Requirements

v		voltage
----------	--	---------

baseVoltageConcDepVariable

extends [baseVoltageDepVariable](#)

Base ComponentType for voltage and calcium concentration dependent variable **x**, which depends on **v** and **caConc..**

Exposures

x	(from baseVoltageDepVariable)	Dimensionless
----------	--	---------------

Requirements

caConc		concentration
v	(from baseVoltageDepVariable)	voltage

baseHHVariable

extends [baseVoltageDepVariable](#)

Base ComponentType for voltage dependent dimensionless variable which follow one of the typical forms for variable equations in the standard HH formalism, using the parameters **rate**, **midpoint**, **scale**.

Parameters

mid-point		voltage
rate		Dimensionless
scale		voltage

Exposures

x	(from baseVoltageDepVariable)	Dimensionless
----------	-------------------------------	---------------

Requirements

v	(from baseVoltageDepVariable)	voltage
----------	-------------------------------	---------

HHExpVariable

extends baseHHVariable

Exponential form for variable equation.

Parameters

mid-point	(from baseHHVariable)	voltage
rate	(from baseHHVariable)	Dimensionless
scale	(from baseHHVariable)	voltage

Exposures

x	(from baseVoltageDepVariable)	Dimensionless
----------	-------------------------------	---------------

Requirements

v	(from baseVoltageDepVariable)	voltage
----------	-------------------------------	---------

Dynamics

Derived Variables $x = \text{rate} * \exp((v - \text{midpoint})/\text{scale})$ (exposed as **x**)

HHSigmoidVariable

extends baseHHVariable

Sigmoidal form for variable equation.

Parameters

mid-point	(from baseHHVariable)	voltage
rate	(from baseHHVariable)	Dimensionless
scale	(from baseHHVariable)	voltage

Exposures

x	(from baseVoltageDepVariable)	Dimensionless
----------	-------------------------------	---------------

Requirements

v	(from baseVoltageDepVariable)	voltage
----------	-------------------------------	---------

Dynamics

Derived Variables $x = \text{rate} / (1 + \exp(0 - (v - \text{midpoint})/\text{scale}))$ (exposed as **x**)

HHExpLinearVariable

extends `baseHHVariable`

Exponential linear form for variable equation. Linear for large positive **v**, exponentially decays for large negative **v**.

Parameters

mid-point	(from baseHHVariable)	voltage
rate	(from baseHHVariable)	Dimensionless
scale	(from baseHHVariable)	voltage

Exposures

x	(from baseVoltageDepVariable)	Dimensionless
----------	-------------------------------	---------------

Requirements

v	(from baseVoltageDepVariable)	voltage
---	-------------------------------	---------

Dynamics

Derived Variables $a = (v - \text{midpoint}) / \text{scale}$

$x = \text{rate} * a / (1 - \exp(0 - a))$ (exposed as x)

baseVoltageDepTime

Base ComponentType for voltage dependent ComponentType producing value **t** with dimension time (e.g. for time course of rate variable). Note: time course would not normally be fit to exp/sigmoid etc.

Exposures

t		time
---	--	------

Requirements

v		voltage
---	--	---------

baseVoltageConcDepTime

extends baseVoltageDepTime

Base type for voltage and calcium concentration dependent ComponentType producing value **t** with dimension time (e.g. for time course of rate variable).

Exposures

t	(from baseVoltageDepTime)	time
---	---------------------------	------

Requirements

caConc		concentration
v	(from baseVoltageDepTime)	voltage

fixedTimeCourse

extends [baseVoltageDepTime](#)

Time course of a fixed magnitude **tau** which can be used for the time course in *gateHHtauInf*, *gateHhratesTau* or *gateHhratesTauInf*.

Parameters

tau	(from baseVoltageDepTime)	<i>time</i>
------------	--	-------------

Exposures

t	(from baseVoltageDepTime)	<i>time</i>
----------	--	-------------

Requirements

v	(from baseVoltageDepTime)	<i>voltage</i>
----------	--	----------------

Dynamics

Derived Variables **t** = tau (exposed as **t**)

baseQ10Settings

Base ComponentType for a scaling to apply to gating variable time course, usually temperature dependent.

Exposures

q10		Dimensionless
------------	--	---------------

Requirements

temper- ature		<i>temperature</i>
--------------------------	--	--------------------

q10Fixed

extends [baseQ10Settings](#)

A fixed value, **fixedQ10**, for the scaling of the time course of the gating variable.

Parameters

fixedQ10	Dimensionless
-----------------	---------------

Exposures

q10	(from baseQ10Settings)	Dimensionless
------------	---	---------------

Requirements

temperature	(from baseQ10Settings)	<i>temperature</i>
--------------------	---	--------------------

Dynamics

Derived Variables **q10** = fixedQ10 (exposed as **q10**)

q10ExpTemp

extends [baseQ10Settings](#)

A value for the Q10 scaling which varies as a standard function of the difference between the current temperature, **temperature**, and the temperature at which the gating variable equations were determined, **experimentalTemp**.

Parameters

experimental-Temp		<i>temperature</i>
q10Factor		Dimensionless

Constants

TENDEGREES = 10K	<i>temperature</i>
------------------	--------------------

Exposures

q10	(from baseQ10Settings)	Dimensionless
-----	------------------------	---------------

Requirements

temperature	(from baseQ10Settings)	<i>temperature</i>
-------------	------------------------	--------------------

Dynamics

Derived Variables **q10** = q10Factor^{(({temperature} - {experimentalTemp})/{TENDEGREES})} (exposed as **q10**)

baseConductanceScaling

Base ComponentType for a scaling to apply to a gate's conductance, e.g. temperature dependent scaling.

Exposures

factor		Dimensionless
--------	--	---------------

Requirements

temperature		<i>temperature</i>
-------------	--	--------------------

q10ConductanceScaling

extends *baseConductanceScaling*

A value for the conductance scaling which varies as a standard function of the difference between the current temperature, **temperature**, and the temperature at which the conductance was originally determined, **experimentalTemp**.

Parameters

experimentalTemp		<i>temperature</i>
q10Factor		Dimensionless

Constants

TENDEGREES = 10K		<i>temperature</i>
-------------------------	--	--------------------

Exposures

factor	(from <code>baseConductanceScaling</code>)	Dimensionless
---------------	---	---------------

Requirements

temperature	(from <code>baseConductanceScaling</code>)	<i>temperature</i>
--------------------	---	--------------------

Dynamics

Derived Variables `factor` = `q10Factor^(temperature - experimentalTemp)/TENDEGREES` (exposed as **factor**)

Usage: Python

Go to the *libNeuroML documentation*

```
from neuroml import Q10ConductanceScaling

variable = Q10ConductanceScaling(q10_factor=None, experimental_temp=None, gds_
    ↴collector=None, **kwargs_)
```

baseConductanceScalingCaDependent

extends `baseConductanceScaling`

Base ComponentType for a scaling to apply to a gate's conductance which depends on Ca concentration. Usually a generic expression of `caConc` (so no standard, non-base form here).

Exposures

factor	(from baseConductanceScaling)	Dimensionless
---------------	--	---------------

Requirements

caConc		<i>concentration</i>
temperature	(from baseConductanceScaling)	<i>temperature</i>

baseGate

Base ComponentType for a voltage and/or concentration dependent gate.

Parameters

in- stances		Dimensionless
------------------------	--	---------------

Child list

notes		<i>notes</i>
--------------	--	--------------

Exposures

fcond		Dimensionless
q		Dimensionless

gate

extends [baseGate](#)

Conveniently named baseGate.

Parameters

in- stances	(from baseGate)	Dimensionless
------------------------	----------------------------------	---------------

Exposures

fcond	(from baseGate)	Dimensionless
q	(from baseGate)	Dimensionless

gateHHrates

extends [gate](#)

Gate which follows the general Hodgkin Huxley formalism.

Parameters

in- stances	(from baseGate)	Dimensionless
------------------------	-----------------	---------------

Child list

for- wardRate		<i>baseVoltageDepRate</i>
re- verseR- ate		<i>baseVoltageDepRate</i>

Children list

q10Settings	<i>baseQ10Settings</i>
--------------------	------------------------

Exposures

alpha		<i>per_time</i>
beta		<i>per_time</i>
fcond	(from baseGate)	Dimensionless
inf		Dimensionless
q	(from baseGate)	Dimensionless
rateScale		Dimensionless
tau		<i>time</i>

Dynamics

State Variables **q**: Dimensionless (exposed as **q**)

On Start **q** = inf

Derived Variables **rateScale** = q10Settings[*]->q10(reduce method: multiply) (exposed as **rateScale**)

alpha = forwardRate->r (exposed as **alpha**)

beta = reverseRate->r (exposed as **beta**)

fcond = **q**^{instances} (exposed as **fcond**)

inf = alpha/(alpha+beta) (exposed as **inf**)

tau = 1/((alpha+beta) * rateScale) (exposed as **tau**)

Time Derivatives d **q** /dt = (**inf** - **q**) / **tau**

Usage: XML

```
<gateHHrates id="m" instances="3">
    <forwardRate type="HHExpLinearRate" rate="1per_ms" midpoint="-40mV" scale=
    ↵ "10mV"/>
    <reverseRate type="HHExpRate" rate="4per_ms" midpoint="-65mV" scale="-18mV
    ↵ "/>
</gateHHrates>
```

```
<gateHHrates id="h" instances="1">
    <forwardRate type="HHExpRate" rate="0.07per_ms" midpoint="-65mV" scale="-
    ↵ 20mV"/>
    <reverseRate type="HHSigmoidRate" rate="1per_ms" midpoint="-35mV" scale=
    ↵ "10mV"/>
</gateHHrates>
```

```
<gateHHrates id="m" instances="3">
    <forwardRate type="HHExpLinearRate" rate="1per_ms" midpoint="-40mV" scale=
    ↵ "10mV"/>
    <reverseRate type="HHExpRate" rate="4per_ms" midpoint="-65mV" scale="-18mV
    ↵ "/>
</gateHHrates>
```

gateHHtaulnF

extends *gate*

Gate which follows the general Hodgkin Huxley formalism.

Parameters

instances	(from baseGate)	Dimensionless
------------------	-----------------	---------------

Child list

time-Course		<i>baseVoltageDep-Time</i>
steadyState		<i>baseVoltageDep-Variable</i>

Children list

q10Settings	<i>baseQ10Settings</i>
--------------------	------------------------

Exposures

fcond	(from baseGate)	Dimensionless
inf		Dimensionless
q	(from baseGate)	Dimensionless
rateScale		Dimensionless
tau		<i>time</i>

Dynamics

State Variables **q**: Dimensionless (exposed as **q**)

On Start **q** = **inf**

Derived Variables **rateScale** = **q10Settings**[*]->**q10**(reduce method: multiply) (exposed as **rateScale**)

fcond = **q**^{instances} (exposed as **fcond**)

inf = **steadyState->x** (exposed as **inf**)

tauUnscaled = **timeCourse->t**

tau = **tauUnscaled** / **rateScale** (exposed as **tau**)

Time Derivatives $d\mathbf{q}/dt = (\mathbf{inf} - \mathbf{q}) / \mathbf{tau}$

gateHHInstantaneousextends [gate](#)

Gate which follows the general Hodgkin Huxley formalism but is instantaneous, so tau = 0 and gate follows exactly inf value.

Parameters

in- stances	(from baseGate)	Dimensionless
------------------------	----------------------------------	---------------

Child list

steadyS- tate		<i>baseVoltageDep- Variable</i>
--------------------------	--	-------------------------------------

Constants

SEC = 1 s		<i>time</i>
------------------	--	-------------

Exposures

fcond	(from baseGate)	Dimensionless
inf		Dimensionless
q	(from baseGate)	Dimensionless
tau		<i>time</i>

Dynamics

Derived Variables **inf** = steadyState->x (exposed as **inf**)

tau = 0 * SEC (exposed as **tau**)

q = inf (exposed as **q**)

fcond = q^instances (exposed as **fcond**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GateHHInstantaneous

variable = GateHHInstantaneous(neuro_lex_id=None, id=None, instances=None, notes=None,
                                ↵ steady_state=None, gds_collector_=None, **kwargs_)
```

gateHHratesTau

extends [gate](#)

Gate which follows the general Hodgkin Huxley formalism.

Parameters

instances	(from baseGate)	Dimensionless
-----------	-----------------	---------------

Child list

forwardRate		baseVoltageDepRate
reverseRate		baseVoltageDepRate
timeCourse		baseVoltageDepTime

Children list

q10Settings	baseQ10Settings
-------------	-----------------

Exposures

alpha		per_time
beta		per_time
fcond	(from baseGate)	Dimensionless
inf		Dimensionless
q	(from baseGate)	Dimensionless
rateScale		Dimensionless
tau		time

Dynamics

State Variables **q**: Dimensionless (exposed as **q**)

On Start **q** = inf

Derived Variables **rateScale** = q10Settings[*]->q10(reduce method: multiply) (exposed as **rateScale**)

alpha = forwardRate->r (exposed as **alpha**)

beta = reverseRate->r (exposed as **beta**)

fcond = q^instances (exposed as **fcond**)

inf = alpha/(alpha+beta) (exposed as **inf**)

tauUnscaled = timeCourse->t

tau = tauUnscaled / rateScale (exposed as **tau**)

Time Derivatives d **q** /dt = (inf - q) / tau

gateHHratesInf

extends *gate*

Gate which follows the general Hodgkin Huxley formalism.

Parameters

in- stances	(from <i>baseGate</i>)	Dimensionless
------------------------	-------------------------	---------------

Child list

for- wardRate		<i>baseVoltageDepRate</i>
re- verseR- ate		<i>baseVoltageDepRate</i>
steadyS- tate		<i>baseVoltageDep- Variable</i>

Children list

q10Settings	<i>baseQ10Settings</i>
--------------------	------------------------

Exposures

alpha		<i>per_time</i>
beta		<i>per_time</i>
fcond	(from baseGate)	Dimensionless
inf		Dimensionless
q	(from baseGate)	Dimensionless
rateScale		Dimensionless
tau		<i>time</i>

Dynamics

State Variables **q**: Dimensionless (exposed as **q**)

On Start **q** = inf

Derived Variables **rateScale** = q10Settings[*]->q10(reduce method: multiply) (exposed as **rateScale**)

alpha = forwardRate->r (exposed as **alpha**)

beta = reverseRate->r (exposed as **beta**)

fcond = q^instances (exposed as **fcond**)

inf = steadyState->x (exposed as **inf**)

tau = 1/((alpha+beta) * rateScale) (exposed as **tau**)

Time Derivatives d **q** /dt = (inf - q) / tau

gateHHratesTauInf

extends [gate](#)

Gate which follows the general Hodgkin Huxley formalism.

Parameters

in- stances	(from baseGate)	Dimensionless
------------------------	-----------------	---------------

Child list

for- wardRate		<i>baseVoltageDepRate</i>
re- verseR- ate		<i>baseVoltageDepRate</i>
time- Course		<i>baseVoltageDep- Time</i>
steadyS- tate		<i>baseVoltageDep- Variable</i>

Children list

q10Settings	<i>baseQ10Settings</i>
--------------------	------------------------

Exposures

alpha		<i>per_time</i>
beta		<i>per_time</i>
fcond	(from <i>baseGate</i>)	Dimensionless
inf		Dimensionless
q	(from <i>baseGate</i>)	Dimensionless
rateScale		Dimensionless
tau		<i>time</i>

Dynamics

State Variables **q**: Dimensionless (exposed as **q**)

On Start **q** = inf

Derived Variables **rateScale** = q10Settings[*]->q10(reduce method: multiply) (exposed as **rateScale**)

alpha = forwardRate->r (exposed as **alpha**)

beta = reverseRate->r (exposed as **beta**)

inf = steadyState->x (exposed as **inf**)

tauUnscaled = timeCourse->

tau = tauUnscaled / rateScale (exposed as **tau**)

fcond = q^instances (exposed as **fcond**)

Time Derivatives d **q** /dt = (inf - q) / tau

gateFractional

extends *gate*

Gate composed of subgates contributing with fractional conductance.

Parameters

in- stances	(from <i>baseGate</i>)	Dimensionless
------------------------	-------------------------	---------------

Children list

q10Settings		<i>baseQ10Settings</i>
sub-Gate		<i>subGate</i>

Exposures

fcond	(from <i>baseGate</i>)	Dimensionless
q	(from <i>baseGate</i>)	Dimensionless
rateScale		Dimensionless

Dynamics

Derived Variables **q** = *subGate*[*]->qfrac(reduce method: add) (exposed as **q**)

fcond = *q*^{instances} (exposed as **fcond**)

rateScale = *q10Settings*[*]->*q10*(reduce method: multiply) (exposed as **rateScale**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GateFractional

variable = GateFractional(neuro_lex_id=None, id=None, instances=None, notes=None, q10_
    ↪settings=None, sub_gates=None, gds_collector_=None, **kwargs_)
```

subGate

Gate composed of subgates contributing with fractional conductance.

Parameters

fractional-Conductance		Dimensionless
-------------------------------	--	---------------

Child list

notes		<i>notes</i>
time- Course		<i>baseVoltageDep- Time</i>
steadyS- tate		<i>baseVoltageDep- Variable</i>

Exposures

inf		Dimensionless
q		Dimensionless
qfrac		Dimensionless
tau		<i>time</i>

Requirements

rateScale	Dimensionless
------------------	---------------

Dynamics

State Variables **q**: Dimensionless (exposed as **q**)

On Start **q** = inf

Derived Variables **inf** = steadyState->x (exposed as **inf**)

tauUnscaled = timeCourse->t

tau = tauUnscaled / rateScale (exposed as **tau**)

qfrac = q * fractionalConductance (exposed as **qfrac**)

Time Derivatives d **q** /dt = (inf - q) / tau

baseIonChannel

Base for all ion channel ComponentTypes.

Parameters

conduc- tance		<i>conductance</i>
--------------------------	--	--------------------

Text fields

<code>neu-</code> <code>roLexId</code>	
---	--

Child list

<code>notes</code>		<i>notes</i>
<code>annotation</code>		<i>annotation</i>

Exposures

<code>fopen</code>		Dimensionless
<code>g</code>		<i>conductance</i>

Requirements

<code>v</code>		<i>voltage</i>
----------------	--	----------------

ionChannelPassive

extends *ionChannel*

Simple passive ion channel where the constant conductance through the channel is equal to **conductance**.

Parameters

<code>conductance</code>	(from <code>baseIonChannel</code>)	<i>conductance</i>
--------------------------	-------------------------------------	--------------------

Text fields

<code>species</code>	
----------------------	--

Exposures

fopen	(from baseIonChannel)	Dimensionless
g	(from baseIonChannel)	<i>conductance</i>

Requirements

v	(from baseIonChannel)	<i>voltage</i>
----------	-----------------------	----------------

Dynamics

Derived Variables **fopen** = 1 (exposed as **fopen**)

g = conductance (exposed as **g**)

ionChannelHH

extends [baseIonChannel](#)

Note [ionChannel](#) and [ionChannelHH](#) are currently functionally identical. This is needed since many existing examples use ionChannel, some use ionChannelHH. NeuroML v2beta4 should remove one of these, probably ionChannelHH.

Parameters

conductance	(from baseIonChannel)	<i>conductance</i>
--------------------	-----------------------	--------------------

Text fields

species	
----------------	--

Children list

conductanceScaling		<i>baseConductanceScaling</i>
gates		<i>gate</i>

Exposures

fopen	(from baseIonChannel)	Dimensionless
g	(from baseIonChannel)	<i>conductance</i>

Requirements

v	(from baseIonChannel)	<i>voltage</i>
----------	-----------------------	----------------

Dynamics

Derived Variables **conductanceScale** = conductanceScaling[*]->factor(reduce method: multiply)

```
fopen0 = gates[*]->fcond(reduce method: multiply)
fopen = conductanceScale * fopen0  (exposed as fopen)
g = conductance * fopen  (exposed as g)
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IonChannelHH

variable = 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, gds_collector_=None,
    **kwargs_)
```

Usage: XML

```
<ionChannelHH id="pas" conductance="10pS"/>
```

```
<ionChannelHH id="HH_Na" conductance="10pS" species="na">
</ionChannelHH>
```

```
<ionChannelHH id="NaConductance" conductance="10pS" species="na">
    <gateHHrates id="m" instances="3">
        <forwardRate type="HHExpLinearRate" rate="1per_ms" midpoint="-40mV" scale=
        "10mV"/>
        <reverseRate type="HHExpRate" rate="4per_ms" midpoint="-65mV" scale="-18mV
        "/>
    </gateHHrates>
    <gateHHrates id="h" instances="1">
        <forwardRate type="HHExpRate" rate="0.07per_ms" midpoint="-65mV" scale="-
        20mV"/>
```

(continues on next page)

(continued from previous page)

```

<reverseRate type="HHSigmoidRate" rate="1per_ms" midpoint="-35mV" scale=
↪ "10mV" />
  </gateHHrates>
</ionChannelHH>

```

ionChannel

extends *ionChannelHH*

Note *ionChannel* and *ionChannelHH* are currently functionally identical. This is needed since many existing examples use *ionChannel*, some use *ionChannelHH*. NeuroML v2beta4 should remove one of these, probably *ionChannelHH*.

Parameters

conductance	(from <i>baseIonChannel</i>)	<i>conductance</i>
--------------------	-------------------------------	--------------------

Exposures

fopen	(from <i>baseIonChannel</i>)	Dimensionless
g	(from <i>baseIonChannel</i>)	<i>conductance</i>

Requirements

v	(from <i>baseIonChannel</i>)	<i>voltage</i>
----------	-------------------------------	----------------

Dynamics

Derived Variables **conductanceScale** = *conductanceScaling*[*]->*factor*(reduce method: multiply)

fopen0 = *gates*[*]->*fcond*(reduce method: multiply)

fopen = *conductanceScale* * **fopen0** (exposed as **fopen**)

g = *conductance* * **fopen** (exposed as **g**)

Usage: Python

Go to the libNeuroML documentation

```

from neuroml import IonChannel

variable = 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, extenstiontype_=None,
↪ gds_collector_=None, **kwargs_)

```

(continues on next page)

(continued from previous page)

ionChannelVShift

extends *ionChannel*

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.

Parameters

conductance	(from <i>baseIonChannel</i>)	<i>conductance</i>
vShift		<i>voltage</i>

Text fields

species	
----------------	--

Exposures

fopen	(from <i>baseIonChannel</i>)	Dimensionless
g	(from <i>baseIonChannel</i>)	<i>conductance</i>

Requirements

v	(from <i>baseIonChannel</i>)	<i>voltage</i>
----------	-------------------------------	----------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IonChannelVShift

variable = 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_
                             _tau=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, gds_
                             _collector_=None, **kwargs_)
```

KSState

One of the states in which a *gateKS* can be. The rates of transitions between these states are given by *KSTransitions*.

Parameters

relativeConductance		Dimensionless
---------------------	--	---------------

Exposures

occupancy		Dimensionless
q		Dimensionless

Dynamics

State Variables **occupancy**: Dimensionless (exposed as **occupancy**)

Derived Variables **q** = relativeConductance * occupancy (exposed as **q**)

closedState

extends *KSState*

A *KSState* with **relativeConductance** of 0.

Parameters

relativeConductance	(from <i>KSState</i>)	Dimensionless
---------------------	------------------------	---------------

Exposures

occupancy	(from KSState)	Dimensionless
q	(from KSState)	Dimensionless

Usage: Python

Go to the [libNeuroML documentation](#)

```
from neuroml import ClosedState

variable = ClosedState(neuro_lex_id=None, id=None, gds_collector_=None, **kwargs_)
```

openState

extends [KSState](#)

A [KSState](#) with **relativeConductance** of 1.

Parameters

relative-Conductance	(from KSState)	Dimensionless
-----------------------------	---------------------------------	---------------

Exposures

occupancy	(from KSState)	Dimensionless
q	(from KSState)	Dimensionless

Usage: Python

Go to the [libNeuroML documentation](#)

```
from neuroml import OpenState

variable = OpenState(neuro_lex_id=None, id=None, gds_collector_=None, **kwargs_)
```

ionChannelKS

extends [baseIonChannel](#)

A kinetic scheme based ion channel with multiple *gateKSs*, each of which consists of multiple *KSStates* and *KSTransitions* giving the rates of transition between them.

Parameters

conductance	(from baseIonChannel)	<i>conductance</i>
--------------------	--	--------------------

Text fields

species	
----------------	--

Children list

conductanceScaling		<i>baseConductanceScaling</i>
gates		<i>gateKS</i>

Exposures

fopen	(from baseIonChannel)	Dimensionless
g	(from baseIonChannel)	<i>conductance</i>

Requirements

v	(from baseIonChannel)	<i>voltage</i>
----------	--	----------------

Dynamics

Derived Variables **fopen** = gates[*]->fcond(reduce method: multiply) (exposed as **fopen**)

g = fopen * conductance (exposed as **g**)

Usage: Python

Go to the [libNeuroML documentation](#)

```
from neuroml import IonChannelKS

variable = IonChannelKS(neuro_lex_id=None, id=None, metaid=None, notes=None,_
    ↪properties=None, annotation=None, species=None, conductance=None, gate_kses=None,_
    ↪gds_collector=None, **kwargs_)
```

KSTransition

Specified the forward and reverse rates of transition between two [KSSStates](#) in a [gateKS](#).

Exposures

rf		<i>per_time</i>
rr		<i>per_time</i>

forwardTransition

extends [KSTransition](#)

A forward only [KSTransition](#) for a [gateKS](#) which specifies a **rate** (type [baseHHRate](#)) which follows one of the standard Hodgkin Huxley forms (e.g. [HHExpRate](#), [HHSigmoidRate](#), [HHExpLinearRate](#).

Child list

rate		<i>baseHHRate</i>
-------------	--	-------------------

Constants

SEC = 1s		<i>time</i>
-----------------	--	-------------

Exposures

rf	(from KSTransition)	<i>per_time</i>
rr	(from KSTransition)	<i>per_time</i>

Dynamics

Derived Variables **rf0** = rate->r

rf = rf0 (exposed as **rf**)

rr = 0/SEC (exposed as **rr**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ForwardTransition

variable = ForwardTransition(neuro_lex_id=None, id=None, from_=None, to=None, _  
    ↪anytypeobjs_=None, gds_collector_=None, **kwargs_)
```

reverseTransition

extends *KSTransition*

A reverse only *KSTransition* for a *gateKS* which specifies a **rate** (type *baseHHRate*) which follows one of the standard Hodgkin Huxley forms (e.g. *HHExpRate*, *HHSigmoidRate*, *HHExpLinearRate*.

Child list

rate		<i>baseHHRate</i>
-------------	--	-------------------

Constants

SEC = 1s		<i>time</i>
-----------------	--	-------------

Exposures

rf	(from <i>KSTransition</i>)	<i>per_time</i>
rr	(from <i>KSTransition</i>)	<i>per_time</i>

Dynamics

Derived Variables **rr0** = rate->r

rr = rr0 (exposed as **rr**)

rf = 0/SEC (exposed as **rf**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ReverseTransition

variable = ReverseTransition(neuro_lex_id=None, id=None, from_=None, to=None, _  
    ↪anytypeobjs_=None, gds_collector_=None, **kwargs_)
```

vHalfTransition

extends *KSTransition*

Transition which specifies both the forward and reverse rates of transition.

Parameters

gamma		Dimensionless
tau		<i>time</i>
tauMin		<i>time</i>
vHalf		<i>voltage</i>
z		Dimensionless

Constants

kte = 25.3mV		<i>voltage</i>
---------------------	--	----------------

Exposures

rf	(from <i>KSTransition</i>)	<i>per_time</i>
rr	(from <i>KSTransition</i>)	<i>per_time</i>

Requirements

v		<i>voltage</i>
----------	--	----------------

Dynamics

Derived Variables **rf0** = $\exp(z * \text{gamma} * (v - vHalf) / kte) / \tau$

rr0 = $\exp(-z * (1 - \text{gamma}) * (v - vHalf) / kte) / \tau$

rf = $1 / (1/rf0 + \tau_{\min})$ (exposed as **rf**)

rr = $1 / (1/rr0 + \tau_{\min})$ (exposed as **rr**)

tauInfTransition

extends *KSTransition*

KS Transition specified in terms of time constant schema:tau and steady state schema:inf.

Child list

time-Course		<i>baseVoltageDep-Time</i>
steadyState		<i>baseVoltageDep-Variable</i>

Exposures

rf	(from <i>KSTransition</i>)	<i>per_time</i>
rr	(from <i>KSTransition</i>)	<i>per_time</i>

Dynamics

Derived Variables **tau** = timeCourse->t

inf = steadyState->x

rf = inf/tau (exposed as **rf**)

rr = (1-inf)/tau (exposed as **rr**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import TauInfTransition

variable = TauInfTransition(neuro_lex_id=None, id=None, from_=None, to=None, steady_
                             _state=None, time_course=None, gds_collector_=None, **kwargs_)
```

gateKS

extends [baseGate](#)

A gate which consists of multiple [KSStates](#) and [KSTransitions](#) giving the rates of transition between them.

Parameters

in- stances	(from baseGate)	Dimensionless
------------------------	----------------------------------	---------------

Children list

states		KSState
transi- tions		KSTransition
q10Settings		baseQ10Settings

Exposures

fcond	(from baseGate)	Dimensionless
q	(from baseGate)	Dimensionless
rateScale		Dimensionless

Dynamics

Derived Variables **rateScale** = q10Settings[*]->q10(reduce method: multiply) (exposed as **rateScale**)

q = states[*]->q(reduce method: add) (exposed as **q**)

fcond = q^instances (exposed as **fcond**)

Usage: Python

Go to the [libNeuroML documentation](#)

```
from neuroml import GateKS

variable = 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, gds_collector_=None, **kwargs_)
```

10.1.6 Synapses

A number of synaptic ComponentTypes for use in NeuroML 2 documents, e.g. *expOneSynapse*, *expTwoSynapse*, *blockingPlasticSynapse*. These extend the *baseSynapse* ComponentType. Also defined continuously transmitting synapses, e.g. *gapJunction* and *gradedSynapse*.

Original ComponentType definitions: [Synapses.xml](#). Schema against which NeuroML based on these should be valid: [NeuroML_v2.2.xsd](#). Generated on 07/06/22 from [this commit](#). Please file any issues or questions at the [issue tracker here](#).

baseSynapse

extends [basePointCurrent](#)

Base type for all synapses, i.e. ComponentTypes which produce a current (dimension current) and change Dynamics in response to an incoming event.

Bioportal entry for Computational Neuroscience Ontology related to [baseSynapse](#).

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	current
----------	--	-------------------------

Event Ports

in	Direction: in
-----------	---------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BaseSynapse

variable = BaseSynapse(neuro_lex_id=None, id=None, metaid=None, notes=None,_
    properties=None, annotation=None, extensiontype_=None, gds_collector_=None,_
    **kwargs_)
```

baseVoltageDepSynapseextends [baseSynapse](#)

Base type for synapses with a dependence on membrane potential.

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	current
----------	--	-------------------------

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed	voltage
----------	--	-------------------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
-----------	-----------------------------	---------------

Usage: PythonGo to the [libNeuroML documentation](#)

```
from neuroml import BaseVoltageDepSynapse

variable = BaseVoltageDepSynapse(neuro_lex_id=None, id=None, metaid=None, notes=None, ↴
properties=None, annotation=None, extensiontype_=None, gds_collector_=None, ↴
**kwargs_)
```

baseSynapseDLextends [baseVoltageDepPointCurrentDL](#)

Base type for all synapses, i.e. ComponentTypes which produce a dimensionless current and change Dynamics in response to an incoming event.

Biportal entry for Computational Neuroscience Ontology related to [baseSynapseDL](#).

Exposures

I	The total (time varying) current produced by this ComponentType (<i>from basePointCurrentDL</i>)	Dimensionless
---	--	---------------

Requirements

V	The current may vary with the dimensionless voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepPointCurrentDL</i>)	Dimensionless
---	---	---------------

baseCurrentBasedSynapse

extends [baseSynapse](#)

Synapse model which produces a synaptic current.

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
---	--	----------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
----	-----------------------------	---------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BaseCurrentBasedSynapse

variable = BaseCurrentBasedSynapse(neuro_lex_id=None, id=None, metaid=None, _notes=None, properties=None, annotation=None, extensiontype_=None, gds_collector_=None, **kwargs_)
```

alphaCurrentSynapse

extends [baseCurrentBasedSynapse](#)

Alpha current synapse: rise time and decay time are both **tau..**

Parameters

ibase	Baseline current increase after receiving a spike	<i>current</i>
tau	Time course for rise and decay	<i>time</i>

Properties

weight (default: 1)	Dimensionless
----------------------------	---------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
----------	--	----------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
-----------	-----------------------------	---------------

Dynamics

State Variables **I**: *current*

J: *current*

On Start **I** = 0

J = 0

On Events EVENT IN on port: **in**

J = **J** + weight * ibase

Derived Variables **i** = **I** (exposed as **i**)

Time Derivatives **d I /dt** = (2.7182818284590451***J** - **I**)/tau

d J /dt = -**J**/tau

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import AlphaCurrentSynapse

variable = AlphaCurrentSynapse(neuro_lex_id=None, id=None, metaid=None, notes=None, ↪
    properties=None, annotation=None, tau=None, ibase=None, gds_collector_=None, ↪
    **kwargs_)
```

baseConductanceBasedSynapse

extends [baseVoltageDepSynapse](#)

Synapse model which exposes a conductance **g** in addition to producing a current. Not necessarily ohmic!!

Bioportal entry for Computational Neuroscience Ontology related to [baseConductanceBasedSynapse](#).

Parameters

erev	Reversal potential of the synapse	<i>voltage</i>
gbase	Baseline conductance, generally the maximum conductance following a single spike	<i>conductance</i>

Exposures

g	Time varying conductance through the synapse	<i>conductance</i>
i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepSynapse</i>)	<i>voltage</i>
----------	--	----------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
-----------	-----------------------------	---------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BaseConductanceBasedSynapse

variable = BaseConductanceBasedSynapse(neuro_lex_id=None, id=None, metaid=None,
notes=None, properties=None, annotation=None, gbase=None, erev=None, extensiontype_
=None, gds_collector_=None, **kwargs_)
```

baseConductanceBasedSynapseTwo

extends [baseVoltageDepSynapse](#)

Synapse model suited for a sum of two expTwoSynapses which exposes a conductance **g** in addition to producing a current. Not necessarily ohmic!!

Biportal entry for Computational Neuroscience Ontology related to [baseConductanceBasedSynapseTwo](#).

Parameters

erev	Reversal potential of the synapse	<i>voltage</i>
gbase1	Baseline conductance 1	<i>conductance</i>
gbase2	Baseline conductance 2	<i>conductance</i>

Exposures

g	Time varying conductance through the synapse	<i>conductance</i>
i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepSynapse</i>)	<i>voltage</i>
----------	--	----------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
-----------	-----------------------------	---------------

Usage: Python

Go to the [libNeuroML documentation](#)

```
from neuroml import BaseConductanceBasedSynapseTwo

variable = BaseConductanceBasedSynapseTwo(neuro_lex_id=None, id=None, metaid=None,
                                          notes=None, properties=None, annotation=None, gbase1=None, gbase2=None, erev=None,
                                          extensiontype_=None, gds_collector_=None, **kwargs_)
```

expOneSynapseextends [baseConductanceBasedSynapse](#)

Ohmic synapse model whose conductance rises instantaneously by (**gbase * weight**) on receiving an event, and which decays exponentially to zero with time course **tauDecay**.

Parameters

erev	Reversal potential of the synapse (<i>from baseConductanceBasedSynapse</i>)	<i>voltage</i>
gbase	Baseline conductance, generally the maximum conductance following a single spike (<i>from baseConductanceBasedSynapse</i>)	<i>conductance</i>
tauDecay	Time course of decay	<i>time</i>

Properties

weight (default: 1)		Dimensionless
----------------------------	--	---------------

Exposures

g	Time varying conductance through the synapse (<i>from baseConductanceBasedSynapse</i>)	<i>conductance</i>
i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepSynapse</i>)	<i>voltage</i>
----------	--	----------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
-----------	-----------------------------	---------------

Dynamics

State Variables **g**: *conductance* (exposed as **g**)

On Start $g = 0$

On Events EVENT IN on port: **in**

$$g = g + (\text{weight} * \text{gbase})$$

Derived Variables $i = g * (\text{erev} - v)$ (exposed as **i**)

Time Derivatives $d g /dt = -g / \tau_{\text{decay}}$

Usage: Python

Go to the [libNeuroML documentation](#)

```
from neuroml import ExpOneSynapse

variable = ExpOneSynapse(neuro_lex_id=None, id=None, metaid=None, notes=None, ↵
    ↵properties=None, annotation=None, gbase=None, erev=None, tau_decay=None, gds_ ↵
    ↵collector=None, **kwargs_)
```

Usage: XML

```
<expOneSynapse id="syn1" gbase="5nS" erev="0mV" tauDecay="3ms"/>
```

```
<expOneSynapse id="syn2" gbase="10nS" erev="0mV" tauDecay="2ms"/>
```

```
<expOneSynapse id="syn1" gbase="5nS" erev="0mV" tauDecay="3ms"/>
```

alphaSynapse

extends [baseConductanceBasedSynapse](#)

Ohmic synapse model where rise time and decay time are both **tau**. Max conductance reached during this time (assuming zero conductance before) is **gbase * weight..**

Parameters

erev	Reversal potential of the synapse (<i>from baseConductanceBasedSynapse</i>)	<i>voltage</i>
gbase	Baseline conductance, generally the maximum conductance following a single spike (<i>from baseConductanceBasedSynapse</i>)	<i>conductance</i>
tau	Time course of rise/decay	<i>time</i>

Properties

weight (default: 1)		Dimensionless
----------------------------	--	---------------

Exposures

g	Time varying conductance through the synapse (<i>from</i> baseConductanceBasedSynapse)	<i>conductance</i>
i	The total (usually time varying) current produced by this ComponentType (<i>from</i> basePointCurrent)	<i>current</i>

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from</i> baseVoltageDepSynapse)	<i>voltage</i>
----------	---	----------------

Event Ports

in	(<i>from</i> baseSynapse)	Direction: in
-----------	----------------------------	---------------

Dynamics

State Variables **g**: *conductance* (exposed as **g**)

A: *conductance*

On Start g = 0

A = 0

On Events EVENT IN on port: **in**

A = A + (gbase*weight)

Derived Variables i = g * (erev - v) (exposed as **i**)

Time Derivatives d g /dt = (2.7182818284590451 * A - g)/tau

d A /dt = -A / tau

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import AlphaSynapse

variable = AlphaSynapse(neuro_lex_id=None, id=None, metaid=None, notes=None, properties=None, annotation=None, gbase=None, erev=None, tau=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<alphaSynapse id="synalpha" gbase="0.5nS" erev="0mV" tau="2ms">
    <notes>An alpha synapse with time for rise equal to decay.</notes>
</alphaSynapse>
```

expTwoSynapse

extends `baseConductanceBasedSynapse`

Ohmic synapse model whose conductance waveform on receiving an event has a rise time of `tauRise` and a decay time of `tauDecay`. Max conductance reached during this time (assuming zero conductance before) is `gbase * weight..`

Parameters

<code>erev</code>	Reversal potential of the synapse (<i>from baseConductanceBasedSynapse</i>)	<i>voltage</i>
<code>gbase</code>	Baseline conductance, generally the maximum conductance following a single spike (<i>from baseConductanceBasedSynapse</i>)	<i>conductance</i>
<code>tauDecay</code>		<i>time</i>
<code>tauRise</code>		<i>time</i>

Derived parameters

<code>peak-Time</code>		<i>time</i>
<code>wave-form-Factor</code>		Dimensionless

Properties

<code>weight</code> (default: 1)		Dimensionless
----------------------------------	--	---------------

Exposures

<code>g</code>	Time varying conductance through the synapse (<i>from baseConductanceBasedSynapse</i>)	<i>conductance</i>
<code>i</code>	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepSynapse</i>)	<i>voltage</i>
----------	--	----------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
-----------	-----------------------------	---------------

Dynamics

State Variables **A**: Dimensionless

B: Dimensionless

On Start **A** = 0

B = 0

On Events EVENT IN on port: **in**

A = **A** + (weight * waveformFactor)

B = **B** + (weight * waveformFactor)

Derived Variables **g** = **gbase** * (**B** - **A**) (exposed as **g**)

i = **g** * (erev - **v**) (exposed as **i**)

Time Derivatives $d\mathbf{A}/dt = -\mathbf{A} / \tau_{\text{rise}}$

$d\mathbf{B}/dt = -\mathbf{B} / \tau_{\text{decay}}$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ExpTwoSynapse

variable = 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, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<expTwoSynapse id="AMPA" gbase="0.5nS" erev="0mV" tauRise="1ms" tauDecay="2ms"/>
```

```
<expTwoSynapse id="synInput" gbase="8nS" erev="20mV" tauRise="1ms" tauDecay="5ms"/>
```

```
<expTwoSynapse id="synInputFast" gbase="1nS" erev="20mV" tauRise="0.2ms" tauDecay="1ms"
                />
```

expThreeSynapse

extends [baseConductanceBasedSynapseTwo](#)

Ohmic synapse similar to [expTwoSynapse](#) but consisting of two components that can differ in decay times and max conductances but share the same rise time.

Parameters

erev	Reversal potential of the synapse (<i>from baseConductanceBasedSynapseTwo</i>)	<i>voltage</i>
gbase1	Baseline conductance 1 (<i>from baseConductanceBasedSynapseTwo</i>)	<i>conductance</i>
gbase2	Baseline conductance 2 (<i>from baseConductanceBasedSynapseTwo</i>)	<i>conductance</i>
tauDecay1		<i>time</i>
tauDecay2		<i>time</i>
tauRise		<i>time</i>

Derived parameters

peak-Time1		<i>time</i>
peak-Time2		<i>time</i>
wave-form-Factor1		Dimensionless
wave-form-Factor2		Dimensionless

Properties

weight (default: 1)		Dimensionless
----------------------------	--	---------------

Exposures

g	Time varying conductance through the synapse (<i>from baseConductanceBasedSynapseTwo</i>)	<i>conductance</i>
i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepSynapse</i>)	<i>voltage</i>
----------	--	----------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
-----------	-----------------------------	---------------

Dynamics

State Variables **A**: Dimensionless

B: Dimensionless

C: Dimensionless

On Start **A** = 0

B = 0

C = 0

On Events EVENT IN on port: **in**

A = **A** + (gbase1weight * waveformFactor1 + gbase2weight*waveformFactor2)/(gbase1+gbase2)

B = **B** + (weight * waveformFactor1)

C = **C** + (weight * waveformFactor2)

Derived Variables **g** = gbase1*(B - A) + gbase2*(C-A) (exposed as **g**)

i = **g** * (erev - **v**) (exposed as **i**)

Time Derivatives $d\mathbf{A}/dt = -A/\tau_{rise}$

$d\mathbf{B}/dt = -B/\tau_{decay1}$

$d\mathbf{C}/dt = -C/\tau_{decay2}$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ExpThreeSynapse

variable = 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, gds_collector=None, **kwargs)
```

Usage: XML

```
<expThreeSynapse id="synInputFastTwo" gbase1="1.5nS" tauRise="0.1ms" tauDecay1="0.7ms"
    ↵" gbase2="0.5nS" tauDecay2="2.5ms" erev="0mV"/>
```

```
<expThreeSynapse id="AMPA" gbase1="1.5nS" tauRise="0.1ms" tauDecay1="0.7ms" gbase2="0.
    ↵5nS" tauDecay2="2.5ms" erev="0mV">
    <notes>A synapse consisting of one rise and two decay time courses.</notes>
</expThreeSynapse>
```

baseBlockMechanism

Base of any ComponentType which produces a varying scaling (or blockage) of synaptic strength of magnitude **scaling**.

Exposures

block-Factor		Dimensionless
--------------	--	---------------

voltageConcDepBlockMechanism

extends [baseBlockMechanism](#)

Synaptic blocking mechanism which varies with membrane potential across the synapse, e.g. in NMDA receptor mediated synapses.

Parameters

block-Concentration		<i>concentration</i>
scaling-Conc		<i>concentration</i>
scaling-Volt		<i>voltage</i>

Text fields

species	
---------	--

Exposures

block-Factor	(from baseBlockMechanism)	Dimensionless
---------------------	---------------------------	---------------

Requirements

v	voltage
---	---------

Dynamics

Derived Variables `blockFactor = 1/(1 + (blockConcentration / scalingConc)* exp(-1 * (v / scalingVolt)))` (exposed as `blockFactor`)

basePlasticityMechanism

Base plasticity mechanism.

Exposures

plasticityFactor		Dimensionless
-------------------------	--	---------------

Event Ports

in	This is where the plasticity mechanism receives spike events from the parent synapse.	Direction: in
-----------	---	---------------

tsodyksMarkramDepMechanism

extends `basePlasticityMechanism`

Depression-only Tsodyks-Markram model, as in Tsodyks and Markram 1997.

Parameters

initReleaseProb		Dimensionless
tauRec		<i>time</i>

Exposures

plasticityFactor	(from basePlasticityMechanism)	Dimensionless
-------------------------	--------------------------------	---------------

Event Ports

in	This is where the plasticity mechanism receives spike events from the parent synapse. (from basePlasticityMechanism)	Direction: in
-----------	--	---------------

Dynamics

Structure WITH parent AS **a**

WITH this AS **b**

EVENT CONNECTION from **a** TO **b**

State Variables **R**: Dimensionless

On Start **R** = 1

On Events EVENT IN on port: **in**

$$\mathbf{R} = \mathbf{R} * (1 - \mathbf{U})$$

Derived Variables **U** = initReleaseProb

$$\text{plasticityFactor} = \mathbf{R} * \mathbf{U} \quad (\text{exposed as } \text{plasticityFactor})$$

Time Derivatives $d \mathbf{R} / dt = (1 - \mathbf{R}) / \tau_{\text{Rec}}$

tsodyksMarkramDepFacMechanism

extends [basePlasticityMechanism](#)

Full Tsodyks-Markram STP model with both depression and facilitation, as in Tsodyks, Pawelzik and Markram 1998.

Parameters

initReleaseProb		Dimensionless
tauFac		<i>time</i>
tauRec		<i>time</i>

Exposures

plasticityFactor	(from basePlasticityMechanism)	Dimensionless
-------------------------	--------------------------------	---------------

Event Ports

in	This is where the plasticity mechanism receives spike events from the parent synapse. (from basePlasticityMechanism)	Direction: in
-----------	--	---------------

Dynamics

Structure WITH parent AS **a**

WITH this AS **b**

EVENT CONNECTION from **a** TO **b**

State Variables **R**: Dimensionless

U: Dimensionless

On Start **R** = 1

U = initReleaseProb

On Events EVENT IN on port: **in**

R = **R** * (1 - **U**)

U = **U** + initReleaseProb * (1 - **U**)

Derived Variables **plasticityFactor** = **R** * **U** (exposed as **plasticityFactor**)

Time Derivatives $d \mathbf{R} / dt = (1 - \mathbf{R}) / \tau_{\text{rec}}$

$d \mathbf{U} / dt = (\text{initReleaseProb} - \mathbf{U}) / \tau_{\text{fac}}$

blockingPlasticSynapse

extends *expTwoSynapse*

Biexponential synapse that allows for optional block and plasticity mechanisms, which can be expressed as child elements.

Parameters

erev	Reversal potential of the synapse (from baseConductanceBasedSynapse)	<i>voltage</i>
gbase	Baseline conductance, generally the maximum conductance following a single spike (from baseConductanceBasedSynapse)	<i>conductance</i>
tauDecay	(from expTwoSynapse)	<i>time</i>
tauRise	(from expTwoSynapse)	<i>time</i>

Derived parameters

peak-Time	(from expTwoSynapse)	<i>time</i>
wave-form-Factor	(from expTwoSynapse)	Dimensionless

Children list

plasticityMechanisms		<i>basePlasticityMechanism</i>
block-Mechanisms		<i>baseBlockMechanism</i>

Properties

weight (default: 1)		Dimensionless
----------------------------	--	---------------

Exposures

g	Time varying conductance through the synapse (from baseConductanceBasedSynapse)	<i>conductance</i>
i	The total (usually time varying) current produced by this ComponentType (from basePointCurrent)	<i>current</i>

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (from baseVoltageDepSynapse)	<i>voltage</i>
----------	---	----------------

Event Ports

in	(from baseSynapse)	Direction: in
relay	Used to relay incoming spikes to child plasticity mechanism	Direction: out

Dynamics

State Variables **A**: Dimensionless

B: Dimensionless

On Start **A** = 0

B = 0

On Events EVENT IN on port: **in**

A = **A** + (weight * plasticityFactor * waveformFactor)

B = **B** + (weight * plasticityFactor * waveformFactor)

EVENT OUT on port: **relay**

Derived Variables **plasticityFactor** = plasticityMechanisms[*]->plasticityFactor(reduce method: multiply)

blockFactor = blockMechanisms[*]->blockFactor(reduce method: multiply)

g = blockFactor * gbase * (**B** - **A**) (exposed as **g**)

i = **g** * (erev - v) (exposed as **i**)

Time Derivatives $d\ A / dt = -A / \tau_{rise}$

$d\ B / dt = -B / \tau_{decay}$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BlockingPlasticSynapse

variable = 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, gds_collector_=None, ↪
    ↪ **kwargs_)
```

Usage: XML

```
<blockingPlasticSynapse id="NMDA" gbase=".8nS" tauRise="1e-3s" tauDecay="13.3333e-3s" ↪
    ↪erev="0V">
    <blockMechanism type="voltageConcDepBlockMechanism" species="mg" ↪
    ↪blockConcentration="1.2mM" scalingConc="1.9205441817997078mM" scalingVolt="0.
    ↪016129032258064516V"/>
</blockingPlasticSynapse>
```

```
<blockingPlasticSynapse id="blockStpSynDep" gbase="1nS" erev="0mV" tauRise="0.1ms" ↪
    ↪tauDecay="2ms">
    <notes>A biexponential blocking synapse, with STD.</notes>
    <plasticityMechanism type="tsodyksMarkramDepMechanism" initReleaseProb="0.5" ↪
    ↪tauRec="120 ms"/>
    <blockMechanism type="voltageConcDepBlockMechanism" species="mg" ↪
    ↪blockConcentration="1.2 mM" scalingConc="1.920544 mM" scalingVolt="16.129 mV"/>
</blockingPlasticSynapse>
```

```

<blockingPlasticSynapse id="blockStpSynDepFac" gbase="1nS" erev="0mV" tauRise="0.1ms"_
tauDecay="2ms">
    <notes>A biexponential blocking synapse with short term
    depression and facilitation.</notes>
    <plasticityMechanism type="tsodyksMarkramDepFacMechanism" initReleaseProb="0.5
    " tauRec="120 ms" tauFac="10 ms"/>
    <blockMechanism type="voltageConcDepBlockMechanism" species="mg"_
blockConcentration="1.2 mM" scalingConc="1.920544 mM" scalingVolt="16.129 mV"/>
</blockingPlasticSynapse>

```

doubleSynapse

extends `baseVoltageDepSynapse`

Synapse consisting of two independent synaptic mechanisms (e.g. AMPA-R and NMDA-R), which can be easily colocated in connections.

Paths

<code>synapse1Path</code>
<code>synapse2Path</code>

Component References

<code>synapse1</code>		<code>baseSynapse</code>
<code>synapse2</code>		<code>baseSynapse</code>

Properties

<code>weight</code> (default: 1)		Dimensionless
----------------------------------	--	---------------

Exposures

<code>i</code>	The total (usually time varying) current produced by this ComponentType (<i>from</i> <code>basePointCurrent</code>)	<code>current</code>
----------------	---	----------------------

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepSynapse</i>)	<i>voltage</i>
----------	--	----------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
relay	Used to relay incoming spikes to child mechanisms	Direction: out

Dynamics

Structure WITH this AS **a**

WITH **synapse1Path** AS **b**

WITH **synapse2Path** AS **c**

CHILD INSTANCE: **synapse1**

CHILD INSTANCE: **synapse2**

EVENT CONNECTION from **a** TO **c**

EVENT CONNECTION from **a** TO **b**

State Variables **weightFactor**: Dimensionless

On Events EVENT IN on port: **in**

weightFactor = weight

EVENT OUT on port: **relay**

Derived Variables **i1** = **synapse1->i**

i2 = **synapse2->i**

i = **weightFactor** * (**i1** + **i2**) (exposed as **i**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import DoubleSynapse

variable = 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, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<doubleSynapse id="AMPA_NMDA" synapse1="AMPA" synapse1Path=". /AMPA" synapse2="NMDA"_
synapse2Path=". /NMDA">
    <notes>A single "synapse" which contains both AMPA and NMDA. It is planned_
that the need for extra synapse1Path/synapse2Path attributes can be removed in_
later versions.</notes>
</doubleSynapse>
```

stdpSynapse

extends *expTwoSynapse*

Spike timing dependent plasticity mechanism, NOTE: EXAMPLE NOT YET WORKING!!!

Bioportal entry for Computational Neuroscience Ontology related to stdpSynapse.

Parameters

erev	Reversal potential of the synapse (<i>from baseConductanceBasedSynapse</i>)	<i>voltage</i>
gbase	Baseline conductance, generally the maximum conductance following a single spike (<i>from baseConductanceBasedSynapse</i>)	<i>conductance</i>
tauDe- cay	(<i>from expTwoSynapse</i>)	<i>time</i>
tauRise	(<i>from expTwoSynapse</i>)	<i>time</i>

Derived parameters

peak-Time	(<i>from expTwoSynapse</i>)	<i>time</i>
wave-form-Factor	(<i>from expTwoSynapse</i>)	Dimensionless

Constants

tsinceRate = 1		Dimensionless
longTime = 1000s		<i>time</i>

Exposures

M		Dimensionless
P		Dimensionless
g	Time varying conductance through the synapse (<i>from baseConductanceBasedSynapse</i>)	<i>conductance</i>
i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
tsince		<i>time</i>

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepSynapse</i>)	<i>voltage</i>
----------	--	----------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
-----------	-----------------------------	---------------

Dynamics

State Variables **A**: Dimensionless

B: Dimensionless

M: Dimensionless (exposed as **M**)

P: Dimensionless (exposed as **P**)

tsince: *time* (exposed as **tsince**)

On Start **A** = 0

B = 0

M = 1

P = 1

tsince = longTime

On Events EVENT IN on port: **in**

A = **A** + waveformFactor

B = **B** + waveformFactor

tsince = 0

Derived Variables **g** = gbase * (B - A) (exposed as **g**)

i = **g** * (erev - v) (exposed as **i**)

Time Derivatives d **A** /dt = -A / tauRise

d **B** /dt = -B / tauDecay

$d \text{tsince} /dt = \text{tsinceRate}$

gapJunction

extends [baseSynapse](#)

Gap junction/single electrical connection.

Parameters

conductance		<i>conductance</i>
-------------	--	--------------------

Properties

weight (default: 1)	Dimensionless
---------------------	---------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
---	--	----------------

Requirements

v	<i>voltage</i>
---	----------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
----	-----------------------------	---------------

Dynamics

Derived Variables $v_{peer} = \text{peer-} > v$

$\mathbf{i} = \text{weight} * \text{conductance} * (v_{peer} - v) \quad (\text{exposed as } \mathbf{i})$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GapJunction

variable = GapJunction(neuro_lex_id=None, id=None, metaid=None, notes=None,_
    properties=None, annotation=None, conductance=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<gapJunction id="gj1" conductance="10pS"/>
```

```
<gapJunction id="gj1" conductance="10pS"/>
```

baseGradedSynapse

extends baseSynapse

Base type for graded synapses.

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from</i> basePointCurrent)	current
---	---	---------

Event Ports

in	(<i>from</i> baseSynapse)	Direction: in
----	----------------------------	---------------

silentSynapse

extends baseGradedSynapse

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

Properties

weight (default: 1)		Dimensionless
---------------------	--	---------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
----------	--	----------------

Requirements

v	<i>voltage</i>
----------	----------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
-----------	-----------------------------	---------------

Dynamics

Derived Variables `vpeer = peer->v`

`i = 0` (exposed as **i**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SilentSynapse

variable = SilentSynapse(neuro_lex_id=None, id=None, metaid=None, notes=None, _properties=None, annotation=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<silentSynapse id="silent1"/>
```

```
<silentSynapse id="silent2"/>
```

```
<silentSynapse id="silent1"/>
```

linearGradedSynapse

extends [baseGradedSynapse](#)

Behaves just like a one way gap junction.

Parameters

conductance		<i>conductance</i>
-------------	--	--------------------

Properties

weight (default: 1)		Dimensionless
---------------------	--	---------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
---	--	----------------

Requirements

v		<i>voltage</i>
---	--	----------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
----	-----------------------------	---------------

Dynamics

Derived Variables vpeer = peer->v

i = weight * conductance * (vpeer - v) (exposed as **i**)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import LinearGradedSynapse

variable = LinearGradedSynapse(neuro_lex_id=None, id=None, metaid=None, notes=None, properties=None, annotation=None, conductance=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<linearGatedSynapse id="gs1" conductance="5pS"/>
```

gradedSynapse

extends [baseGradedSynapse](#)

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

Parameters

Vth	The half-activation voltage of the synapse	<i>voltage</i>
conduc-tance		<i>conductance</i>
delta	Slope of the activation curve	<i>voltage</i>
erev	The reversal potential of the synapse	<i>voltage</i>
k	Rate constant for transmitter-receptor dissociation rate	<i>per_time</i>

Properties

weight (default: 1)		Dimensionless
----------------------------	--	---------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
inf		Dimensionless
tau		<i>time</i>

Requirements

v		<i>voltage</i>
----------	--	----------------

Event Ports

in	(from baseSynapse)	Direction: in
----	--------------------	---------------

Dynamics

State Variables s: Dimensionless

On Conditions IF (1-inf) < 1e-4 THEN

$$s = \inf$$

Derived Variables vpeer = peer->v

$$\inf = 1/(1 + \exp((V_{th} - vpeer)/\delta)) \quad (\text{exposed as } \inf)$$

$$\tau = (1-\inf)/k \quad (\text{exposed as } \tau)$$

$$i = weight * conductance * s * (erev - v) \quad (\text{exposed as } i)$$

Conditional Derived Variables IF (1-inf) > 1e-4 THEN

$$s_rate = (\inf - s)/\tau$$

OTHERWISE

$$s_rate = 0$$

Time Derivatives d s /dt = s_rate

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GradedSynapse

variable = 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, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<gradedSynapse id="gs2" conductance="5pS" delta="5mV" Vth="-55mV" k="0.025per_ms"_  
    erev="0mV"/>
```

```
<gradedSynapse id="gs1" conductance="0.1nS" delta="5mV" Vth="-35mV" k="0.025per_ms"_  
    erev="0mV"/>
```

10.1.7 Inputs

A number of ComponentTypes for providing spiking (e.g. *spikeGeneratorPoisson*, *spikeArray*) and current inputs (e.g. *pulseGenerator*, *voltageClamp*, *timedSynapticInput*, *poissonFiringSynapse*) to other ComponentTypes

Original ComponentType definitions: [Inputs.xml](#). Schema against which NeuroML based on these should be valid: [NeuroML_v2.2.xsd](#). Generated on 07/06/22 from [this commit](#). Please file any issues or questions at the issue tracker [here](#).

basePointCurrent

extends [baseStandalone](#)

Base type for all ComponentTypes which produce a current **i** (with dimension current).

Exposures

i	The total (usually time varying) current produced by this ComponentType	<i>current</i>
----------	---	----------------

baseVoltageDepPointCurrent

extends [basePointCurrent](#)

Base type for all ComponentTypes which produce a current **i** (with dimension current) and require a voltage **v** exposed on the parent Component, which would often be the membrane potential of a Component extending [baseCellMembPot](#).

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
----------	--	----------------

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed	<i>voltage</i>
----------	--	----------------

baseVoltageDepPointCurrentSpiking

extends [baseVoltageDepPointCurrent](#)

Base type for all ComponentTypes which produce a current **i**, require a membrane potential **v** exposed on the parent and emit spikes (on a port **spike**). The exposed variable **tsince** can be used for plotting the time since the Component has spiked last.

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
tsince	Time since the last spike was emitted	<i>time</i>

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepPointCurrent</i>)	<i>voltage</i>
----------	---	----------------

Event Ports

spike	Port on which spikes are emitted	Direction: out
--------------	----------------------------------	----------------

basePointCurrentDL

Base type for all ComponentTypes which produce a dimensionless current **I**. There are many dimensionless equivalents of all the core current producing ComponentTypes such as *pulseGenerator / pulseGeneratorDL*, *sineGenerator / sineGeneratorDL* and *rampGenerator / rampGeneratorDL*.

Exposures

I	The total (time varying) current produced by this ComponentType	Dimensionless
----------	---	---------------

baseVoltageDepPointCurrentDL

extends **basePointCurrentDL**

Base type for all ComponentTypes which produce a dimensionless current **I** and require a dimensionless membrane potential **V** exposed on the parent Component.

Exposures

I	The total (time varying) current produced by this ComponentType (<i>from basePointCurrentDL</i>)	Dimensionless
----------	--	---------------

Requirements

V	The current may vary with the dimensionless voltage exposed by the ComponentType on which this is placed	Dimensionless
----------	--	---------------

baseSpikeSource

Base for any ComponentType whose main purpose is to emit spikes (on a port **spike**). The exposed variable **tsince** can be used for plotting the time since the Component has spiked last.

Exposures

tsince	Time since the last spike was emitted	<i>time</i>
---------------	---------------------------------------	-------------

Event Ports

spike	Port on which spikes are emitted	Direction: out
--------------	----------------------------------	----------------

spikeGenerator

extends [baseSpikeSource](#)

Simple generator of spikes at a regular interval set by **period**.

Parameters

period	Time between spikes. The first spike will be emitted after this time.	<i>time</i>
---------------	---	-------------

Constants

SMALL_TIME = 1e-9ms		<i>time</i>
----------------------------	--	-------------

Exposures

tnext	When the next spike should ideally be emitted (dt permitting)	<i>time</i>
tsince	Time since the last spike was emitted (<i>from baseSpikeSource</i>)	<i>time</i>

Event Ports

spike	Port on which spikes are emitted (<i>from baseSpikeSource</i>)	Direction: out
--------------	--	----------------

Dynamics

State Variables **tsince**: *time* (exposed as **tsince**)

tnext: *time* (exposed as **tnext**)

On Start **tsince** = 0

tnext = period

On Conditions IF **tnext** - **t** < SMALL_TIME THEN

tsince = 0

tnext = **tnext**+period

EVENT OUT on port: **spike**

Time Derivatives $d \text{ tsince} /dt = 1$

$d \text{ tnext} /dt = 0$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SpikeGenerator

variable = SpikeGenerator(neuro_lex_id=None, id=None, metaid=None, notes=None,_
                           properties=None, annotation=None, period=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<spikeGenerator id="spikeGenRegular" period="20 ms"/>
```

spikeGeneratorRandom

extends **baseSpikeSource**

Generator of spikes with a random interspike interval of at least **minISI** and at most **maxISI**.

Parameters

maxISI	Maximum interspike interval	<i>time</i>
minISI	Minimum interspike interval	<i>time</i>

Constants

MSEC = 1ms		<i>time</i>
-------------------	--	-------------

Exposures

isi	The interval until the next spike	<i>time</i>
tnext	When the next spike should ideally be emitted (dt permitting)	<i>time</i>
tsince	Time since the last spike was emitted (<i>from baseSpikeSource</i>)	<i>time</i>

Event Ports

spike	Port on which spikes are emitted (<i>from baseSpikeSource</i>)	Direction: out
--------------	--	----------------

Dynamics

State Variables **tsince**: *time* (exposed as **tsince**)

tnext: *time* (exposed as **tnext**)

isi: *time* (exposed as **isi**)

On Start **tsince** = 0

isi = minISI + MSEC * random((maxISI - minISI) / MSEC)

tnext = **isi**

On Conditions IF t > tnext THEN

isi = minISI + MSEC * random((maxISI - minISI) / MSEC)

tsince = 0

tnext = tnext+**isi**

EVENT OUT on port: **spike**

Time Derivatives d **tsince** /dt = 1

d **tnext** /dt = 0

Usage: Python

Go to the [libNeuroML documentation](#)

```
from neuroml import SpikeGeneratorRandom

variable = SpikeGeneratorRandom(neuro_lex_id=None, id=None, metaid=None, notes=None,_
                                properties=None, annotation=None, max_isi=None, min_isi=None, gds_collector_=None,_
                                **kwargs_)
```

Usage: XML

```
<spikeGeneratorRandom id="spikeGenRandom" minISI="10 ms" maxISI="30 ms"/>
```

spikeGeneratorPoisson

extends [baseSpikeSource](#)

Generator of spikes whose ISI is distributed according to an exponential PDF with scale: 1 / **averageRate**.

Parameters

aver-ageRate	The average rate at which spikes are emitted	<i>per_time</i>
---------------------	--	-----------------

Constants

SMALL_TIME = 1e-9ms	<i>time</i>
----------------------------	-------------

Exposures

isi	The interval until the next spike	<i>time</i>
tnex-tIdeal	This is the ideal/perfect next spike time, based on a newly generated isi, but dt precision will mean that it's usually slightly later than this	<i>time</i>
tnex-tUsed	This is the next spike time for practical purposes, ensuring that it's later than the current time	<i>time</i>
tsince	Time since the last spike was emitted (<i>from baseSpikeSource</i>)	<i>time</i>

Event Ports

spike	Port on which spikes are emitted (<i>from baseSpikeSource</i>)	Direction: out
--------------	--	----------------

Dynamics

State Variables **tsince**: *time* (exposed as **tsince**)

tnextIdeal: *time* (exposed as **tnextIdeal**)

tnextUsed: *time* (exposed as **tnextUsed**)

isi: *time* (exposed as **isi**)

On Start **tsince** = 0

isi = -1 * log(random(1)) / averageRate

tnextIdeal = **isi**

tnextUsed = **isi**

On Conditions IF t > **tnextUsed** THEN

tsince = 0

isi = -1 * log(random(1)) / averageRate

tnextIdeal = (**tnextIdeal**+**isi**)

tnextUsed = **tnextIdeal***H((**tnextIdeal**-t)/t) + (t+SMALL_TIME)*H((t-**tnextIdeal**)/t)

EVENT OUT on port: **spike**

Time Derivatives d **tsince** /dt = 1

d **tnextUsed** /dt = 0

d **tnextIdeal** /dt = 0

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SpikeGeneratorPoisson

variable = SpikeGeneratorPoisson(neuro_lex_id=None, id=None, metaid=None, notes=None, _properties=None, annotation=None, average_rate=None, extensiontype_=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<spikeGeneratorPoisson id="spikeGenPoisson" averageRate="50 Hz"/>
```

spikeGeneratorRefPoissonextends *spikeGeneratorPoisson*

Generator of spikes whose ISI distribution is the maximum entropy distribution over [**minimumISI**, +infinity) with mean: 1 / **averageRate**.

Parameters

aver-ageRate	The average rate at which spikes are emitted (<i>from spikeGeneratorPoisson</i>)	<i>per_time</i>
min-imu-mISI	The minimum interspike interval	<i>time</i>

Derived parameters

aver-ageIsi	The average interspike interval	<i>time</i>
--------------------	---------------------------------	-------------

Exposures

isi	The interval until the next spike (<i>from spikeGeneratorPoisson</i>)	<i>time</i>
tnex-tIdeal	This is the ideal/perfect next spike time, based on a newly generated isi, but dt precision will mean that it's usually slightly later than this (<i>from spikeGeneratorPoisson</i>)	<i>time</i>
tnex-tUsed	This is the next spike time for practical purposes, ensuring that it's later than the current time (<i>from spikeGeneratorPoisson</i>)	<i>time</i>
tsince	Time since the last spike was emitted (<i>from baseSpikeSource</i>)	<i>time</i>

Event Ports

spike	Port on which spikes are emitted (<i>from baseSpikeSource</i>)	Direction: out
--------------	--	----------------

Dynamics

State Variables **tsince**: *time* (exposed as **tsince**)

tnextIdeal: *time* (exposed as **tnextIdeal**)

tnextUsed: *time* (exposed as **tnextUsed**)

isi: *time* (exposed as **isi**)

On Start **tsince** = 0

isi = minimumISI - (averageIsi-minimumISI) * log(random(1))

tnextIdeal = **isi**

tnextUsed = **isi**

On Conditions IF t > tnextUsed THEN

tsince = 0

isi = minimumISI - (averageIsi-minimumISI) * log(random(1))

tnextIdeal = (tnextIdeal+isi)

tnextUsed = tnextIdeal*H((tnextIdeal-t)/t) + (t+SMALL_TIME)*H((t-tnextIdeal)/t)

EVENT OUT on port: **spike**

Time Derivatives d **tsince** /dt = 1

d **tnextUsed** /dt = 0

d **tnextIdeal** /dt = 0

Usage: Python

Go to the *libNeuroML* documentation

```
from neuroml import SpikeGeneratorRefPoisson

variable = SpikeGeneratorRefPoisson(neuro_lex_id=None, id=None, metaid=None,
notes=None, properties=None, annotation=None, average_rate=None, minimum_isi=None,
gds_collector_=None, **kwargs_)
```

Usage: XML

```
<spikeGeneratorRefPoisson id="spikeGenRefPoisson" averageRate="50 Hz" minimumISI="10
ms"/>
```

poissonFiringSynapse

extends [baseVoltageDepPointCurrentSpiking](#)

Poisson spike generator firing at **averageRate**, which is connected to single **synapse** that is triggered every time a spike is generated, producing an input current. See also [transientPoissonFiringSynapse](#).

Parameters

aver-ageRate	The average rate at which spikes are emitted	<i>per_time</i>
---------------------	--	-----------------

Derived parameters

aver-ageIsi	The average interspike interval	<i>time</i>
--------------------	---------------------------------	-------------

Paths

spikeTar-get	The target of the spikes, i.e. the synapse
---------------------	--

Component References

synapse	<i>baseSynapse</i>
----------------	--------------------

Constants

SMALL_TIME = 1e-9ms	<i>time</i>
----------------------------	-------------

Properties

weight (default: 1)	Dimensionless
----------------------------	---------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
isi	The interval until the next spike	<i>time</i>
tnextIdeal		<i>time</i>
tnextUsed		<i>time</i>
tsince	Time since the last spike was emitted (<i>from baseVoltageDepPointCurrentSpiking</i>)	<i>time</i>

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepPointCurrent</i>)	<i>voltage</i>
----------	---	----------------

Event Ports

in	Note this is not used here. Will be removed in future	Direction: in
spike	Port on which spikes are emitted	Direction: out
spike	Port on which spikes are emitted (<i>from baseVoltageDepPointCurrentSpiking</i>)	Direction: out

Dynamics

Structure WITH this AS a

WITH **spikeTarget** AS **b**

CHILD INSTANCE: **synapse**

EVENT CONNECTION from **a** TO **b**

State Variables **tsince:** *time* (exposed as **tsince**)

tnextIdeal: *time* (exposed as **tnextIdeal**)

tnextUsed: *time* (exposed as **tnextUsed**)

isi: *time* (exposed as **isi**)

On Start **tsince** = 0

isi = - averageIsi * log(random(1))

tnextIdeal = **isi**

tnextUsed = **isi**

On Conditions IF t > **tnextUsed** THEN

tsince = 0

isi = - averageIsi * log(1 - random(1))

tnextIdeal = (**tnextIdeal**+**isi**)

tnextUsed = tnextIdeal*H((tnextIdeal-t)/t) + (t+SMALL_TIME)*H((t-tnextIdeal)/t)

EVENT OUT on port: **spike**

Derived Variables **iSyn** = synapse->i

i = weight * iSyn (exposed as **i**)

Time Derivatives $d \text{ tsince} /dt = 1$

$d \text{ tnextUsed} /dt = 0$

$d \text{ tnextIdeal} /dt = 0$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import PoissonFiringSynapse

variable = PoissonFiringSynapse(neuro_lex_id=None, id=None, metaid=None, notes=None,
                                properties=None, annotation=None, average_rate=None, synapse=None, spike_
                                _target=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<poissonFiringSynapse id="poissonFiringSyn" averageRate="10 Hz" synapse="synInput"_
    &spikeTarget=".//synInput"/>
```

transientPoissonFiringSynapse

extends `baseVoltageDepPointCurrentSpiking`

Poisson spike generator firing at **averageRate** after a **delay** and for a **duration**, connected to single **synapse** that is triggered every time a spike is generated, providing an input current. Similar to `ComponentType poissonFiringSynapse`.

Parameters

aver-ageRate		<i>per_time</i>
delay		<i>time</i>
dura-tion		<i>time</i>

Derived parameters

averageIsi		<i>time</i>
------------	--	-------------

Paths

spikeTarget	
-------------	--

Component References

synapse		<i>baseSynapse</i>
---------	--	--------------------

Constants

SMALL_TIME = 1e-9ms		<i>time</i>
LONG_TIME = 1e9hour		<i>time</i>

Properties

weight (default: 1)		Dimensionless
---------------------	--	---------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
isi		<i>time</i>
tnex-tIdeal		<i>time</i>
tnex-tUsed		<i>time</i>
tsince	Time since the last spike was emitted (<i>from baseVoltageDepPointCurrentSpiking</i>)	<i>time</i>

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepPointCurrent</i>)	<i>voltage</i>
----------	---	----------------

Event Ports

in	Note this is not used here. Will be removed in future	Direction: in
spike	Port on which spikes are emitted	Direction: out
spike	Port on which spikes are emitted (<i>from baseVoltageDepPointCurrentSpiking</i>)	Direction: out

Dynamics

Structure WITH this AS **a**

WITH **spikeTarget** AS **b**

CHILD INSTANCE: **synapse**

EVENT CONNECTION from **a** TO **b**

State Variables **tsince**: *time* (exposed as **tsince**)

tnextIdeal: *time* (exposed as **tnextIdeal**)

tnextUsed: *time* (exposed as **tnextUsed**)

isi: *time* (exposed as **isi**)

On Start **tsince** = 0

isi = - averageIsi * log(1 - random(1)) + delay

tnextIdeal = **isi**

tnextUsed = **isi**

On Conditions IF t > **tnextUsed** THEN

tsince = 0

isi = - averageIsi * log(1 - random(1))

tnextIdeal = (**tnextIdeal**+**isi**) + H(((t+**isi**) - (delay+duration))/duration)*LONG_TIME

tnextUsed = **tnextIdeal***H((tnextIdeal-t)/t) + (t+SMALL_TIME)*H((t-tnextIdeal)/t)

EVENT OUT on port: **spike**

Derived Variables **iSyn** = **synapse->i**

i = weight * **iSyn** (exposed as **i**)

Time Derivatives $d \text{ tsince} / dt = 1$

$d \text{ tnextUsed} / dt = 0$

$d \text{ tnextIdeal} / dt = 0$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import TransientPoissonFiringSynapse

variable = 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, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<transientPoissonFiringSynapse id="transPoissonFiringSyn" delay="50ms" duration="50ms"
                                averageRate="300 Hz" synapse="synInputFast" spikeTarget="./synInputFast"/>
```

```
<transientPoissonFiringSynapse id="transPoissonFiringSyn2" delay="50ms" duration=
                                "500ms" averageRate="10 Hz" synapse="synInputFastTwo" spikeTarget="./synInputFastTwo"
                                />
```

timedSynapticInput

extends baseVoltageDepPointCurrentSpiking

Spike array connected to a single **synapse**, producing a current triggered by each *spike* in the array.

Paths

spikeTar- get	
------------------	--

Component References

synapse	baseSynapse
---------	-------------

Children list

spikes	spike
--------	-------

Properties

weight (default: 1)	Dimensionless
----------------------------	---------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
tsince	Time since the last spike was emitted (<i>from baseVoltageDepPointCurrentSpiking</i>)	<i>time</i>

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepPointCurrent</i>)	<i>voltage</i>
----------	---	----------------

Event Ports

in	This will receive events from the children	Direction: in
spike	Port on which spikes are emitted (<i>from baseVoltageDepPointCurrentSpiking</i>)	Direction: out

Dynamics

Structure WITH this AS a

WITH **spikeTarget** AS **b**

CHILD INSTANCE: **synapse**

EVENT CONNECTION from **a** TO **b**

State Variables **tsince**: *time* (exposed as **tsince**)

On Events EVENT IN on port: **in**

tsince = 0

EVENT OUT on port: **spike**

Derived Variables **iSyn** = synapse->i

i = weight * **iSyn** (exposed as **i**)

Time Derivatives d **tsince** /dt = 1

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import TimedSynapticInput

variable = TimedSynapticInput(neuro_lex_id=None, id=None, metaid=None, notes=None,
    ↪properties=None, annotation=None, synapse=None, spike_target=None, spikes=None,
    ↪collector_=None, **kwargs_)
```

Usage: XML

```
<timedSynapticInput id="synTrain" synapse="synInputFastTwo" spikeTarget=". /"
    ↪synInputFastTwo">
    <spike id="0" time="2 ms"/>
    <spike id="1" time="15 ms"/>
    <spike id="2" time="27 ms"/>
    <spike id="3" time="40 ms"/>
    <spike id="4" time="45 ms"/>
    <spike id="5" time="50 ms"/>
    <spike id="6" time="52 ms"/>
    <spike id="7" time="54 ms"/>
    <spike id="8" time="54.5 ms"/>
    <spike id="9" time="54.6 ms"/>
    <spike id="10" time="54.7 ms"/>
    <spike id="11" time="54.8 ms"/>
    <spike id="12" time="54.9 ms"/>
    <spike id="13" time="55 ms"/>
    <spike id="14" time="55.1 ms"/>
    <spike id="15" time="55.2 ms"/>
</timedSynapticInput>
```

spikeArray

extends baseSpikeSource

Set of spike ComponentTypes, each emitting one spike at a certain time. Can be used to feed a predetermined spike train into a cell.

Children list

spikes	<i>spike</i>
---------------	--------------

Exposures

tsince	Time since the last spike was emitted (<i>from baseSpikeSource</i>)	<i>time</i>
---------------	---	-------------

Event Ports

in	This will receive events from the children	Direction: in
spike	Port on which spikes are emitted (<i>from baseSpikeSource</i>)	Direction: out

Dynamics

State Variables `tsince: time` (exposed as `tsince`)

On Start `tsince = 0`

On Events EVENT IN on port: `in`

`tsince = 0`

EVENT OUT on port: `spike`

Time Derivatives `d tsince /dt = 1`

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SpikeArray

variable = SpikeArray(neuro_lex_id=None, id=None, metaid=None, notes=None,_
    properties=None, annotation=None, spikes=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<spikeArray id="spkArr">
    <spike id="0" time="50 ms"/>
    <spike id="1" time="100 ms"/>
    <spike id="2" time="150 ms"/>
    <spike id="3" time="155 ms"/>
    <spike id="4" time="250 ms"/>
</spikeArray>
```

spikeextends [baseSpikeSource](#)Emits a single spike at the specified **time**.**Parameters**

time	Time at which to emit one spike event	<i>time</i>
-------------	---------------------------------------	-------------

Exposures

spiked	0 signals not yet spiked, 1 signals has spiked	Dimensionless
tsince	Time since the last spike was emitted (<i>from baseSpikeSource</i>)	<i>time</i>

Event Ports

spike	Port on which spikes are emitted (<i>from baseSpikeSource</i>)	Direction: out
--------------	--	----------------

Dynamics**Structure** WITH this AS **a**WITH parent AS **b**EVENT CONNECTION from **a** TO **b****State Variables** **tsince**: *time* (exposed as **tsince**)**spiked**: Dimensionless (exposed as **spiked**)**On Start** **tsince** = 0**On Conditions** IF (t >= time) AND (spiked = 0) THEN **spiked** = 1 **tsince** = 0 EVENT OUT on port: **spike****Time Derivatives** d **tsince** /dt = 1

Usage: Python

Go to the [libNeuroML documentation](#)

```
from neuroml import Spike

variable = Spike(neuro_lex_id=None, id=None, time=None, gds_collector_=None, **kwargs_
    )
```

Usage: XML

```
<spike id="0" time="50 ms"/>
```

```
<spike id="1" time="100 ms"/>
```

```
<spike id="2" time="150 ms"/>
```

pulseGenerator

extends [basePointCurrent](#)

Generates a constant current pulse of a certain **amplitude** for a specified **duration** after a **delay**. Scaled by **weight**, if set.

Parameters

amplitude	Amplitude of current pulse	<i>current</i>
delay	Delay before change in current. Current is zero prior to this.	<i>time</i>
duration	Duration for holding current at amplitude. Current is zero after delay + duration.	<i>time</i>

Properties

weight (default: 1)		Dimensionless
----------------------------	--	---------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
----------	--	----------------

Event Ports

in	Note: this is not used here. Will be removed in future	Direction: in
-----------	--	---------------

Dynamics

State Variables **i**: *current* (exposed as **i**)

On Events EVENT IN on port: **in**

On Conditions IF $t < \text{delay}$ THEN

i = 0

IF $t \geq \text{delay}$ AND $t < \text{duration} + \text{delay}$ THEN

i = weight * amplitude

IF $t \geq \text{duration} + \text{delay}$ THEN

i = 0

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import PulseGenerator

variable = PulseGenerator(neuro_lex_id=None, id=None, metaid=None, notes=None,
                           properties=None, annotation=None, delay=None, duration=None, amplitude=None, gds_
                           _collector=None, **kwargs_)
```

Usage: XML

```
<pulseGenerator id="pulseGen1" delay="50ms" duration="200ms" amplitude="0.0032nA"/>
```

```
<pulseGenerator id="pulseGen2" delay="400ms" duration="200ms" amplitude="0.0020nA"/>
```

```
<pulseGenerator id="pulseGen3" delay="700ms" duration="200ms" amplitude="0.0010nA"/>
```

compoundInput

extends *basePointCurrent*

Generates a current which is the sum of all its child *basePointCurrent* element, e.g. can be a combination of *pulseGenerator*, *sineGenerator* elements producing a single **i**. Scaled by **weight**, if set.

Children list

currents		basePointCurrent
----------	--	------------------

Properties

weight (default: 1)		Dimensionless
---------------------	--	---------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from</i> basePointCurrent)	current
---	---	---------

Event Ports

in	Note this is not used here. Will be removed in future	Direction: in
----	---	---------------

Dynamics

On Events EVENT IN on port: in

Derived Variables i_total = currents[*]->i(reduce method: add)

i = weight * i_total (exposed as i)

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import CompoundInput

variable = CompoundInput(neuro_lex_id=None, id=None, metaid=None, notes=None,
                           properties=None, annotation=None, pulse_generators=None, sine_generators=None, ramp_
                           generators=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<compoundInput id="ci0">
    <pulseGenerator id="pg1" delay="50ms" duration="200ms" amplitude=".8 nA"/>
    <pulseGenerator id="pg2" delay="100ms" duration="100ms" amplitude=".4 nA"/>
    <sineGenerator id="sg0" phase="0" delay="125ms" duration="50ms" amplitude=".4nA" period="25ms"/>
</compoundInput>
```

compoundInputDLextends [basePointCurrentDL](#)

Generates a current which is the sum of all its child [basePointCurrentDL](#) elements, e.g. can be a combination of [pulseGeneratorDL](#), [sineGeneratorDL](#) elements producing a single **i**. Scaled by **weight**, if set.

Children list

cur- rents		basePointCurrentDL
-----------------------------	--	------------------------------------

Properties

weight (default: 1)	Dimensionless
----------------------------	---------------

Exposures

I	The total (time varying) current produced by this ComponentType (<i>from base-PointCurrentDL</i>)	Dimensionless
----------	---	---------------

Event Ports

in	Note this is not used here. Will be removed in future	Direction: in
-----------	---	---------------

Dynamics**On Events** EVENT IN on port: **in****Derived Variables** **I_total** = currents[*]->I(reduce method: add)**I** = weight * I_total (exposed as **I**)**Usage: Python***Go to the libNeuroML documentation*

```
from neuroml import CompoundInputDL

variable = 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, gds_collector_=None, **kwargs_)
```

pulseGeneratorDL

extends [basePointCurrentDL](#)

Dimensionless equivalent of [pulseGenerator](#). Generates a constant current pulse of a certain **amplitude** for a specified **duration** after a **delay**. Scaled by **weight**, if set.

Parameters

ampli-tude	Amplitude of current pulse	Dimensionless
delay	Delay before change in current. Current is zero prior to this.	<i>time</i>
dura-tion	Duration for holding current at amplitude. Current is zero after delay + duration.	<i>time</i>

Properties

weight (default: 1)		Dimensionless
----------------------------	--	---------------

Exposures

I	The total (time varying) current produced by this ComponentType (<i>from base-PointCurrentDL</i>)	Dimensionless
----------	---	---------------

Event Ports

in	Note this is not used here. Will be removed in future	Direction: in
-----------	---	---------------

Dynamics

State Variables **I**: Dimensionless (exposed as **I**)

On Events EVENT IN on port: **in**

On Conditions IF $t < \text{delay}$ THEN

$$I = 0$$

IF $t \geq \text{delay}$ AND $t < \text{duration} + \text{delay}$ THEN

$$I = \text{weight} * \text{amplitude}$$

IF $t \geq \text{duration} + \text{delay}$ THEN

$$I = 0$$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import PulseGeneratorDL

variable = PulseGeneratorDL(neuro_lex_id=None, id=None, metaid=None, notes=None,_
    ↪properties=None, annotation=None, delay=None, duration=None, amplitude=None, gds__
    ↪collector=None, **kwargs_)
```

sineGenerator

extends `basePointCurrent`

Generates a sinusoidally varying current after a time **delay**, for a fixed **duration**. The **period** and maximum **amplitude** of the current can be set as well as the **phase** at which to start. Scaled by **weight**, if set.

Parameters

ampli-tude	Maximum amplitude of current	<i>current</i>
delay	Delay before change in current. Current is zero prior to this.	<i>time</i>
dura-tion	Duration for holding current at amplitude. Current is zero after delay + duration.	<i>time</i>
period	Time period of oscillation	<i>time</i>
phase	Phase (between 0 and 2*pi) at which to start the varying current (i.e. at time given by delay)	Dimensionless

Properties

weight (default: 1)		Dimensionless
----------------------------	--	---------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
----------	--	----------------

Event Ports

in	Direction: in
-----------	---------------

Dynamics

State Variables **i**: *current* (exposed as **i**)

On Events EVENT IN on port: **in**

On Conditions IF $t < \text{delay}$ THEN

i = 0

IF $t \geq \text{delay}$ AND $t < \text{duration} + \text{delay}$ THEN

i = weight * amplitude * sin(phase + (2 * 3.14159265 * (t-delay)/period))

IF $t \geq \text{duration} + \text{delay}$ THEN

i = 0

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SineGenerator

variable = 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, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<sineGenerator id="sg0" phase="0" delay="50ms" duration="200ms" amplitude="1.4nA"_
               period="50ms"/>
```

```
<sineGenerator id="sg0" phase="0" delay="125ms" duration="50ms" amplitude=".4nA"_
               period="25ms"/>
```

sineGeneratorDL

extends `basePointCurrentDL`

Dimensionless equivalent of `sineGenerator`. Generates a sinusoidally varying current after a time **delay**, for a fixed **duration**. The **period** and maximum **amplitude** of the current can be set as well as the **phase** at which to start. Scaled by **weight**, if set.

Parameters

amplitude	Maximum amplitude of current	Dimensionless
delay	Delay before change in current. Current is zero prior to this.	<i>time</i>
duration	Duration for holding current at amplitude. Current is zero after delay + duration.	<i>time</i>
period	Time period of oscillation	<i>time</i>
phase	Phase (between 0 and 2*pi) at which to start the varying current (i.e. at time given by delay)	Dimensionless

Properties

weight (default: 1)	Dimensionless
----------------------------	---------------

Exposures

I	The total (time varying) current produced by this ComponentType (<i>from</i> base-PointCurrentDL)	Dimensionless
----------	---	---------------

Event Ports

in	Direction: in
-----------	---------------

Dynamics

State Variables **I**: Dimensionless (exposed as **I**)

On Events EVENT IN on port: **in**

On Conditions IF t < delay THEN

I = 0

IF t >= delay AND t < duration+delay THEN

I = weight * amplitude * sin(phase + (2 * 3.14159265 * (t-delay)/period))

IF t >= duration+delay THEN

I = 0

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SineGeneratorDL

variable = SineGeneratorDL(neuro_lex_id=None, id=None, metaid=None, notes=None,
                           properties=None, annotation=None, delay=None, duration=None,
                           amplitude=None, period=None, gds_collector_=None, **kwargs_)
```

rampGenerator

extends [basePointCurrent](#)

Generates a ramping current after a time **delay**, for a fixed **duration**. During this time the current steadily changes from **startAmplitude** to **finishAmplitude**. Scaled by **weight**, if set.

Parameters

baselineAmplitude	Amplitude of current before time delay, and after time delay + duration	<i>current</i>
delay	Delay before change in current. Current is baselineAmplitude prior to this.	<i>time</i>
duration	Duration for holding current at amplitude. Current is baselineAmplitude after delay + duration.	<i>time</i>
finishAmplitude	Amplitude of linearly varying current at time delay + duration	<i>current</i>
startAmplitude	Amplitude of linearly varying current at time delay	<i>current</i>

Properties

weight (default: 1)		Dimensionless
----------------------------	--	---------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
----------	--	----------------

Event Ports

in	Direction: in
----	---------------

Dynamics

State Variables **i**: *current* (exposed as **i**)

On Start **i** = baselineAmplitude

On Events EVENT IN on port: **in**

On Conditions IF t < delay THEN

i = weight * baselineAmplitude

IF t >= delay AND t < duration+delay THEN

i = weight * (startAmplitude + (finishAmplitude - startAmplitude) * (t - delay) / (duration))

IF t >= duration+delay THEN

i = weight * baselineAmplitude

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import RampGenerator

variable = 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, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<rampGenerator id="rg0" delay="50ms" duration="200ms" startAmplitude="0.5nA"_
               finishAmplitude="4nA" baselineAmplitude="0nA"/>
```

rampGeneratorDL

extends `basePointCurrentDL`

Dimensionless equivalent of `rampGenerator`. Generates a ramping current after a time **delay**, for a fixed **duration**. During this time the dimensionless current steadily changes from **startAmplitude** to **finishAmplitude**. Scaled by **weight**, if set.

Parameters

baselineAmplitude	Amplitude of current before time delay, and after time delay + duration	Dimensionless
delay	Delay before change in current. Current is baselineAmplitude prior to this.	<i>time</i>
duration	Duration for holding current at amplitude. Current is baselineAmplitude after delay + duration.	<i>time</i>
finishAmplitude	Amplitude of linearly varying current at time delay + duration	Dimensionless
startAmplitude	Amplitude of linearly varying current at time delay	Dimensionless

Properties

weight (default: 1)	Dimensionless
----------------------------	---------------

Exposures

I	The total (time varying) current produced by this ComponentType (<i>from base-PointCurrentDL</i>)	Dimensionless
----------	---	---------------

Event Ports

in	Direction: in
-----------	---------------

Dynamics

State Variables **I**: Dimensionless (exposed as **I**)

On Start **I** = baselineAmplitude

On Events EVENT IN on port: **in**

On Conditions IF t < delay THEN

I = weight * baselineAmplitude

IF t >= delay AND t < duration+delay THEN

I = weight * (startAmplitude + (finishAmplitude - startAmplitude) * (t - delay) / (duration))

IF t >= duration+delay THEN

I = weight * baselineAmplitude

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import RampGeneratorDL

variable = 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, gds_collector_=None, **kwargs_)
```

voltageClamp

extends `baseVoltageDepPointCurrent`

Voltage clamp. Applies a variable current `i` to try to keep parent at `targetVoltage`. Not yet fully tested!!! Consider using `voltageClampTriple!!`

Parameters

<code>delay</code>	Delay before change in current. Current is zero prior to this.	<i>time</i>
<code>dura-</code> <code>tion</code>	Duration for attempting to keep parent at targetVoltage. Current is zero after delay + duration.	<i>time</i>
<code>simple-</code> <code>Series-</code> <code>Resis-</code> <code>tance</code>	Current will be calculated by the difference in voltage between the target and parent, divided by this value	<i>resistance</i>
<code>target-</code> <code>Voltage</code>	Current will be applied to try to get parent to this target voltage	<i>voltage</i>

Properties

<code>weight</code> (default: 1)		Dimensionless
----------------------------------	--	---------------

Exposures

<code>i</code>	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
----------------	--	----------------

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepPointCurrent</i>)	voltage
---	---	---------

Event Ports

in	Note this is not used here. Will be removed in future	Direction: in
----	---	---------------

Dynamics

State Variables i: *current* (exposed as i)

On Events EVENT IN on port: in

On Conditions IF t < delay THEN

i = 0

IF t >= delay THEN

i = weight * (targetVoltage - v) / simpleSeriesResistance

IF t > duration + delay THEN

i = 0

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import VoltageClamp

variable = 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, gds_collector_=None, **kwargs_)
```

voltageClampTriple

extends *baseVoltageDepPointCurrent*

Voltage clamp with 3 clamp levels. Applies a variable current i (through **simpleSeriesResistance**) to try to keep parent cell at **conditioningVoltage** until time **delay**, **testingVoltage** until **delay + duration**, and **returnVoltage** afterwards. Only enabled if **active** = 1.

Parameters

active	Whether the voltage clamp is active (1) or inactive (0).	Dimensionless
condi-tioning-Voltage	Target voltage before time delay	<i>voltage</i>
delay	Delay before switching from conditioningVoltage to testingVoltage.	<i>time</i>
dura-tion	Duration to hold at testingVoltage.	<i>time</i>
return-Voltage	Target voltage after time duration	<i>voltage</i>
simple-Series-Resis-tance	Current will be calculated by the difference in voltage between the target and parent, divided by this value	<i>resistance</i>
testing-Voltage	Target voltage between times delay and delay + duration	<i>voltage</i>

Properties

weight (default: 1)		Dimensionless
----------------------------	--	---------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
----------	--	----------------

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepPointCurrent</i>)	<i>voltage</i>
----------	---	----------------

Event Ports

in	Note this is not used here. Will be removed in future	Direction: in
-----------	---	---------------

Dynamics

State Variables **i**: *current* (exposed as **i**)

On Events EVENT IN on port: **in**

On Conditions IF active = 1 AND t < delay THEN

i = weight * (conditioningVoltage - v) / simpleSeriesResistance

IF active = 1 AND t >= delay THEN

i = weight * (testingVoltage - v) / simpleSeriesResistance

IF active = 1 AND t > duration + delay THEN

i = weight * (returnVoltage - v) / simpleSeriesResistance

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import VoltageClampTriple

variable = 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, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<voltageClampTriple id="vClamp0" active="1" delay="50ms" duration="200ms"_
    conditioningVoltage="-70mV" testingVoltage="-50mV" returnVoltage="-70mV"_
    simpleSeriesResistance="1e6ohm"/>
```

10.1.8 Networks

Network descriptions for NeuroML 2. Describes *network* elements containing *populations* (potentially of type *populationList*, and so specifying a list of cell *locations*), *projections* (i.e. lists of *connections*) and *inputs*.

Original ComponentType definitions: [Networks.xml](#). Schema against which NeuroML based on these should be valid: [NeuroML_v2.2.xsd](#). Generated on 07/06/22 from [this commit](#). Please file any issues or questions at the [issue tracker](#) here.

network

extends [baseStandalone](#)

Network containing: *populations* (potentially of type *populationList*, and so specifying a list of cell *locations*); *projections* (with lists of *connections*) and/or *explicitConnections*; and *inputLists* (with lists of *inputs*) and/or *explicitInputs*. Note: often in NeuroML this will be of type *networkWithTemperature* if there are temperature dependent elements (e.g. ion channels).

Children list

regions		<i>region</i>
popula-tions		<i>basePopulation</i>
projec-tions		<i>projection</i>
synap-tic-Connec-tions		<i>explicitConnection</i>
electri-calProjec-tion		<i>electricalProjection</i>
contin-uous-Projec-tion		<i>continuousProjection</i>
explicit-Inputs		<i>explicitInput</i>
inputs		<i>inputList</i>

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Network

variable = 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, gds_collector_=
    None, **kwargs_)
```

Usage: XML

```
<network id="net1">
    <population id="iafPop1" component="iaf" size="1"/>
    <population id="iafPop2" component="iaf" size="1"/>
    <population id="iafPop3" component="iaf" size="1"/>
    <continuousProjection id="testLinearGradedConn" presynapticPopulation="iafPop1"
        postsynapticPopulation="iafPop2">
        <continuousConnection id="0" preCell="0" postCell="0" preComponent=
        "silent1" postComponent="gs1"/>
    </continuousProjection>
    <continuousProjection id="testGradedConn" presynapticPopulation="iafPop1"
        postsynapticPopulation="iafPop3">
        <continuousConnection id="0" preCell="0" postCell="0" preComponent=
        "silent2" postComponent="gs2"/>
    </continuousProjection>
    <explicitInput target="iafPop1[0]" input="pulseGen1" destination="synapses"/>
    <explicitInput target="iafPop1[0]" input="pulseGen2" destination="synapses"/>
    <explicitInput target="iafPop1[0]" input="pulseGen3" destination="synapses"/>
</network>
```

```
<network id="net2">
    <population id="hhPop1" component="hhcell" size="1" type="populationList">
        <instance id="0">
            <location x="0" y="0" z="0"/>
        </instance>
    </population>
    <population id="hhPop2" component="hhcell" size="1" type="populationList">
        <instance id="0">
            <location x="100" y="0" z="0"/>
        </instance>
    </population>
    <continuousProjection id="testGradedConn" presynapticPopulation="hhPop1"
        postsynapticPopulation="hhPop2">
        <continuousConnectionInstanceW id="0" preCell="..//hhPop1/0/hhcell"
        postCell="..//hhPop2/0/hhcell" preComponent="silent1" postComponent="gs1" weight="1"/
    <>
        </continuousProjection>
        <inputList id="i1" component="pulseGen1" population="hhPop1">
            <input id="0" target="..//hhPop1/0/hhcell" destination="synapses"/>
        </inputList>
    </network>
```

```
<network id="PyrCellNet">

    <population id="Population1" component="PyrCell" extracellularProperties=
    "extracellular" size="9">
    </population>
    <projection id="Proj1" presynapticPopulation="Population1"
        postsynapticPopulation="Population1" synapse="AMPA">
        </projection>
</network>
```

networkWithTemperature

extends [network](#)

Same as [network](#), but with an explicit **temperature** for temperature dependent elements (e.g. ion channels).

Parameters

temper- ature		<i>temperature</i>
--------------------------------	--	--------------------

basePopulation

extends [baseStandalone](#)

A population of multiple instances of a specific **component**, which anything which extends [baseCell](#).

Component References

compo- nent		<i>baseCell</i>
------------------------------	--	-----------------

Child list

notes		<i>notes</i>
annota- tion		<i>annotation</i>

Children list

prop- erty		<i>property</i>
-----------------------------	--	-----------------

population

extends [basePopulation](#)

A population of components, with just one parameter for the **size**, i.e. number of components to create. Note: quite often this is used with type= [populationList](#) which means the size is determined by the number of *instances* (with [locations](#)) in the list. The **size** attribute is still set, and there will be a validation error if this does not match the number in the list.

Parameters

size	Number of instances of this Component to create when the population is instantiated	Dimensionless
-------------	---	---------------

Usage: Python

Go to the [libNeuroML documentation](#)

```
from neuroml import Population

variable = 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, gds_collector=None, ↵
    **kwargs)
```

Usage: XML

```
<population id="iafPop1" component="iaf" size="1"/>
```

```
<population id="iafPop2" component="iaf" size="1"/>
```

```
<population id="iafPop3" component="iaf" size="1"/>
```

populationList

extends [basePopulation](#)

An explicit list of [*instances*](#) (with [*locations*](#)) of components in the population.

Text fields

size	Note: the size of the populationList to create is set by the number of explicitly defined instances. The size attribute is still set, and there will be a validation error if this does not match the number in the list.
-------------	---

Children list

in- stances		<i>instance</i>
------------------------	--	-----------------

instance

Specifies a single instance of a component in a *population* (placed at *location*).

Child list

location	<i>location</i>
-----------------	-----------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Instance

variable = Instance(id=None, i=None, j=None, k=None, location=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<instance id="0">
    <location x="0" y="0" z="0"/>
</instance>
```

```
<instance id="0">
    <location x="100" y="0" z="0"/>
</instance>
```

```
<instance id="0">
    <location x="0" y="0" z="0"/>
</instance>
```

location

Specifies the (x, y, z) location of a single *instance* of a component in a *population*.

Parameters

x	Dimensionless
y	Dimensionless
z	Dimensionless

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Location

variable = Location(x=None, y=None, z=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<location x="0" y="0" z="0"/>
```

```
<location x="100" y="0" z="0"/>
```

```
<location x="0" y="0" z="0"/>
```

region

Initial attempt to specify 3D region for placing cells. Work in progress...

Child list

rect-angularExtent		<i>rectangularExtent</i>
--------------------	--	--------------------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Region

variable = Region(neuro_lex_id=None, id=None, spaces=None, anytypeobjs_=None, gds_
                   collector_=None, **kwargs_)
```

rectangularExtent

For defining a 3D rectangular box.

Parameters

xLength	Dimensionless
xStart	Dimensionless
yLength	Dimensionless
yStart	Dimensionless
zLength	Dimensionless
zStart	Dimensionless

projection

Projection from one population, **presynapticPopulation** to another, **postsynapticPopulation**, through **synapse**. Contains lists of *connection* or *connectionWD* elements.

Paths

presynapticPopulation	
postsynapticPopulation	

Component References

synapse	<i>baseSynapse</i>
----------------	--------------------

Children list

connections		<i>connection</i>
connection- sWD		<i>connectionWD</i>

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Projection

variable = Projection(neuro_lex_id=None, id=None, presynaptic_population=None,
                     postsynaptic_population=None, synapse=None, connections=None, connection_wds=None,
                     gds_collector=None, **kwargs_)
```

Usage: XML

```
<projection id="Proj1" presynapticPopulation="Population1" postsynapticPopulation="Population1" synapse="AMPA">  
    </projection>
```

```
<projection id="internal1" presynapticPopulation="iafCells" postsynapticPopulation="iafCells" synapse="syn1">  
    <!--TODO: Fix! want to define synapse in here, so that multiple synapses per connection can be defined  
    <synapseComponent component="syn1"/>-->  
    <connection id="0" preCellId="..../iafCells/0/iaf" postCellId="..../iafCells/1/iaf"/>  
    </projection>
```

```
<projection id="internal2" presynapticPopulation="iafCells" postsynapticPopulation="iafCells" synapse="syn2">  
    <connection id="0" preCellId="..../iafCells/0/iaf" postCellId="..../iafCells/2/iaf"/>  
    </projection>
```

explicitConnection

Explicit event connection between components.

Text fields

targetPort	
------------	--

Paths

from	
to	

connection

Event connection directly between named components, which gets processed via a new instance of a **synapse** component which is created on the target component. Normally contained inside a *projection* element.

Text fields

destina- tion	
preFrac- tionAlong	
postFrac- tionAlong	
preSeg- mentId	
postSeg- mentId	

Paths

preCellId	
postCellId	

Usage: Python

Go to the *libNeuroML* documentation

```
from neuroml import Connection

variable = 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', gds_collector_=None, **kwargs_)
```

Usage: XML

```
<connection id="0" preCellId="..../iafCells/0/iaf" postCellId="..../iafCells/1/iaf"/>
```

```
<connection id="0" preCellId="..../iafCells/0/iaf" postCellId="..../iafCells/2/iaf"/>
```

```
<connection id="0" preCellId="..../pop0/0/MultiCompCell" postCellId="..../pop0/1/
    ↵MultiCompCell" preSegmentId="0" preFractionAlong="0.5" postSegmentId="0"_
    ↵postFractionAlong="0.5"/>
```

synapticConnection

extends *explicitConnection*

Explicit event connection between named components, which gets processed via a new instance of a **synapse** component which is created on the target component.

Text fields

destination	
--------------------	--

Paths

from	
to	

Component References

synapse	<i>baseSynapse</i>
----------------	--------------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SynapticConnection

variable = SynapticConnection(from_=None, to=None, synapse=None, destination=None, _  
    ↪gds_collector_=None, **kwargs_)
```

synapticConnectionWD

extends *synapticConnection*

Explicit event connection between named components, which gets processed via a new instance of a **synapse** component which is created on the target component, includes setting of **weight** and **delay** for the synaptic connection.

Parameters

delay	<i>time</i>
weight	Dimensionless

Paths

from	
to	

connectionWD

extends *connection*

Event connection between named components, which gets processed via a new instance of a synapse component which is created on the target component, includes setting of **weight** and **delay** for the synaptic connection.

Parameters

delay		<i>time</i>
weight		Dimensionless

Text fields

destina- tion	
preFrac- tionAlong	
postFrac- tionAlong	
preSeg- mentId	
postSeg- mentId	

Paths

preCellId	
postCellId	

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ConnectionWD

variable = 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, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<connectionWD id="0" preCellId="..../pop_EIF_cond_exp_isfa_ista[0]" postCellId="..../pop_
˓→target[0]" weight="0.01" delay="10ms"/>
```

```
<connectionWD id="0" preCellId="..../pop_EIF_cond_alpha_isfa_ista[0]" postCellId="..../
˓→pop_target[1]" weight="0.005" delay="20ms"/>
```

```
<connectionWD id="0" preCellId="..../pop_IF_curr_alpha[0]" postCellId="..../pop_target[2]
˓→" weight="1" delay="30ms"/>
```

electricalConnection

To enable connections between populations through gap junctions.

Component References

<code>synapse</code>	<code>gapJunction</code>
----------------------	--------------------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ElectricalConnection

variable = 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, gds_collector_=None,_
˓→**kwargs_)
```

Usage: XML

```
<electricalConnection id="0" preCell="0" postCell="0" synapse="gj1"/>
```

electricalConnectionInstance

To enable connections between populations through gap junctions. Populations need to be of type `populationList` and contain `instance` and `location` elements.

Text fields

<code>preFractionAlong</code>	
<code>postFractionAlong</code>	
<code>preSegment</code>	
<code>postSegment</code>	

Paths

<code>preCell</code>	
<code>postCell</code>	

Component References

<code>synapse</code>	<code>gapJunction</code>
----------------------	--------------------------

Usage: Python

Go to the [libNeuroML documentation](#)

```
from neuroml import ElectricalConnectionInstance

variable = 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, gds_collector_=None,
                                         **kwargs_)
```

Usage: XML

```
<electricalConnectionInstance id="0" preCell="..../iafPop1/0/iaf" postCell="..../iafPop2/
  0/iaf" preSegment="0" preFractionAlong="0.5" postSegment="0" postFractionAlong="0.5
  " synapse="gj1"/>
```

electricalConnectionInstanceW

extends [electricalConnectionInstance](#)

To enable connections between populations through gap junctions. Populations need to be of type [populationList](#) and contain [instance](#) and [location](#) elements. Includes setting of **weight** for the connection.

Parameters

weight	Dimensionless
---------------	---------------

Text fields

preFractionAlong	
postFractionAlong	
preSegment	
postSegment	

Paths

preCell	
postCell	

Usage: Python

Go to the [libNeuroML documentation](#)

```
from neuroml import ElectricalConnectionInstanceW

variable = 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, gds_collector_=None, **kwargs_)
```

electricalProjection

A projection between **presynapticPopulation** to another **postsynapticPopulation** through gap junctions.

Component References

presy- nap- ticPop- ulation		<i>population</i>
post- synap- ticPop- ulation		<i>population</i>

Children list

connec- tions		<i>electricalConnection</i>
connec- tionIn- stances		<i>electricalConnection- Instance</i>

Usage: Python

Go to the *libNeuroML* documentation

```
from neuroml import ElectricalProjection

variable = 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, gds_
    ↪collector_=None, **kwargs_)
```

Usage: XML

```
<electricalProjection id="testGJconn" presynapticPopulation="iafPop1" ↪
    ↪postsynapticPopulation="iafPop2">
    <electricalConnectionInstance id="0" preCell="..../iafPop1/0/iaf" postCell=
        ↪"../iafPop2/0/iaf" preSegment="0" preFractionAlong="0.5" postSegment="0" ↪
        ↪postFractionAlong="0.5" synapse="gj1"/>
</electricalProjection>
```

```
<electricalProjection id="testGJconn" presynapticPopulation="iafPop1" ↪
    ↪postsynapticPopulation="iafPop2">
    <electricalConnection id="0" preCell="0" postCell="0" synapse="gj1"/>
</electricalProjection>
```

continuousConnection

An instance of a connection in a *continuousProjection* between **presynapticPopulation** to another **postsynapticPopulation** through a **preComponent** at the start and **postComponent** at the end. Can be used for analog synapses.

Component References

pre-Component		<i>baseGradedSynapse</i>
post-Component		<i>baseGradedSynapse</i>

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ContinuousConnection

variable = 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, ↪
    ↪gds_collector=None, **kwargs_)
```

Usage: XML

```
<continuousConnection id="0" preCell="0" postCell="0" preComponent="silent1" ↪
    ↪postComponent="gs1"/>
```

```
<continuousConnection id="0" preCell="0" postCell="0" preComponent="silent2" ↪
    ↪postComponent="gs2"/>
```

continuousConnectionInstance

An instance of a connection in a *continuousProjection* between **presynapticPopulation** to another **postsynapticPopulation** through a **preComponent** at the start and **postComponent** at the end. Populations need to be of type *populationList* and contain *instance* and *location* elements. Can be used for analog synapses.

Text fields

<code>preFractionAlong</code>	
<code>postFractionAlong</code>	
<code>preSegment</code>	
<code>postSegment</code>	

Paths

<code>preCell</code>	
<code>postCell</code>	

Component References

<code>pre-Component</code>		<i>baseGradedSynapse</i>
<code>post-Component</code>		<i>baseGradedSynapse</i>

Usage: Python

Go to the [libNeuroML documentation](#)

```
from neuroml import ContinuousConnectionInstance

variable = 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,_
                                         gds_collector_=None, **kwargs_)
```

continuousConnectionInstanceW

extends [*continuousConnectionInstance*](#)

An instance of a connection in a [*continuousProjection*](#) between **presynapticPopulation** to another **postsynapticPopulation** through a **preComponent** at the start and **postComponent** at the end. Populations need to be of type [*populationList*](#) and contain [*instance*](#) and [*location*](#) elements. Can be used for analog synapses. Includes setting of **weight** for the connection.

Parameters

weight	Dimensionless
---------------	---------------

Text fields

preFrac-tionAlong	
postFrac-tionAlong	
preSeg-ment	
postSeg-ment	

Paths

preCell	
postCell	

Usage: Python

Go to the *libNeuroML* documentation

```
from neuroml import ContinuousConnectionInstanceW

variable = 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, gds_
                                         collector=None, **kwargs_)
```

Usage: XML

```
<continuousConnectionInstanceW id="0" preCell="..//hhPop1/0/hhcell" postCell="../
                                         ..//hhPop2/0/hhcell" preComponent="silent1" postComponent="gs1" weight="1"/>
```

continuousProjection

A projection between **presynapticPopulation** and **postsynapticPopulation** through components **preComponent** at the start and **postComponent** at the end of a *continuousConnection* or *continuousConnectionInstance*. Can be used for analog synapses.

Component References

<code>presynapticPopulation</code>		<i>population</i>
<code>postsynapticPopulation</code>		<i>population</i>

Children list

<code>connections</code>		<i>continuousConnection</i>
<code>connectionInstances</code>		<i>continuousConnectionInstance</i>

Usage: Python

Go to the *libNeuroML* documentation

```
from neuroml import ContinuousProjection

variable = 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, gds_collector=None, **kwargs_)
```

Usage: XML

```
<continuousProjection id="testLinearGradedConn" presynapticPopulation="iafPop1" postsynapticPopulation="iafPop2">
    <continuousConnection id="0" preCell="0" postCell="0" preComponent="silent1" postComponent="gs1"/>
</continuousProjection>
```

```
<continuousProjection id="testGradedConn" presynapticPopulation="iafPop1" postsynapticPopulation="iafPop3">
    <continuousConnection id="0" preCell="0" postCell="0" preComponent="silent2" postComponent="gs2"/>
</continuousProjection>
```

```
<continuousProjection id="testGradedConn" presynapticPopulation="hhPop1" postsynapticPopulation="hhPop2">
    <continuousConnectionInstanceW id="0" preCell="..../hhPop1/0/hhcell" postCell="..../hhPop2/0/hhcell" preComponent="silent1" postComponent="gs1" weight="1"/>
```

(continues on next page)

(continued from previous page)

```
</continuousProjection>
```

explicitInput

An explicit input (anything which extends *basePointCurrent*) to a target cell in a population.

Text fields

destina- tion	
sourcePort	
targetPort	

Paths

target	
--------	--

Component References

input		<i>basePointCurrent</i>
-------	--	-------------------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ExplicitInput

variable = ExplicitInput(target=None, input=None, destination=None, gds_collector_=  
..., **kwargs_)
```

Usage: XML

```
<explicitInput target="iafPop1[0]" input="pulseGen1" destination="synapses"/>
```

```
<explicitInput target="iafPop1[0]" input="pulseGen2" destination="synapses"/>
```

```
<explicitInput target="iafPop1[0]" input="pulseGen3" destination="synapses"/>
```

inputList

An explicit list of *inputs* to a **population**.

Text fields

population	
-------------------	--

Component References

component	<i>basePointCurrent</i>
------------------	-------------------------

Children list

inputs	<i>input</i>
---------------	--------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import InputList

variable = InputList(neuro_lex_id=None, id=None, populations=None, component=None,_
                     input=None, input_ws=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<inputList id="i1" component="pulseGen1" population="hhPop1">
    <input id="0" target="../hhPop1/0/hhcell" destination="synapses"/>
</inputList>
```

```
<inputList id="i1" component="pulseGen1" population="iafPop1">
    <input id="0" target="../iafPop1/0/iaf" destination="synapses"/>
</inputList>
```

```
<inputList id="i2" component="pulseGen2" population="iafPop2">
    <input id="0" target="../iafPop2/0/iaf" destination="synapses"/>
</inputList>
```

input

Specifies a single input to a **target**, optionally giving the **segmentId** (default 0) and **fractionAlong** the segment (default 0.5).

Text fields

segmentId	Optional specification of the segment to target, default 0
fractionA-long	Optional specification of the fraction along the specified segment, default 0.5
destination	

Paths

target	
--------	--

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Input

variable = Input(id=None, target=None, destination=None, segment_id=None, fraction_
    ↪along=None, extensiontype_=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<input id="0" target=".../hhPop1/0/hhcell" destination="synapses"/>
```

```
<input id="0" target=".../iafPop1/0/iaf" destination="synapses"/>
```

```
<input id="0" target=".../iafPop2/0/iaf" destination="synapses"/>
```

inputW

extends *input*

Specifies input lists. Can set **weight** to scale individual inputs.

Parameters

weight	Dimensionless
---------------	---------------

Text fields

destina- tion	
--------------------------	--

Paths

target	
---------------	--

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import InputW

variable = InputW(id=None, target=None, destination=None, segment_id=None, fraction_
    ↪along=None, weight=None, gds_collector_=None, **kwargs_)
```

10.1.9 PyNN

A number of ComponentType description of PyNN standard cells. All of the cells extend [basePyNNCell](#), and the synapses extend [basePynnSynapse](#).

Original ComponentType definitions: [PyNN.xml](#). Schema against which NeuroML based on these should be valid: [NeuroML_v2.2.xsd](#). Generated on 07/06/22 from [this](#) commit. Please file any issues or questions at the [issue tracker](#) here.

[basePyNNCell](#)

extends [baseCellMembPot](#)

Base type of any PyNN standard cell model. Note: membrane potential **v** has dimensions voltage, but all other parameters are dimensionless. This is to facilitate translation to and from PyNN scripts in Python, where these parameters have implicit units, see <http://neuralensemble.org/trac/PyNN/wiki/StandardModels>.

Parameters

cm		Dimensionless
i_offset		Dimensionless
tau_syn_E	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell	Dimensionless
tau_syn_I	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell	Dimensionless
v_init		Dimensionless

Constants

MSEC = 1ms		<i>time</i>
MVOLT = 1mV		<i>voltage</i>
NFARAD = 1nF		<i>capacitance</i>

Exposures

iSyn		<i>current</i>
v	Membrane potential (<i>from baseCellMembPot</i>)	<i>voltage</i>

Event Ports

spike	Spike event (<i>from baseSpikingCell</i>)	Direction: out
spike_in_E		Direction: in
spike_in_I		Direction: in

basePyNNlaFCell

extends [basePyNNCell](#)

Base type of any PyNN standard integrate and fire model.

Parameters

cm	(from basePyNNCell)	Dimensionless
i_offset	(from basePyNNCell)	Dimensionless
tau_m		Dimensionless
tau_refrac		Dimensionless
tau_syn_E	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
tau_syn_I	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
v_init	(from basePyNNCell)	Dimensionless
v_reset		Dimensionless
v_rest		Dimensionless
v_thresh		Dimensionless

Exposures

iSyn	(from basePyNNCell)	<i>current</i>
v	Membrane potential (from baseCellMembPot)	<i>voltage</i>

Event Ports

spike	Spike event (from baseSpikingCell)	Direction: out
spike_in_E	(from basePyNNCell)	Direction: in
spike_in_I	(from basePyNNCell)	Direction: in

basePyNNIaFCondCell

extends [basePyNNIaFCell](#)

Base type of conductance based PyNN IaF cell models.

Parameters

cm	(from basePyNNCell)	Dimensionless
e_rev_E	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell	Dimensionless
e_rev_I	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell	Dimensionless
i_offset	(from basePyNNCell)	Dimensionless
tau_m	(from basePyNNIaFCell)	Dimensionless
tau_refrac	(from basePyNNIaFCell)	Dimensionless
tau_syn_E	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
tau_syn_I	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
v_init	(from basePyNNCell)	Dimensionless
v_reset	(from basePyNNIaFCell)	Dimensionless
v_rest	(from basePyNNIaFCell)	Dimensionless
v_thresh	(from basePyNNIaFCell)	Dimensionless

Exposures

iSyn	(from basePyNNCell)	<i>current</i>
v	Membrane potential (from baseCellMembPot)	<i>voltage</i>

Event Ports

spike	Spike event (from baseSpikingCell)	Direction: out
spike_in_E	(from basePyNNCell)	Direction: in
spike_in_I	(from basePyNNCell)	Direction: in

IF_curr_alpha

extends [basePyNNIaFCell](#)

Leaky integrate and fire model with fixed threshold and alpha-function-shaped post-synaptic current.

Parameters

cm	(from basePyNNCell)	Dimensionless
i_offset	(from basePyNNCell)	Dimensionless
tau_m	(from basePyNNIaFCell)	Dimensionless
tau_refrac	(from basePyNNIaFCell)	Dimensionless
tau_syn_E	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
tau_syn_I	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
v_init	(from basePyNNCell)	Dimensionless
v_reset	(from basePyNNIaFCell)	Dimensionless
v_rest	(from basePyNNIaFCell)	Dimensionless
v_thresh	(from basePyNNIaFCell)	Dimensionless

Exposures

iSyn	(from basePyNNCell)	<i>current</i>
v	Membrane potential (from baseCellMembPot)	<i>voltage</i>

Event Ports

spike	Spike event (from baseSpikingCell)	Direction: out
spike_in_E	(from basePyNNCell)	Direction: in
spike_in_I	(from basePyNNCell)	Direction: in

Attachments

synapses	<i>baseSynapse</i>
-----------------	--------------------

Dynamics

State Variables **v**: *voltage* (exposed as **v**)

lastSpikeTime: *time*

On Start $v = v_{\text{init}} * \text{MVOLT}$

Derived Variables **iSyn** = synapses[*]->i(reduce method: add) (exposed as **iSyn**)

Regime: refractory (initial) On Entry

lastSpikeTime = t

$v = v_{\text{reset}} * \text{MVOLT}$

On Conditions

IF $t > \text{lastSpikeTime} + (\tau_{\text{refrac}} * \text{MSEC})$ THEN

TRANSITION to REGIME **integrating**

Regime: integrating (initial) On Conditions

```
IF v > v_thresh * MVOLT THEN
    EVENT OUT on port: spike
    TRANSITION to REGIME refractory
```

Time Derivatives

$$d v / dt = (MVOLT * ((i_offset/cm) + ((v_{rest} - (v/MVOLT)) / tau_m)) / MSEC) + (iSyn / (cm * NFARAD))$$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IF_curr_alpha

variable = 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,
                        gds_collector=None, **kwargs_)
```

Usage: XML

```
<IF_curr_alpha id="IF_curr_alpha" cm="1.0" i_offset="0.9" tau_m="20.0" tau_refrac="10.
                     tau_syn_E="0.5" tau_syn_I="0.5" v_init="-65" v_reset="-62.0" v_rest="-65.0" v_
                     thresh="-52.0"/>
```

IF_curr_exp

extends basePyNNIaFCell

Leaky integrate and fire model with fixed threshold and decaying-exponential post-synaptic current.

Parameters

cm	(from basePyNNCell)	Dimensionless
i_offset	(from basePyNNCell)	Dimensionless
tau_m	(from basePyNNIaFCell)	Dimensionless
tau_refrac	(from basePyNNIaFCell)	Dimensionless
tau_syn_E	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
tau_syn_I	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
v_init	(from basePyNNCell)	Dimensionless
v_reset	(from basePyNNIaFCell)	Dimensionless
v_rest	(from basePyNNIaFCell)	Dimensionless
v_thresh	(from basePyNNIaFCell)	Dimensionless

Exposures

iSyn	(from basePyNNCell)	<i>current</i>
v	Membrane potential (from baseCellMembPot)	<i>voltage</i>

Event Ports

spike	Spike event (from baseSpikingCell)	Direction: out
spike_in_E (from basePyNNCell)		Direction: in
spike_in_I (from basePyNNCell)		Direction: in

Attachments

synapses	<i>baseSynapse</i>
-----------------	--------------------

Dynamics

State Variables **v**: *voltage* (exposed as **v**)

lastSpikeTime: *time*

On Start **v** = **v_init** * MVOLT

Derived Variables **iSyn** = **synapses**[*]->i(reduce method: add) (exposed as **iSyn**)

Regime: refractory (initial) On Entry

lastSpikeTime = t

v = **v_reset** * MVOLT

On Conditions

IF t > lastSpikeTime + (tau_refrac*MSEC) THEN

TRANSITION to REGIME **integrating**

Regime: integrating (initial) On Conditions

IF v > **v_thresh** * MVOLT THEN

EVENT OUT on port: **spike**

TRANSITION to REGIME **refractory**

Time Derivatives

$d v / dt = (MVOLT * (((i_offset)/cm) + ((v_rest - (v/MVOLT)) / tau_m))/MSEC) + (iSyn / (cm * NFARAD))$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IF_curr_exp

variable = 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,
    gds_collector_=None, **kwargs_)
```

Usage: XML

```
<IF_curr_exp id="IF_curr_exp" cm="1.0" i_offset="1.0" tau_m="20.0" tau_refrac="8.0" tau_syn_E="5.0" tau_syn_I="5.0" v_init="-65" v_reset="-70.0" v_rest="-65.0" v_thresh="-50.0"/>
```

IF_cond_alpha

extends [basePyNNIaFCondCell](#)

Leaky integrate and fire model with fixed threshold and alpha-function-shaped post-synaptic conductance.

Parameters

cm	(from basePyNNCell)	Dimensionless
e_rev_E	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNIaFCondCell)	Dimensionless
e_rev_I	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNIaFCondCell)	Dimensionless
i_offset	(from basePyNNCell)	Dimensionless
tau_m	(from basePyNNIaFCell)	Dimensionless
tau_refrac	(from basePyNNIaFCell)	Dimensionless
tau_syn_E	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
tau_syn_I	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
v_init	(from basePyNNCell)	Dimensionless
v_reset	(from basePyNNIaFCell)	Dimensionless
v_rest	(from basePyNNIaFCell)	Dimensionless
v_thresh	(from basePyNNIaFCell)	Dimensionless

Exposures

iSyn	(from basePyNNCell)	<i>current</i>
v	Membrane potential (from baseCellMembPot)	<i>voltage</i>

Event Ports

spike	Spike event (from baseSpikingCell)	Direction: out
spike_in_E (from basePyNNCell)		Direction: in
spike_in_I (from basePyNNCell)		Direction: in

Attachments

synapses	<i>baseSynapse</i>
-----------------	--------------------

Dynamics

State Variables **v**: *voltage* (exposed as **v**)

lastSpikeTime: *time*

On Start **v** = **v_init** * MVOLT

Derived Variables **iSyn** = **synapses**[*]->i(reduce method: add) (exposed as **iSyn**)

Regime: refractory (initial) On Entry

lastSpikeTime = t

v = **v_reset** * MVOLT

On Conditions

IF t > lastSpikeTime + (tau_refrac*MSEC) THEN

TRANSITION to REGIME **integrating**

Regime: integrating (initial) On Conditions

IF v > **v_thresh** * MVOLT THEN

EVENT OUT on port: **spike**

TRANSITION to REGIME **refractory**

Time Derivatives

$$\frac{dv}{dt} = (\text{MVOLT} * (((i_{\text{offset}}) / \text{cm}) + ((v_{\text{rest}} - (v / \text{MVOLT})) / \tau_{\text{m}})) / \text{MSEC}) + (i_{\text{Syn}} / (\text{cm} * \text{NFARAD}))$$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IF_cond_alpha

variable = 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, gds_collector=None, **kwargs_)
```

Usage: XML

```
<IF_cond_alpha id="IF_cond_alpha" cm="1.0" e_rev_E="0.0" e_rev_I="-70.0" i_offset="0.9_
    ↪" tau_m="20.0" tau_refrac="5.0" tau_syn_E="0.3" tau_syn_I="0.5" v_init="-65" v_
    ↪reset="-65.0" v_rest="-65.0" v_thresh="-50.0"/>
```

```
<IF_cond_alpha id="silent_cell" cm="1.0" e_rev_E="0.0" e_rev_I="-70.0" i_offset="0"_
    ↪tau_m="20.0" tau_refrac="5.0" tau_syn_E="5" tau_syn_I="10" v_init="-65" v_reset="-
    ↪65.0" v_rest="-65.0" v_thresh="-50.0"/>
```

IF_cond_exp

extends [basePyNNIaFCondCell](#)

Leaky integrate and fire model with fixed threshold and exponentially-decaying post-synaptic conductance.

Parameters

cm	(from basePyNNCell)	Dimensionless
e_rev_E	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNIaFCondCell)	Dimensionless
e_rev_I	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNIaFCondCell)	Dimensionless
i_offset	(from basePyNNCell)	Dimensionless
tau_m	(from basePyNNIaFCell)	Dimensionless
tau_refrac	(from basePyNNIaFCell)	Dimensionless
tau_syn_E	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
tau_syn_I	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
v_init	(from basePyNNCell)	Dimensionless
v_reset	(from basePyNNIaFCell)	Dimensionless
v_rest	(from basePyNNIaFCell)	Dimensionless
v_thresh	(from basePyNNIaFCell)	Dimensionless

Exposures

iSyn	(from basePyNNCell)	<i>current</i>
v	Membrane potential (from baseCellMembPot)	<i>voltage</i>

Event Ports

spike	Spike event (from baseSpikingCell)	Direction: out
spike_in_E (from basePyNNCell)		Direction: in
spike_in_I (from basePyNNCell)		Direction: in

Attachments

synapses	<i>baseSynapse</i>
-----------------	--------------------

Dynamics

State Variables **v**: *voltage* (exposed as **v**)

lastSpikeTime: *time*

On Start **v** = **v_init** * MVOLT

Derived Variables **iSyn** = **synapses**[*]->i(reduce method: add) (exposed as **iSyn**)

Regime: refractory (initial) On Entry

lastSpikeTime = t

v = **v_reset** * MVOLT

On Conditions

IF t > lastSpikeTime + (tau_refrac*MSEC) THEN

TRANSITION to REGIME **integrating**

Regime: integrating (initial) On Conditions

IF v > **v_thresh** * MVOLT THEN

EVENT OUT on port: **spike**

TRANSITION to REGIME **refractory**

Time Derivatives

$\frac{d}{dt} v = (\text{MVOLT} * (((i_{\text{offset}})/\text{cm}) + ((v_{\text{rest}} - (v / \text{MVOLT})) / \tau_m)) / \text{MSEC}) + (i_{\text{Syn}} / (\text{cm} * \text{NFARAD}))$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IF_cond_exp

variable = 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, gds_collector=None, **kwargs_)
```

Usage: XML

```
<IF_cond_exp id="IF_cond_exp" cm="1.0" e_rev_E="0.0" e_rev_I="-70.0" i_offset="1.0" 
    tau_m="20.0" tau_refrac="5.0" tau_syn_E="5.0" tau_syn_I="5.0" v_init="-65" v_reset=
    "-68.0" v_rest="-65.0" v_thresh="-52.0"/>
```

EIF_cond_exp_isfa_ista

extends [basePyNNIaFCondCell](#)

Adaptive exponential integrate and fire neuron according to Brette R and Gerstner W (2005) with exponentially-decaying post-synaptic conductance.

Parameters

a		Dimensionless
b		Dimensionless
cm	(<i>from basePyNNCell</i>)	Dimensionless
delta_T		Dimensionless
e_rev_E	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (<i>from basePyNNIaFCondCell</i>)	Dimensionless
e_rev_I	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (<i>from basePyNNIaFCondCell</i>)	Dimensionless
i_offset	(<i>from basePyNNCell</i>)	Dimensionless
tau_m	(<i>from basePyNNIaFCell</i>)	Dimensionless
tau_refrac	(<i>from basePyNNIaFCell</i>)	Dimensionless
tau_syn_E	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (<i>from basePyNNCell</i>)	Dimensionless
tau_syn_I	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (<i>from basePyNNCell</i>)	Dimensionless
tau_w		Dimensionless
v_init	(<i>from basePyNNCell</i>)	Dimensionless
v_reset	(<i>from basePyNNIaFCell</i>)	Dimensionless
v_rest	(<i>from basePyNNIaFCell</i>)	Dimensionless
v_spike		Dimensionless
v_thresh	(<i>from basePyNNIaFCell</i>)	Dimensionless

Derived parameters

eif_threshold	Dimensionless
----------------------	---------------

Exposures

iSyn	(from basePyNNCell)	<i>current</i>
v	Membrane potential (from baseCellMembPot)	<i>voltage</i>
w		Dimensionless

Event Ports

spike	Spike event (from baseSpikingCell)	Direction: out
spike_in_E (from basePyNNCell)		Direction: in
spike_in_I (from basePyNNCell)		Direction: in

Attachments

synapses	baseSynapse
-----------------	-----------------------------

Dynamics

State Variables **v**: *voltage* (exposed as **v**)

w: Dimensionless (exposed as **w**)

lastSpikeTime: *time*

On Start **v** = **v_init** * MVOLT

w = 0

Derived Variables **iSyn** = **synapses**[*]->i(reduce method: add) (exposed as **iSyn**)

Conditional Derived Variables IF delta_T > 0 THEN

delta_I = delta_T * exp(((v / MVOLT) - v_thresh) / delta_T)

IF delta_T = 0 THEN

delta_I = 0

Regime: refractory (initial) On Entry

lastSpikeTime = t

v = **v_reset** * MVOLT

w = w+b

On Conditions

IF t > lastSpikeTime + (tau_refrac*MSEC) THEN

TRANSITION to REGIME **integrating**

Time Derivatives

$$\frac{d w}{dt} = (1 / \tau_w) * (a * ((v / \text{MVOLT}) - v_{\text{rest}}) - w) / \text{MSEC}$$

Regime: integrating (initial) On Conditions

IF $v > \text{eif_threshold} * \text{MVOLT}$ THEN

EVENT OUT on port: **spike**

TRANSITION to REGIME **refractory**

Time Derivatives

$$\frac{d v}{dt} = (\text{MVOLT} * ((-1 * ((v / \text{MVOLT}) - v_{\text{rest}}) + \delta_I) / \tau_m + (i_{\text{offset}} - w) / \text{cm}) / \text{MSEC}) + (i_{\text{Syn}} / (\text{cm} * \text{NFARAD}))$$

$$\frac{d w}{dt} = (1 / \tau_w) * (a * ((v / \text{MVOLT}) - v_{\text{rest}}) - w) / \text{MSEC}$$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import EIF_cond_exp_isfa_ista

variable = 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, gds_collector=None, **kwargs)
```

Usage: XML

```
<EIF_cond_exp_isfa_ista id="EIF_cond_exp_isfa_ista" a="0.0" b="0.0805" cm="0.281" ↪
    ↪ delta_T="2.0" e_rev_E="0.0" e_rev_I="-80.0" i_offset="0.6" tau_m="9.3667" tau_
    ↪ refrac="5" tau_syn_E="5.0" tau_syn_I="5.0" tau_w="144.0" v_init="-65" v_reset="-68.0" ↪
    ↪ v_rest="-70.6" v_spike="-40.0" v_thresh="-52.0"/>
```

EIF_cond_alpha_isfa_ista

extends [basePyNNIaFCondCell](#)

Adaptive exponential integrate and fire neuron according to Brette R and Gerstner W (2005) with alpha-function-shaped post-synaptic conductance.

Parameters

a		Dimensionless
b		Dimensionless
cm	(from basePyNNCell)	Dimensionless
delta_T		Dimensionless
e_rev_E	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNIaFCond-Cell)	Dimensionless
e_rev_I	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNIaFCond-Cell)	Dimensionless
i_offset	(from basePyNNCell)	Dimensionless
tau_m	(from basePyNNIaFCell)	Dimensionless
tau_refrac	(from basePyNNIaFCell)	Dimensionless
tau_syn_E	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
tau_syn_I	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
tau_w		Dimensionless
v_init	(from basePyNNCell)	Dimensionless
v_reset	(from basePyNNIaFCell)	Dimensionless
v_rest	(from basePyNNIaFCell)	Dimensionless
v_spike		Dimensionless
v_thresh	(from basePyNNIaFCell)	Dimensionless

Derived parameters

eif_threshold	Dimensionless
----------------------	---------------

Exposures

iSyn	(from basePyNNCell)	<i>current</i>
v	Membrane potential (from baseCellMembPot)	<i>voltage</i>
w		Dimensionless

Event Ports

spike	Spike event (from baseSpikingCell)	Direction: out
spike_in_E	(from basePyNNCell)	Direction: in
spike_in_I	(from basePyNNCell)	Direction: in

Attachments

synapses	<i>baseSynapse</i>
-----------------	--------------------

Dynamics**State Variables** **v**: *voltage* (exposed as **v**) **w**: Dimensionless (exposed as **w**) **lastSpikeTime**: *time***On Start** **v** = **v_init** * MVOLT **w** = 0**Derived Variables** **iSyn** = **synapses**[*]->i(reduce method: add) (exposed as **iSyn**)**Conditional Derived Variables** IF **delta_T** > 0 THEN **delta_I** = **delta_T** * exp(((**v** / MVOLT) - **v_thresh**) / **delta_T**) IF **delta_T** = 0 THEN **delta_I** = 0**Regime: refractory (initial) On Entry** **lastSpikeTime** = **t** **v** = **v_reset** * MVOLT **w** = **w** + **b****On Conditions** IF **t** > **lastSpikeTime** + (**tau_refrac** * MSEC) THEN TRANSITION to REGIME **integrating****Time Derivatives** d **w** /dt = (1 / **tau_w**) * (a * ((**v** / MVOLT) - **v_rest**) - **w**) / MSEC**Regime: integrating (initial) On Conditions** IF **v** > **eif_threshold** * MVOLT THEN EVENT OUT on port: **spike** TRANSITION to REGIME **refractory****Time Derivatives** d **v** /dt = (MVOLT * ((-1 * ((**v** / MVOLT) - **v_rest**) + **delta_I**) / **tau_m** + (**i_offset** - **w**) / **cm**) / MSEC) + (**iSyn** / (**cm** * NFARAD)) d **w** /dt = (1 / **tau_w**) * (a*((**v**/MVOLT)-**v_rest**) - **w**) /MSEC

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import EIF_cond_alpha_isfa_ista

variable = 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, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<EIF_cond_alpha_isfa_ista id="EIF_cond_alpha_isfa_ista" a="0.0" b="0.0805" cm="0.281"_
    ↪delta_T="0" e_rev_E="0.0" e_rev_I="-80.0" i_offset="0.6" tau_m="9.3667" tau_refrac=
    ↪"5" tau_syn_E="5.0" tau_syn_I="5.0" tau_w="144.0" v_init="-65" v_reset="-68.0" v_
    ↪rest="-70.6" v_spike="-40.0" v_thresh="-52.0"/>
```

HH_cond_exp

extends [basePyNNCell](#)

Single-compartment Hodgkin-Huxley-type neuron with transient sodium and delayed-rectifier potassium currents using the ion channel models from Traub.

Parameters

cm	(from basePyNNCell)	Dimensionless
e_rev_E		Dimensionless
e_rev_I		Dimensionless
e_rev_K		Dimensionless
e_rev_Na		Dimensionless
e_rev_leak		Dimensionless
g_leak		Dimensionless
gbar_K		Dimensionless
gbar_Na		Dimensionless
i_offset	(from basePyNNCell)	Dimensionless
tau_syn_E	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
tau_syn_I	This parameter is never used in the NeuroML2 description of this cell! Any synapse producing a current can be placed on this cell (from basePyNNCell)	Dimensionless
v_init	(from basePyNNCell)	Dimensionless
v_offset		Dimensionless

Exposures

h		Dimensionless
iSyn	(from <code>basePyNNCell</code>)	<i>current</i>
m		Dimensionless
n		Dimensionless
v	Membrane potential (from <code>baseCellMembPot</code>)	<i>voltage</i>

Event Ports

spike	Spike event (from <code>baseSpikingCell</code>)	Direction: out
spike_in_E (from <code>basePyNNCell</code>)		Direction: in
spike_in_I (from <code>basePyNNCell</code>)		Direction: in

Attachments

synapses	<i>baseSynapse</i>
-----------------	--------------------

Dynamics

State Variables **v**: *voltage* (exposed as **v**)

m: Dimensionless (exposed as **m**)

h: Dimensionless (exposed as **h**)

n: Dimensionless (exposed as **n**)

On Start **v** = **v_init** * MVOLT

Derived Variables **iSyn** = `synapses[*]->i(reduce method: add)` (exposed as **iSyn**)

iLeak = $g_{\text{leak}} * (e_{\text{rev_leak}} - (v / \text{MVOLT}))$

iNa = $g_{\text{bar_Na}} * (m * m * m) * h * (e_{\text{rev_Na}} - (v / \text{MVOLT}))$

iK = $g_{\text{bar_K}} * (n * n * n * n) * (e_{\text{rev_K}} - (v / \text{MVOLT}))$

iMemb = **iLeak** + **iNa** + **iK** + **i_offset**

alpham = $0.32 * (13 - (v / \text{MVOLT}) + v_{\text{offset}}) / (\exp((13 - (v / \text{MVOLT}) + v_{\text{offset}}) / 4.0) - 1)$

betam = $0.28 * ((v / \text{MVOLT}) - v_{\text{offset}} - 40) / (\exp(((v / \text{MVOLT}) - v_{\text{offset}} - 40) / 5.0) - 1)$

alphah = $0.128 * \exp((17 - (v / \text{MVOLT}) + v_{\text{offset}}) / 18.0)$

betah = $4.0 / (1 + \exp((40 - (v / \text{MVOLT}) + v_{\text{offset}}) / 5))$

alphan = $0.032 * (15 - (v / \text{MVOLT}) + v_{\text{offset}}) / (\exp((15 - (v / \text{MVOLT}) + v_{\text{offset}}) / 5) - 1)$

betan = $0.5 * \exp((10 - (v / \text{MVOLT}) + v_{\text{offset}}) / 40)$

Time Derivatives $d \mathbf{v} / dt = (\text{MVOLT} * (iMemb / cm) / \text{MSEC}) + (iSyn / (cm * \text{NFARAD}))$

$d \mathbf{m} / dt = (\text{alpham} * (1 - m) - \text{betam} * m) / \text{MSEC}$

$d \mathbf{h} / dt = (\text{alphah} * (1 - h) - \text{betah} * h) / \text{MSEC}$

$$\frac{dn}{dt} = (\text{alphan} * (1 - n) - \text{betan} * n) / \text{MSEC}$$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import HH_cond_exp

variable = 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, gds_collector_ ↵
    ↵=None, **kwargs_)
```

Usage: XML

```
<HH_cond_exp id="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"/>
```

basePynnSynapse

extends [baseVoltageDepSynapse](#)

Base type for all PyNN synapses. Note, the current **I** produced is dimensionless, but it requires a membrane potential **v** with dimension voltage.

Parameters

tau_syn	Dimensionless
----------------	---------------

Constants

MSEC = 1ms	<i>time</i>
MVOLT = 1mV	<i>voltage</i>
NAMP = 1nA	<i>current</i>

Exposures

i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>
----------	--	----------------

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepSynapse</i>)	<i>voltage</i>
----------	--	----------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
-----------	-----------------------------	---------------

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BasePynnSynapse

variable = BasePynnSynapse(neuro_lex_id=None, id=None, metaid=None, notes=None,_
                           properties=None, annotation=None, tau_syn=None, extensiontype_=None, gds_collector_=
                           _=None, **kwargs_)
```

expCondSynapse

extends [basePynnSynapse](#)

Conductance based synapse with instantaneous rise and single exponential decay (with time constant tau_syn).

Parameters

e_rev		Dimensionless
tau_syn	(<i>from basePynnSynapse</i>)	Dimensionless

Properties

weight (default: 1)		Dimensionless
----------------------------	--	---------------

Exposures

g		Dimensionless
i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepSynapse</i>)	<i>voltage</i>
----------	--	----------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
-----------	-----------------------------	---------------

Dynamics

State Variables **g**: Dimensionless (exposed as **g**)

On Events EVENT IN on port: **in**

$$g = g + \text{weight}$$

Derived Variables **i** = $g * (e_{\text{rev}} - (v/\text{MVOLT})) * \text{NAMP}$ (exposed as **i**)

Time Derivatives $d g / dt = -g / (\tau_{\text{syn}} \text{MSEC})$

Usage: Python

Go to the *libNeuroML* documentation

```
from neuroml import ExpCondSynapse

variable = ExpCondSynapse(neuro_lex_id=None, id=None, metaid=None, notes=None,
                           properties=None, annotation=None, tau_syn=None, e_rev=None, gds_collector_=None,
                           **kwargs_)
```

Usage: XML

```
<expCondSynapse id="syn1" tau_syn="5" e_rev="0"/>
```

expCurrSynapse

extends *basePynnSynapse*

Current based synapse with instantaneous rise and single exponential decay (with time constant *tau_syn*).

Parameters

tau_syn	(from basePynnSynapse)	Dimensionless
----------------	------------------------	---------------

Properties

weight (default: 1)	Dimensionless
----------------------------	---------------

Exposures

i	The total (usually time varying) current produced by this ComponentType (from basePointCurrent)	<i>current</i>
----------	---	----------------

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (from baseVoltageDepSynapse)	<i>voltage</i>
----------	---	----------------

Event Ports

in	(from baseSynapse)	Direction: in
-----------	--------------------	---------------

Dynamics

State Variables **I**: Dimensionless

On Events EVENT IN on port: **in**

$$I = I + \text{weight}$$

Derived Variables **i** = $I * \text{NAMP}$ (exposed as **i**)

Time Derivatives $d I / dt = -I / (\tau_{\text{syn}} * \text{MSEC})$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ExpCurrSynapse

variable = ExpCurrSynapse(neuro_lex_id=None, id=None, metaid=None, notes=None, _properties=None, annotation=None, tau_syn=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<expCurrSynapse id="syn3" tau_syn="5"/>
```

alphaCondSynapseextends [basePynnSynapse](#)

Alpha synapse: rise time and decay time are both tau_syn. Conductance based synapse.

Parameters

e_rev		Dimensionless
tau_syn	(from basePynnSynapse)	Dimensionless

Properties

weight (default: 1)		Dimensionless
----------------------------	--	---------------

Exposures

A		Dimensionless
g		Dimensionless
i	The total (usually time varying) current produced by this ComponentType (from basePointCurrent)	<i>current</i>

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (from baseVoltageDepSynapse)	<i>voltage</i>
----------	--	----------------

Event Ports

in	(from baseSynapse)	Direction: in
-----------	-------------------------------------	---------------

Dynamics

State Variables **g**: Dimensionless (exposed as **g**)

A: Dimensionless (exposed as **A**)

On Events EVENT IN on port: **in**

$A = A + \text{weight}$

Derived Variables **i** = $g * (e_{\text{rev}} - (v/\text{MVOLT})) * \text{NAMP}$ (exposed as **i**)

Time Derivatives $d g / dt = (2.7182818A - g) / (\tau_{\text{syn}} \text{MSEC})$

$d A / dt = -A / (\tau_{\text{syn}} \text{MSEC})$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import AlphaCondSynapse

variable = AlphaCondSynapse(neuro_lex_id=None, id=None, metaid=None, notes=None, _  
    properties=None, annotation=None, tau_syn=None, e_rev=None, gds_collector_=None, _  
    **kwargs_)
```

Usage: XML

```
<alphaCondSynapse id="syn2" tau_syn="5" e_rev="0"/>
```

alphaCurrSynapse

extends [basePynnSynapse](#)

Alpha synapse: rise time and decay time are both tau_syn. Current based synapse.

Parameters

tau_syn (from basePynnSynapse)	Dimensionless
--	---------------

Properties

weight (default: 1)	Dimensionless
----------------------------	---------------

Exposures

A		<i>current</i>
i	The total (usually time varying) current produced by this ComponentType (<i>from basePointCurrent</i>)	<i>current</i>

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed (<i>from baseVoltageDepSynapse</i>)	<i>voltage</i>
----------	--	----------------

Event Ports

in	(<i>from baseSynapse</i>)	Direction: in
-----------	-----------------------------	---------------

Dynamics

State Variables **I**: Dimensionless

A: Dimensionless (exposed as **A**)

On Events EVENT IN on port: **in**

$$A = A + \text{weight}$$

Derived Variables **i** = $I * \text{NAMP}$ (exposed as **i**)

Time Derivatives $d I / dt = (2.7182818A - I) / (\tau_{synMSEC})$

$$d A / dt = -A / (\tau_{syn} * MSEC)$$

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import AlphaCurrSynapse

variable = AlphaCurrSynapse(neuro_lex_id=None, id=None, metaid=None, notes=None, _properties=None, annotation=None, tau_syn=None, gds_collector_=None, **kwargs_)
```

Usage: XML

```
<alphaCurrSynapse id="syn4" tau_syn="5"/>
```

SpikeSourcePoisson

extends [baseSpikeSource](#)

Spike source, generating spikes according to a Poisson process.

Parameters

dura-tion		<i>time</i>
rate		<i>per_time</i>
start		<i>time</i>

Derived parameters

end		<i>time</i>
------------	--	-------------

Constants

LONG_TIME = 1e9hour		<i>time</i>
SMALL_TIME = 1e-9ms		<i>time</i>

Exposures

isi		<i>time</i>
tnex-tIdeal		<i>time</i>
tnex-tUsed		<i>time</i>
tsince	Time since the last spike was emitted (<i>from baseSpikeSource</i>)	<i>time</i>

Event Ports

in		Direction: in
spike	Port on which spikes are emitted (<i>from baseSpikeSource</i>)	Direction: out

Dynamics

State Variables **tsince**: *time* (exposed as **tsince**)

tnextIdeal: *time* (exposed as **tnextIdeal**)

tnextUsed: *time* (exposed as **tnextUsed**)

isi: *time* (exposed as **isi**)

On Start **isi** = start - log(random(1))/rate

tsince = 0

tnextIdeal = isi + H(((isi) - (start+duration))/duration)*LONG_TIME

tnextUsed = tnextIdeal

On Conditions IF t > tnextUsed THEN

isi = -1 * log(random(1))/rate

tnextIdeal = (tnextIdeal+isi) + H(((tnextIdeal+isi) - (start+duration))/duration)*LONG_TIME

tnextUsed = tnextIdeal*H((tnextIdeal-t)/t) + (t+SMALL_TIME)*H((t-tnextIdeal)/t)

tsince = 0

EVENT OUT on port: **spike**

Time Derivatives d **tsince** /dt = 1

d **tnextUsed** /dt = 0

d **tnextIdeal** /dt = 0

Usage: Python

Go to the *libNeuroML* documentation

```
from neuroml import SpikeSourcePoisson

variable = SpikeSourcePoisson(neuro_lex_id=None, id=None, metaid=None, notes=None,
    properties=None, annotation=None, start=None, duration=None, rate=None, gds_
    collector=None, **kwargs_)
```

Usage: XML

```
<SpikeSourcePoisson id="spikes1" start="50ms" duration="400ms" rate="50Hz"/>
```

```
<SpikeSourcePoisson id="spikes2" start="50ms" duration="300ms" rate="80Hz"/>
```

10.1.10 Simulation

Specification of the LEMS Simulation element which is normally used to define simulations of NeuroML2 files. Note: not actually part of NeuroML v2, but this is required by much of the NeuroML toolchain for defining Simulations (which NeuroML model to use and how long to run for), as well as what to *Display* and what to save in *OutputFiles*.

Original ComponentType definitions: [Simulation.xml](#). Schema against which NeuroML based on these should be valid: [NeuroML_v2.2.xsd](#). Generated on 07/06/22 from [this commit](#). Please file any issues or questions at the [issue tracker here](#).

Simulation

The main element in a LEMS Simulation file. Defines the **length** of simulation, the timestep (dt) **step** and an optional **seed** to use for stochastic elements, as well as *Displays*, *OutputFiles* and *EventOutputFiles* to record. Specifies a **target** component to run, usually the id of a *network*.

Parameters

length	Duration of the simulation run	<i>time</i>
step	Time step (dt) to use in the simulation	<i>time</i>

Text fields

seed	The seed to use in the random number generator for stochastic entities
-------------	--

Component References

target		schema:component
---------------	--	------------------

Children list

displays		<i>Display</i>
outputs		<i>OutputFile</i>
events		<i>EventOutputFile</i>

Dynamics

State Variables **t:** *time*

Display

Details of a display to generate (usually a set of traces given by *Lines* in a newly opened window) on completion of the simulation.

Parameters

timeScale	A scaling of the time axis, e.g. 1ms means display in milliseconds. Note: all quantities are recorded in SI units	<i>time</i>
xmax	The maximum value on the x axis (i.e time variable) of the display	Dimensionless
xmin	The minimum value on the x axis (i.e time variable) of the display	Dimensionless
ymax	The maximum value on the x axis of the display	Dimensionless
ymin	The minimum value on the y axis of the display	Dimensionless

Text fields

title	The title of the display, e.g. to use for the window
--------------	--

Children list

lines	<i>Line</i>
--------------	-------------

Line

Specification of a single time varying **quantity** to plot on the *Display*. Note that all quantities are handled internally in LEMS in SI units, and so a **scale** should be used if it is to be displayed in other units.

Parameters

scale	A scaling factor to DIVIDE the quantity by. Can be dimensional, so using scale=1mV means a value of -0.07V is displayed as -70. Alternatively, scale=0.001 would achieve the same thing.	<i>Dimensions</i>
timeScale	An optional scaling of the time axis, e.g. 1ms means display in milliseconds. Note: if present, this overrides timeScale from <i>Display</i>	<i>Dimensions</i>

Text fields

color	A hex string for the color to display the trace for this quantity, e.g. #aa33ff
--------------	---

Paths

quantity	Path to the quantity to display, see see https://docs.neuroml.org/Userdocs/Paths.html .
-----------------	--

OutputFile

A file in which to save recorded values from the simulation.

Text fields

path	Optional path to the directory in which to store the file
fileName	Name of the file to generate. Can include a relative path (from the LEMS Simulation file location).

Children list

output-Column		<i>OutputColumn</i>
----------------------	--	---------------------

OutputColumn

Specification of a single time varying **quantity** to record during the simulation. Note that all quantities are handled internally in LEMS in SI units, and so the value for the quantity in the file (as well as time) will be in SI units.

Paths

quantity	Path to the quantity to save, see see https://docs.neuroml.org/Userdocs/Paths.html . Note that all quantities are saved in SI units.
-----------------	---

EventOutputFile

A file in which to save event information (e.g. spikes from cells in a population) in a specified **format**.

Text fields

path	Optional path to the directory in which to store the file
fileName	Name of the file to generate. Can include a relative path (from the LEMS Simulation file location).
format	Takes values TIME_ID or ID_TIME, depending on the preferred order of the time or event id (from <i>EventSelection</i>) in each row of the file

Children list

eventSelection	<i>EventSelection</i>
-----------------------	-----------------------

EventSelection

A specific source of events with an associated **id**, which will be recorded inside the file specified in the parent *EventOutputFile*. The attribute **select** should point to a cell inside a *population* (e.g. `hhpop[0]`, see <https://docs.neuroml.org/Userdocs/Paths.html>), and the **eventPort** specifies the port for the emitted events, which usually has id: spike. Note: the **id** used on this element (and appearing in the file alongside the event time) can be different from the id/index of the cell in the population.

Text fields

eventPort	The port on the cell which generates the events, usually: spike
------------------	---

Paths

select	The cell which will be emitting the events
---------------	--

10.2 NeuroML v1

Warning: NeuroML v1.x is deprecated. This page is maintained for archival purposes only.

Please use [NeuroML v2](#).

There are three Levels of compliance to the NeuroML v1 specifications:

10.2.1 Level 1

- Metadata v1.8.1
- MorphML v1.8.1

Any Level 1 NeuroML v1 file will also be compliant to [this schema](#).

10.2.2 Level 2

- Biophysics v1.8.1
- ChannelML v1.8.1

Any Level 1 or Level 2 NeuroML v1 file will also be compliant to [this schema](#).

10.2.3 Level 3

- NetworkML v1.8.1

Any Level 1 or Level 2 or Level 3 NeuroML v1 file will also be compliant to [this schema](#).

These files are archived in [this GitHub repository](#).

USING NEUROML 2 AND LEMS

While the *tutorials* cover many of the key points of using LEMS with NeuroML, there are some points which require further explanation:

- *What are the conventions/best practices to follow in naming NeuroML/LEMS files/elements?*
- *How are units and dimensions handled in NeuroML and LEMS?*
- *How do I use a LEMS Simulation file to specify how to execute a NeuroML model?*
- *How can I extend NeuroML model to include new types using LEMS?*
- *What is the correct format/usage of paths and quantities in NeuroML and LEMS?*

11.1 Conventions

This page documents various conventions in use in NeuroML.

In general, please prefer underscores `_` instead of spaces wherever possible, in filenames and ids.

11.1.1 Component IDs: NmlId

Some Components take an `id` parameter to set an ID for them. They can then be referred to using their IDs when constructing paths and so on.

IDs in NeuroML are strings and have certain constraints:

- they **must** start with an alphabet (either small or capital) or an underscore
- they may include alphabets, both small and capital letters, numbers and underscores

IDs are also checked during validation, so if an ID does not follow these constraints, the validation will throw an error.

11.1.2 File naming

When naming different NeuroML files, we suggest the following suffixes:

- `channel.nml` for NeuroML files describing ion channels, for example: `Na.channel.nml`
- `cell.nml` for NeuroML files describing cells, for example: `hh.cell.nml`
- `synapse.nml` for NeuroML files describing synapses, for example: `AMPA.synapse.nml`
- `net.nml` for NeuroML files describing networks of cells, for example: `excitatory.net.nml`

For LEMS files that describe simulations of NeuroML models (“*LEMS Simulation files*”), we suggest that:

- file names start with the `LEMS_` prefix,
- file names end in `.xml`

For example `LEMS_HH_Simulation.xml`.

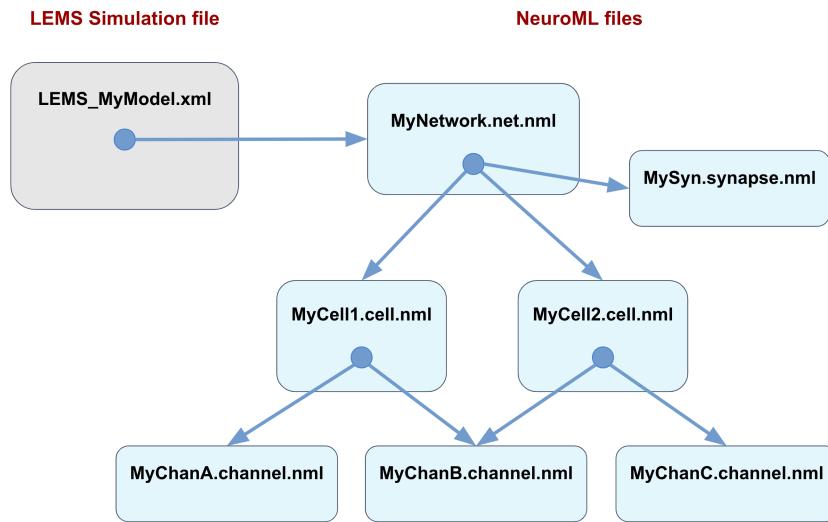


Fig. 11.1: Typical organisation for a NeuroML simulation. The main NeuroML model is specified in a file with the network (`*.net.nml`), which can include/point to files containing individual synapses (`*.synapse.nml`) or cell files (`*.cell.nml`). If the latter are conductance based, they may include external channel files (`*.channel.nml`). The main LEMS Simulation file only needs to include the network file, and tools for running simulations of the model refer to just this LEMS file. Exceptions to these conventions are frequent and simulations will run perfectly well with all the elements inside the main LEMS file, but using this scheme will maximise reusability of model elements.

11.1.3 Neuron segments

When naming segments in multi-compartmental neuron models, we suggest the following prefixes:

- `axon_` for axonal segments
- `dend_` for dendritic segments
- `soma_` for somatic segments

There are 3 specific recommended names for segment groups which contain **ALL** of the somatic, dendritic or axonal segments

- `axon_group` for the group of all axonal segments
- `dend_group` for the group of all dendritic segments
- `soma_group` for the group of all somatic segments

Ideally every segment should be a member of one and only one of these groups.

11.2 Units and dimensions

Support for dimensional quantities is a fundamental (and essential) feature of NeuroML, backed up by support for units and dimensions in LEMS.

The basic rules are:

- specify the **dimensions** of quantities in LEMS
- use compatible **units** defined in the NeuroML schema in NeuroML models.

The main motivation for this is that fundamental expressions for defining a model are independent of any particular units. For example, Ohm's law, $\mathbf{V} = \mathbf{I} * \mathbf{R}$ relates to quantities with dimensions voltage, current and resistance, not millivolts, picoamps, ohms, etc.

Users can therefore use a wide range of commonly used units for each dimension defined in the *standard unit and dimension definitions* of NeuroML 2 without worrying about conversion factors.

Additionally, please keep in mind that:

- all quantities are saved and *recorded* in SI Units
- when plotting data using NeuroML/LEMS using the *Line* component, users can use the `scale` parameter to convert quantities to other units.

11.3 Paths

Since NeuroMLv2 and LEMS are both XML based, entities in models and simulations must be referred to using *paths*. This page documents how paths can be constructed, and how they can be used to refer to entities in NeuroML/LEMS based models and simulations (e.g. in a *LEMS Simulation file*).

11.3.1 Constructing paths

Paths start from any element and ascend/descend to refer to the entity that is to be referenced.

The `.` and `..` path constructs are special constructs that mean “the current node” and “the parent node” respectively.

For example, in the following block of code, based on the *Izhikevich network example*, a network is defined in NeuroML with 2 populations:

```
<network id="IzNet">
    <population id="IzPop0" component="iz2007RS0" size="5">
        <property tag="color" value="0 0 .8"/>
    </population>
    <populationList id="IzPop1" component="iz2007RS0">
        <property tag="color" value=".8 0 0"/>
        <instance id=0>
            <location x="0" y="0" z="0" />
        </instance>
        <instance id=1>
            <location x="1" y="0" z="0" />
        </instance>
        <instance id=2>
            <location x="2" y="0" z="0" />
        </instance>
        <instance id=3>
```

(continues on next page)

(continued from previous page)

```

        <location x="3" y="0" z="0" />
    </instance>
    <instance id=4>
        <location x="4" y="0" z="0" />
    </instance>
</populationList>
<projection id="proj" presynapticPopulation="IzPop0" postsynapticPopulation=
    "IzPop1" synapse="syn0">
    <connection id="0" preCellId="..../IzPop0[0]" postCellId="..../IzPop1/0"/>
    <connection id="1" preCellId="..../IzPop0[0]" postCellId="..../IzPop1/1"/>
    <connection id="2" preCellId="..../IzPop0[0]" postCellId="..../IzPop1/2"/>
    ...
</projection>
<explicitInput target="IzPop0[0]" input="pg_0"/>
<explicitInput target="IzPop0[1]" input="pg_1"/>
<explicitInput target="IzPop0[2]" input="pg_2"/>
<explicitInput target="IzPop0[3]" input="pg_3"/>
<explicitInput target="IzPop0[4]" input="pg_4"/>
</network>
</neuroml>

```

Here, in the `explicitInput` node, we need to refer to neurons of the `IzPop0` population node. Since `explicitInput` and `population` are *siblings* (both have the `IzNet` network as *parent*), they are at the same *level*. Therefore, in `explicitInput`, one can refer directly to `IzPop0`.

The `projection` and `population` nodes are also *siblings* and therefore are at the same level. So, in the `projection` tag also, we can refer to the `population` nodes directly. The `connection` nodes, however, are *children* of the `projection` node. Therefore, for the `connection` nodes, the `population` nodes are at the *parent* level, and we must use `..../IzPop0` to refer to them.

`..../IzPop0` means “go up one level to the parent level (to `projection`) and then refer to `IzPop0`”. `..../` can be used as many times as required and wherever required in the path. For example, `..../..../..../` would mean “go up three levels”.

Additionally, when constructing the path:

- for any `child` elements, the *name* of the element should be used in the path
- for `children` elements, the *id* of the element should be used in the path (because all children will have the same name)
- for elements that have a `size` attribute like `population`, individual elements should be referred to using the `[]` operator (these elements instantiate multiple instances of the provided component without the user having to specify each instance explicitly)

Note the difference in the paths used for the `population` and `populationList` components. Whereas `population` uses the `size` attribute to instantiate multiple instances of the cell, each instance must be explicitly mentioned when using the `populationList` component. This is because unlike `population`, in `populationList` all `instance` elements are `children` elements.

Helper functions in pyNeuroML

Note: These functions require *pyNeuroML* version 0.5.18+, and *pylems* version 0.5.8+.

From version 0.5.18, *pyNeuroML* includes the `list_recording_paths_for_exposures` helper function that can list the exposures and their recordable paths from a NeuroML 2 model:

```
>>> import pynml.pynml
>>> help(pynml.list_recording_paths_for_exposures)

Help on function list_recording_paths_for_exposures in module pynml.pynml:

list_recording_paths_for_exposures(nml_doc_fn, substring='', target='')
    List the recording path strings for exposures.

This wraps around `lems.model.list_recording_paths` to list the recording
paths in the given NeuroML2 model. The only difference between the two is
that the `lems.model.list_recording_paths` function is not aware of the
NeuroML2 component types (since it's for any LEMS models in general), but
this one is.
```

It can be run on the example *Izhikevich network example*:

```
>>> pynml.list_recording_paths_for_exposures("izhikevich2007_network.nml", substring="",
    ↪, target="IzNet")
['IzNet/IzPop0[0]/iMemb',
 'IzNet/IzPop0[0]/iSyn',
 'IzNet/IzPop0[0]/u',
 'IzNet/IzPop0[0]/v',
 'IzNet/IzPop0[1]/iMemb',
 'IzNet/IzPop0[1]/iSyn',
 'IzNet/IzPop0[1]/u',
 'IzNet/IzPop0[1]/v',
 'IzNet/IzPop0[2]/iMemb',
 'IzNet/IzPop0[2]/iSyn',
 'IzNet/IzPop0[2]/u',
 'IzNet/IzPop0[2]/v',
 'IzNet/IzPop0[3]/iMemb',
 'IzNet/IzPop0[3]/iSyn',
 'IzNet/IzPop0[3]/u',
 'IzNet/IzPop0[3]/v',
 'IzNet/IzPop0[4]/iMemb',
 'IzNet/IzPop0[4]/iSyn',
 'IzNet/IzPop0[4]/u',
 'IzNet/IzPop0[4]/v',
 'IzNet/IzPop1[0]/iMemb',
 ..
]
```

Note that this function parses the model description only, not the built simulation description. Therefore, it will not necessarily list the complete list of paths. Also worth noting is that since it parses and iterates over the expanded representation of the model, it can be slow and return long lists of results on larger models. It is therefore, best to use this with the `substring` option to narrow its scope.

An associated helper function `list_exposures` is also available:

```
>>> import pyneuroml.pynml
>>> help(pynml.list_exposures)

list_exposures(nml_doc_fn, substring='')
    List exposures in a NeuroML model document file.

    This wraps around `lems.model.list_exposures` to list the exposures in a
    NeuroML2 model. The only difference between the two is that the
    `lems.model.list_exposures` function is not aware of the NeuroML2 component
    types (since it's for any LEMS models in general), but this one is.

    The returned dictionary is of the form:

    ..
    {
        "component": ["exp1", "exp2"]
    }
```

When run on the example [Izhikevich network example](#), it will return:

```
>>> pynml.list_exposures("izhikevich2007_network.nml")

{<lems.model.component.FatComponent at 0x7f25b62caca0>: {'g': <lems.model.component.
    ↪Exposure at 0x7f25dd1d2be0>,
    'i': <lems.model.component.Exposure at 0x7f25dc921e80>},
<lems.model.component.FatComponent at 0x7f25b62cad00>: {'u': <lems.model.component.
    ↪Exposure at 0x7f25b5f57400>,
    'iSyn': <lems.model.component.Exposure at 0x7f25b607a670>,
    'iMemb': <lems.model.component.Exposure at 0x7f25b607aa00>,
    'v': <lems.model.component.Exposure at 0x7f25b6500220>},
<lems.model.component.FatComponent at 0x7f25b62cadf0>: {'i': <lems.model.component.
    ↪Exposure at 0x7f25dc921e80>},
<lems.model.component.FatComponent at 0x7f25b62caf70>: {'i': <lems.model.component.
    ↪Exposure at 0x7f25dc921e80>},
<lems.model.component.FatComponent at 0x7f25b5fc2ac0>: {'i': <lems.model.component.
    ↪Exposure at 0x7f25dc921e80>},
<lems.model.component.FatComponent at 0x7f25b65be9d0>: {'i': <lems.model.component.
    ↪Exposure at 0x7f25dc921e80>},
<lems.model.component.FatComponent at 0x7f25b65bed00>: {'i': <lems.model.component.
    ↪Exposure at 0x7f25dc921e80>},
...
}
```

This second function is primarily for use by the `list_recording_paths_for_exposures` function.

As noted in the helper documentation, these are both based on a function of the same name implemented in [PyLEMS](#), version 0.5.8+.

11.4 Quantities and recording

In LEMS and NeuroML, quantities from all exposures and all events can be recorded by referring to them using *paths*. For examples, please see the [Getting Started with NeuroML](#) section.

11.4.1 Recording events

In NeuroML, all events can be recorded to files declared using the *EventOutputFile* component. Once an *EventOutputFile* has been declared, events to record can be selected using the *EventSelection* component.

pyNeuroML provides the `create_event_output_file` function to create a *EventOutputFile* to record events to, and the `add_selection_to_event_output_file` function to record events to the declared data file(s).

11.4.2 Recording quantities from exposures

In NeuroML, all quantities can be recorded to files declared using the *OutputFile* component. Once the *OutputFile* has been declared, quantities to record can be selected using the *OutputColumn* component.

pyNeuroML provides the `create_output_file` function to create a *OutputFile* to record quantities to, and the `add_column_to_output_file` function to select quantities to record to the declared data file(s).

11.5 LEMS Simulation files

For many users, the most obvious place that LEMS is used is in the LEMS Simulation file (usually *LEMS_*.xml*).

In short, what a file like this does is:

- point at the NeuroML file containing the model to simulate
- include any other LEMS file it needs, including the *NeuroML core type definitions*
- specify how long to run the simulation for and the simulation timestep (*dt*)
- say what to display when the simulation has finished (e.g. membrane potentials of selected cells)
- say what to save to file, e.g. voltage traces, spike times

These files are crucial in many of the workflows for *simulating NeuroML models*, and are reused across different simulator targets, e.g. `jnml LEMS_MyNetwork.xml` (run in jNeuroML), `jnml LEMS_MyNetwork.xml -neuron` (convert to NEURON), `jnml LEMS_MyNetwork.xml -brian2` (convert to Brian2). See [here](#) for more information.

11.5.1 Specification of format

See [here](#) for definition of the main elements used in the file, including *Display*, *OutputFile*, etc.

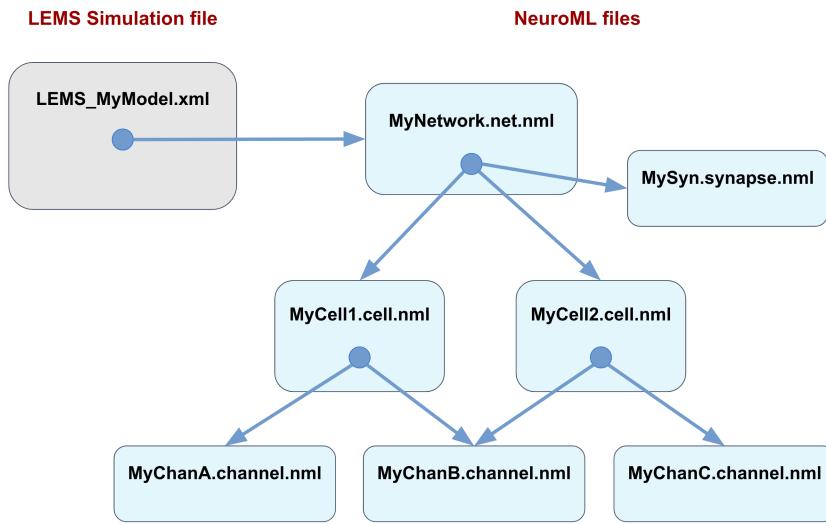


Fig. 11.2: Typical organisation for a NeuroML simulation. The main NeuroML model is specified in a file with the network (*.net.nml), which can include/point to files containing individual synapses (*.synapse.nml) or cell files (*.cell.nml). If the latter are conductance based, they may include external channel files (*.channel.nml). The main LEMS Simulation file only needs to include the network file, and tools for running simulations of the model refer to just this LEMS file. Exceptions to these conventions are frequent and simulations will run perfectly well with all the elements inside the main LEMS file, but using this scheme will maximise reusability of model elements.

11.5.2 Quantities and paths

Specifying the quantities to save/display in a LEMS Simulation file is an important and sometimes confusing process. There is a [dedicated page](#) on quantities and paths in LEMS and NeuroML2.

11.5.3 Creating LEMS Simulation files

Perhaps the easiest way to create a LEMS Simulation file is to base it off of an existing example.

```

<Lems>

    <!-- Specify the Simulation element below as what LEMS should load. Save a
        report of the simulation (e.g. simulator version, run time) in a file-->
    <Target component="sim1" reportFile="report.txt"/>

    <Include file="Cells.xml"/>
    <Include file="Networks.xml"/>
    <Include file="Simulation.xml"/>

    <!-- Including file with a <neuroml> root, a "real" NeuroML 2 file -->
    <Include file="NML2_SingleCompHHCell.nml"/>

    <!-- What to run (from the above NeuroML file) and what duration/timestep -->
    <Simulation id="sim1" length="300ms" step="0.01ms" target="net1">

        <!-- Display a trace in a new window -->
        <Display id="d1" title="HH cell with simple morphology: voltage" timeScale=
        "1ms" xmin="0" xmax="300" ymin="-90" ymax="50">
  
```

(continues on next page)

(continued from previous page)

```

        <Line id="v" quantity="hhpop[0]/v" color="#cccccc" scale="0.001"-
    <timeScale="1ms"/>
    </Display>

    <!-- Save a variable to file -->
    <OutputFile id="of0" fileName="ex_v.dat">
        <OutputColumn id="v" quantity="hhpop[0]/v"/>
    </OutputFile>

    <!-- Save spike times from a cell to file -->
    <EventOutputFile id="spikes" fileName="ex.spikes" format="TIME_ID">
        <EventSelection id="0" select="hhpop[0]" eventPort="spike"/>
    </EventOutputFile>

    </Simulation>

</Lems>
```

Alternatively, it is possible to create a LEMS Simulation file in Python file using pyNeuroML:

```

from pyneuroml.lems import LEMSSimulation

ls = LEMSSimulation('sim1', 500, 0.05, 'net1')
ls.include_neuroml2_file('NML2_SingleCompHHCell.nml')

ls.create_display('display0', "Voltages", "-90", "50")
ls.add_line_to_display('display0', 'v', "hhpop[0]/v", "1mV", "#ffffff")

ls.create_output_file('Volts_file', 'v.dat')
ls.add_column_to_output_file('Volts_file', 'v', "hhpop[0]/v")

ls.save_to_file()
```

See this example for more details.

11.5.4 What about SED-ML?

The Simulation Experiment Description Markup Language (**SED-ML**) is used by a number of other initiatives such as SBML for specifying simulation setup, execution and basic analysis.

We chose to have a LEMS specific format for specifying simulations in NeuroML2 as opposed to natively supporting SED-ML, mainly because of the tight link to the LEMS language and *jLEMS* package, i.e. all of the NeuroML2 elements and elements in a LEMS simulation file have underlying definitions in the LEMS language. However it is possible to convert the LEMS simulation to the equivalent in SED-ML.

Exporting LEMS simulation descriptions to SED-ML

```
# Using jnml
jnml <LEMS simulation file> -sedml

# Using pynml
pynml <LEMS simulation file> -sedml
```

11.6 Extending NeuroML with LEMS

As a language, LEMS defines a set of built-in types which can be used together to build more user-defined types. For example, Python defines `int`, `float`, `str` and so on as built-in types, and these can then be combined to define user defined types, classes. An object of a particular class/type can be instantiated by supplying values for the members defined in the class/type.

ComponentTypes in LEMS are similar to classes in Python. They define the membership structure of the type, but they do not specify values for their members. Once a ComponentType has been defined, an instance of it can be created by setting values for its members. This object is referred to as a **Component** in LEMS.

The NeuroML2 standard is a list of *curated ComponentTypes*. In cases where the set of ComponentTypes defined in the NeuroML standard is not sufficient for a particular modelling project, new ComponentTypes can be defined in LEMS to extend the NeuroMLv2 standard.

Having definitions in LEMS allows their re-use, and these new ComponentTypes can be submitted for inclusion to the NeuroMLv2 specification to be made accessible to other users.

- Like NeuroML, LEMS files are also XML files.
- Next, like NeuroML, LEMS also has a well defined schema (XSD) that is used to validate LEMS XML files.
- However, also similar to NeuroML, you can use the *LEMS Python tools* to work with LEMS and do not need to work directly with the XML files.

11.6.1 LEMS elements

The list of built-in types provided by LEMS can be seen in the [LEMS documentation](#). As the documentation notes, a ComponentType is the “Root element for defining component types”. It must contain a name, and can extend another ComponentType, thus inheriting its members/attributes. Each ComponentType can contain members of other LEMS types: Parameter, DerivedParameter, Dynamics, Exposure and so on. These are also all documented in the [LEMS documentation](#).

11.6.2 Example: Lorenz model for cellular convection

Let us create a new LEMS ComponentType, one that is not neuroscience specific. We will first create it using the plain XML and then see how it can be done using the Python pyLEMS API.

For this example, we will use the Lorenz model for cellular convection [Lor63]. The [Wikipedia article](#) provides a short

summary of the model, and the equations that govern it:

$$\frac{dx}{dt} = \sigma(y - x) \quad (11.1)$$

$$\frac{dy}{dt} = x(\rho - z) - y \quad (11.2)$$

$$\frac{dz}{dt} = xy - \beta z \quad (11.3)$$

So we can see here that we have three parameters:

- σ
- ρ
- β

Next, x , y , and z are the *state variables* for this model, with initial values x_0 , y_0 , and z_0 respectively. We also want to be able to observe the values of x , y , and z , so they must be *exposed* in the LEMS definition.

Let us start with the XML definition of a *ComponentType* that will describe this model. Each XML file must start with a `<Lems>` “root node”. This includes information about the version of the LEMS schema that this document is valid against. In this case, we document that this LEMS file should be valid against version 0.7.6 of the LEMS schema.

```

<Lems xmlns="http://www.neuroml.org/lems/0.7.6"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.neuroml.org/lems/0.7.6 https://raw.github.com/
       ↪LEMS/LEMS/master/Schemas/LEMS/LEMS_v0.7.6.xsd">

    <ComponentType name="lorenz1963" description="The Lorenz system is a simplified
       ↪model for atmospheric convection, derived from the Navier Stokes equations.">

        <!-- Parameters: free parameters to be used in the model -->
        <Parameter name="sigma" dimension="none" description="Prandtl Number"/>
        <Parameter name="beta" dimension="none" description="Also named b elsewhere"/>
        <Parameter name="rho" dimension="none" description="Related to the Rayleigh
       ↪number, also named r elsewhere"/>

        <!-- Initial Conditions: also free parameters to be set when creating a
       ↪Component from the ComponentType -->
        <Parameter name="x0" dimension="none"/>
        <Parameter name="y0" dimension="none"/>
        <Parameter name="z0" dimension="none"/>

        <!-- Exposure: what we want to be able to record from the LEMS simulation -->
        <Exposure name="x" dimension="none"/>
        <Exposure name="y" dimension="none"/>
        <Exposure name="z" dimension="none"/>
    </ComponentType>
</Lems>

```

Note that each parameter has a *dimension*, not a *unit*. This is because LEMS allows us to use any valid units for each dimension, and takes care of the conversion factors and so on. NeuroML also takes advantage of this LEMS feature, as noted [here](#).

Now, we can define the *dynamics* of the model, summarised in the equations above:

```

<Dynamics>
    <!-- State variables: linked to Exposures so that they can be accessed -->
    <StateVariable name="x" dimension="none" exposure="x"/>
    <StateVariable name="y" dimension="none" exposure="y"/>
    <StateVariable name="z" dimension="none" exposure="z"/>

    <!-- Equations defining the dynamics of each state variable -->
    <TimeDerivative variable="x" value="( sigma * (y - x) ) / sec"/>
    <TimeDerivative variable="y" value="( rho * x - y - x * z ) / sec"/>
    <TimeDerivative variable="z" value="( x * y - beta * z ) / sec"/>

    <!-- Actions to take on the start of a LEMS simulation -->
    <OnStart>
        <StateAssignment variable="x" value="x0"/>
        <StateAssignment variable="y" value="y0"/>
        <StateAssignment variable="z" value="z0"/>
    </OnStart>
</Dynamics>

```

Our LEMS file is almost complete. However, notice that we have used `sec` in the dynamics to denote time but have not yet declared it. We define `sec` as a *constant* whose value is defined in the `ComponentType` itself (and will not be set by us when instantiating a Component of this `ComponentType`):

```
<Constant name="sec" dimension="time" value="1s"/>
```

Also note that while we have defined this constant, we have not yet defined the `time` dimension or its units. We can do that outside the `ComponentType`:

```

<Dimension name="time" t="1"/>
<Unit name="second" symbol="s" dimension="time" power="1"/>
<Unit name="milli second" symbol="ms" dimension="time" power="-3"/>

```

We have defined two units for the time dimension, with their conversion factors. LEMS will use this information to correctly convert all dimensions as required. The NeuroML2 standard defines various dimensions and their units *in the schema* for us to use.

The complete LEMS file will be this:

```

<Lems xmlns="http://www.neuroml.org/lems/0.7.6"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.neuroml.org/lems/0.7.6 https://raw.github.com/
       ↪LEMS/LEMS/master/Schemas/LEMS/LEMS_v0.7.6.xsd">

    <Dimension name="time" t="1"/>
    <Unit name="second" symbol="s" dimension="time" power="1"/>
    <Unit name="milli second" symbol="ms" dimension="time" power="-3"/>

    <ComponentType name="lorenz1963" description="The Lorenz system is a simplified
       ↪model for atmospheric convection, derived from the Navier Stokes equations.">

        <!-- Parameters: free parameters to be used in the model -->
        <Parameter name="sigma" dimension="none" description="Prandtl Number"/>
        <Parameter name="beta" dimension="none" description="Also named b elsewhere"/>
        <Parameter name="rho" dimension="none" description="Related to the Rayleigh
       ↪number, also named r elsewhere"/>

```

(continues on next page)

(continued from previous page)

```

<!-- Initial Conditions: also free parameters to be set when creating a ComponentType -->
<Parameter name="x0" dimension="none"/>
<Parameter name="y0" dimension="none"/>
<Parameter name="z0" dimension="none"/>

<!-- Exposure: what we want to be able to record from the LEMS simulation -->
<Exposure name="x" dimension="none"/>
<Exposure name="y" dimension="none"/>
<Exposure name="z" dimension="none"/>
<Constant name="sec" dimension="time" value="1s"/>

<Dynamics>
    <!-- State variables: linked to Exposures so that they can be accessed -->
    <StateVariable name="x" dimension="none" exposure="x"/>
    <StateVariable name="y" dimension="none" exposure="y"/>
    <StateVariable name="z" dimension="none" exposure="z"/>

    <!-- Equations defining the dynamics of each state variable -->
    <TimeDerivative variable="x" value="( sigma * ( y - x ) ) / sec"/>
    <TimeDerivative variable="y" value="( rho * x - y - x * z ) / sec"/>
    <TimeDerivative variable="z" value="( x * y - beta * z ) / sec"/>

    <!-- Actions to take on the start of a LEMS simulation -->
    <OnStart>
        <StateAssignment variable="x" value="x0"/>
        <StateAssignment variable="y" value="y0"/>
        <StateAssignment variable="z" value="z0"/>
    </OnStart>
</Dynamics>
</ComponentType>
</Lems>

```

We now have a complete LEMS model declaration. To use this model, we need to create an instance of the ComponentType, a Component. This requires us to set the values of various parameters of the defined model:

```
<lorenz1963 id="lorenzCell" sigma="10" beta="2.67" rho="28"
    x0="1.0" y0="1.0" z0="1.0"/>
```

Here, we've set parameters that result in the chaotic attractor regime. We could also use different values for the parameters—like a class can have many objects with different parameters, a ComponentType can have also have different Components.

Note that one can also define a Component using the standard constructor form:

```
<Component id="lorenzCell" type="lorenz1963" sigma="10" beta="2.67" rho="28" x0="1.0"
    y0="1.0" z0="1.0"/>
```

The two forms are equivalent. As with other conventions, either form can be used as long as it is used consistently.

The `Include` element type allows us to modularise our models. In NeuroML based models, we use it to break our model down into small independent reusable files.

Writing the model in Python using PyLEMS

While the underlying format for NeuroML and LEMS is XML, Python is the suggested programming language for end users. In this section we will see how the Lorenz model can be written using the [PyLEMS](#) Python LEMS API. The complete script is below:

```
#!/usr/bin/env python3

import lems.api as lems
from lems.base.util import validate_lems

model = lems.Model()

model.add(lems.Dimension(name="time", t=1))
model.add(lems.Unit(name="second", symbol="s", dimension="time", power=1))
model.add(lems.Unit(name="milli second", symbol="ms", dimension="time", power=-3))

lorenz = lems.ComponentType(name="lorenz1963", description="The Lorenz system is a simplified model for atmospheric convection, derived from the Navier Stokes equations")
model.add(lorenz)

lorenz.add(lems.Parameter(name="sigma", dimension="none", description="Prandtl Number"))
lorenz.add(lems.Parameter(name="beta", dimension="none", description="Also named b elsewhere"))
lorenz.add(lems.Parameter(name="rho", dimension="none", description="Related to the Rayleigh number, also named r elsewhere"))

lorenz.add(lems.Parameter(name="x0", dimension="none"))
lorenz.add(lems.Parameter(name="y0", dimension="none"))
lorenz.add(lems.Parameter(name="z0", dimension="none"))

lorenz.add(lems.Exposure(name="x", dimension="none"))
lorenz.add(lems.Exposure(name="y", dimension="none"))
lorenz.add(lems.Exposure(name="z", dimension="none"))

lorenz.add(lems.Constant(name="sec", value="1s", dimension="time"))

lorenz.dynamics.add(lems.StateVariable(name="x", dimension="none", exposure="x"))
lorenz.dynamics.add(lems.StateVariable(name="y", dimension="none", exposure="y"))
lorenz.dynamics.add(lems.StateVariable(name="z", dimension="none", exposure="z"))

lorenz.dynamics.add(lems.TimeDerivative(variable="x", value="( sigma * (y - x) ) / sec"))
lorenz.dynamics.add(lems.TimeDerivative(variable="y", value="( rho * x - y - x * z ) / sec"))
lorenz.dynamics.add(lems.TimeDerivative(variable="z", value="( x * y - beta * z ) / sec"))

onstart = lems.OnStart()
onstart.add(lems.StateAssignment(variable="x", value="x0"))
onstart.add(lems.StateAssignment(variable="y", value="y0"))
onstart.add(lems.StateAssignment(variable="z", value="z0"))
lorenz.dynamics.add(onstart)
```

(continues on next page)

(continued from previous page)

```

model.add(lems.Component(id="lorenzCell", type=lorenz.name, sigma="10",
                        beta="2.67", rho="28", x0="1.0", y0="1.0", z0="1.0"))

file_name = "LEMS_lorenz.xml"
model.export_to_file(file_name)

validate_lems(file_name)

```

As you will see, the PyLEMS API exactly follows the XML constructs that we used before. Running this script, let's call it `LorenzLems.py` gives us:

```

$ python LorenzLems.py
Validating LEMS_lorenz.xml against https://raw.githubusercontent.com/LEMS/LEMS/
  ↪development/Schemas/LEMS/LEMS_v0.7.6.xsd
It's valid!

```

The generated XML file is below. As you can see, it is identical to the XML file that we wrote by hand in the previous section. You will also see that the Python API also provides convenience functions, such as the `export_to_file` and `validate_lems` functions to quickly save your model to an XML file, and validate it.

```

<?xml version="1.0" ?>
<Lems xmlns="http://www.neuroml.org/lems/0.7.6" xmlns:xsi="http://www.w3.org/2001/
  ↪XMLSchema-instance" xsi:schemaLocation="http://www.neuroml.org/lems/0.7.6 https://
  ↪raw.githubusercontent.com/LEMS/LEMS/development/Schemas/LEMS/LEMS_v0.7.6.xsd">
  <Dimension name="time" t="1"/>
  <Unit symbol="s" dimension="time" power="1" scale="1.0"/>
  <Unit symbol="ms" dimension="time" power="-3" scale="1.0"/>
  <ComponentType name="lorenz1963" description="The Lorenz system is a simplified
  ↪model for atmospheric convection, derived from the Navier Stokes equations">
    <Parameter name="sigma" dimension="none" description="Prandtl Number"/>
    <Parameter name="beta" dimension="none" description="Also named b elsewhere"/>
    <Parameter name="rho" dimension="none" description="Related to the Rayleigh
  ↪number, also named r elsewhere"/>
    <Parameter name="x0" dimension="none"/>
    <Parameter name="y0" dimension="none"/>
    <Parameter name="z0" dimension="none"/>
    <Constant name="sec" value="1s" dimension="time"/>
    <Exposure name="x" dimension="none"/>
    <Exposure name="y" dimension="none"/>
    <Exposure name="z" dimension="none"/>
    <Dynamics>
      <StateVariable name="x" dimension="none" exposure="x"/>
      <StateVariable name="y" dimension="none" exposure="y"/>
      <StateVariable name="z" dimension="none" exposure="z"/>
      <TimeDerivative variable="x" value="( sigma * (y - x) ) / sec"/>
      <TimeDerivative variable="y" value="( rho * x - y - x * z ) / sec"/>
      <TimeDerivative variable="z" value="( x * y - beta * z ) / sec"/>
      <OnStart>
        <StateAssignment variable="x" value="x0"/>
        <StateAssignment variable="y" value="y0"/>
        <StateAssignment variable="z" value="z0"/>
      </OnStart>
    </Dynamics>

```

(continues on next page)

(continued from previous page)

```
</ComponentType>
<Component id="lorenzCell" type="lorenz1963" sigma="10" beta="2.67" rho="28" x0="1.0"
           y0="1.0" z0="1.0"/>
</Lems>
```

We strongly suggest that users use the Python tools when working with both NeuroML and LEMS. Not only is Python easier to read and write than XML, it also provides powerful programming constructs and has a rich ecosystem of scientific software.

11.6.3 Examples

Here are some examples of Components written using LEMS to extend NeuroML that can be used as references.

- The Lorenz example XML source code
- An example script for building a LEMS model using Python
- Defining a new synapse in LEMS
- Defining an ion channel in LEMS
- Defining a new Calcium pool in LEMS

SOFTWARE AND TOOLS

12.1 Core NeuroML Tools

The NeuroML initiative supports **a core set of libraries** (mainly in Python and Java) to enable the creation/validation/analysis/simulation of NeuroML models as well as to facilitate adding support for the language to other applications.

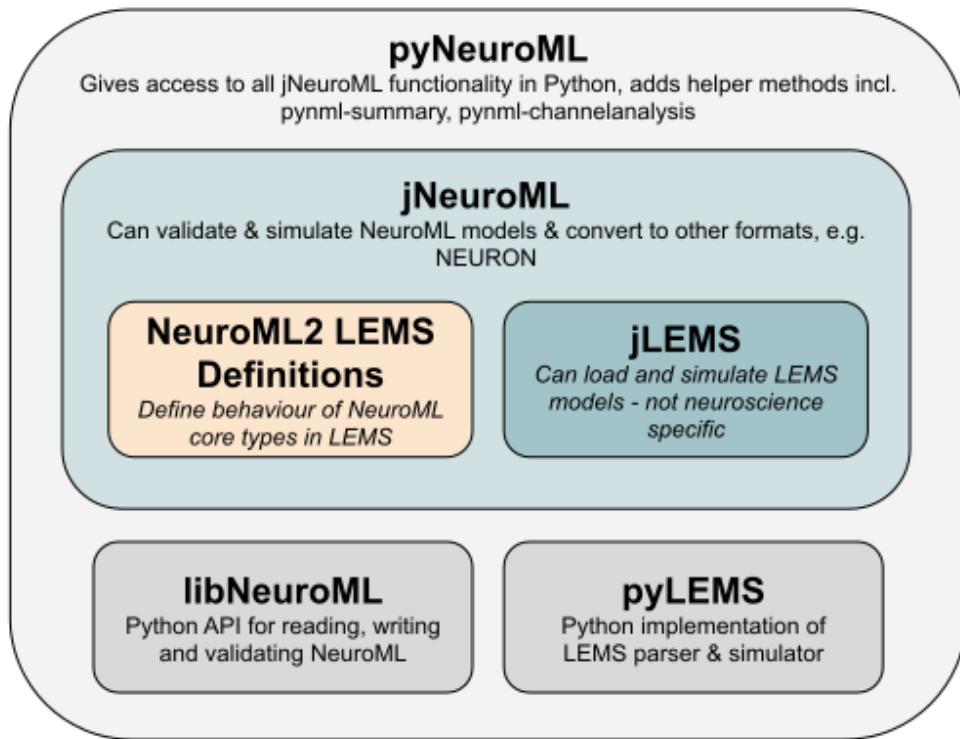


Fig. 12.1: Relationship between *jLEMS*, *jNeuroML*, the *NeuroML 2 LEMS definitions*, *libNeuroML*, *pyLEMS* and *pyNeuroML*.

12.1.1 Python based applications

For most users, [pyNeuroML](#) will provide all of the key functionality for building, validating, simulating, visualising, and converting NeuroML 2 and LEMS models. It builds on [libNeuroML](#) and [pyLEMS](#) and bundles all of the functionality of [jNeuroML](#) to provide access to this through a Python interface.

12.1.2 Java based applications

[jNeuroML](#) (for validating, simulating and converting NeuroML 2 models) and [jLEMS](#) (for simulating LEMS models) are the key applications created in Java for supporting NeuroML 2/LEMS.

12.1.3 NeuroML support in other languages

There are preliminary APIs for using NeuroML in [C++](#) and [MATLAB](#).

12.2 Other NeuroML supporting applications

Many other simulators, applications and libraries support NeuroML. See [here](#) for more details.

A number of databases and neuroinformatics initiatives support NeuroML as a core interchange format. See [here](#) for more details.

12.3 pyNeuroML

Suggested NeuroML tool

pyNeuroML is the suggested software tool for working with NeuroML. It builds on [jNeuroML](#), [libNeuroML](#), and [pyLEMS](#).

Citation

Please cite Vella et al. ([VCC+14]) if you use pyNeuroML.

pyNeuroML is a Python package that allows you to work with NeuroML models using the Python programming language. It includes all the API functions provided by [libNeuroML](#) and [pyLEMS](#), and also wraps all the functions that [jNeuroML](#) provides, which can therefore be used from within Python itself.

With pyNeuroML you can:

- **Create** NeuroML models and simulations
- **Validate** NeuroML v1.8.1 and v2.x files
- **Simulate** NeuroML 2 models
- **Export** NeuroML 2 and LEMS files to many formats such as Neuron, Brian, Matlab, etc.
- **Import** other languages into LEMS (e.g. SBML)
- **Visualise** NeuroML models and simulations

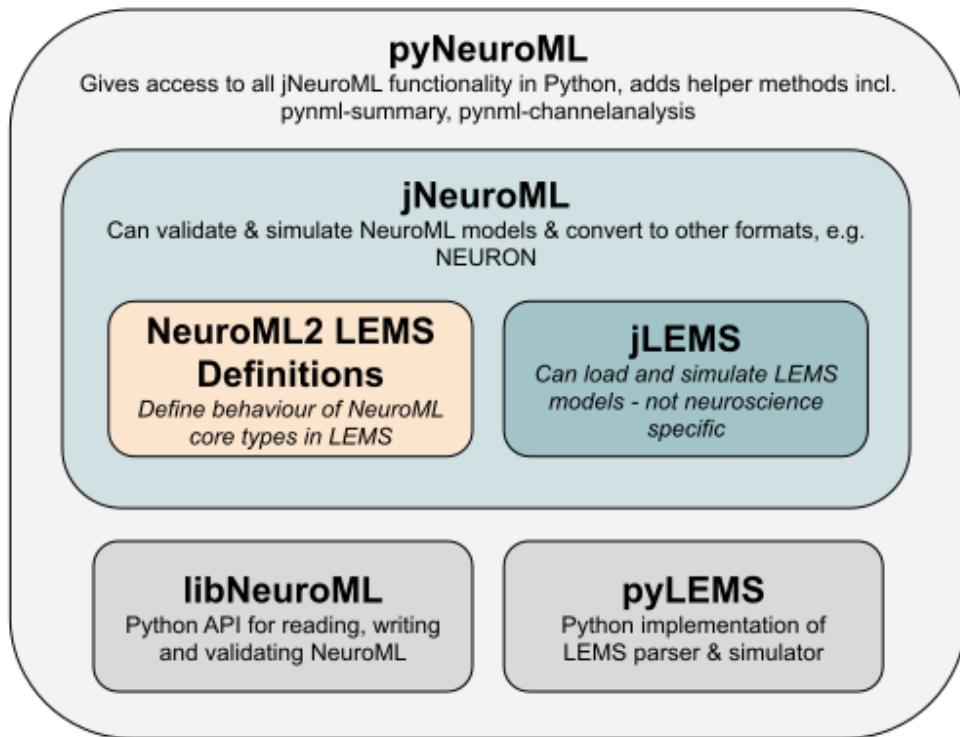


Fig. 12.2: Relationship between *jLEMS*, *jNeuroML*, the *NeuroML 2 LEMS definitions*, *libNeuroML*, *pyLEMS* and *pyNeuroML*.

12.3.1 Quick start

Install Python and the Java Runtime Environment

Python is generally pre-installed on all computers nowadays. However, if you do not have Python installed on your system, please follow the official [installation instructions](#) to install Python on your computer. A number of Free/Open source Integrated Development Environments (IDEs) are also available that make working with Python (even) easier. An example list is [here](#).

Since pyNeuroML wraps around jNeuroML which is written in Java, you will need a Java Runtime Environment (JRE) installed on your system. On most Linux systems [Free/Open source OpenJDK runtime environment](#) is already pre-installed. You can also install Oracle's proprietary Java platform from their [download page](#) if you prefer. Please refer to your operating system's documentation to install a JRE.

Install pyNeuroML with pip

Tip: Use a virtual environment

While using Python packages, it is suggested to use a virtual environment to isolate the software you install from each other. Learn more about using virtual environments in Python [here](#).

The easiest way to install the latest version of pyNeuroML is using the default Python package manager, pip:

```
pip install pyneuroml
```

By default, this will only install the minimal set of packages required to use pyNeuroML. To use pyNeuroML with specific *supporting tools*, please install them as required:

For [NEURON](#)

```
pip install NEURON
```

For compiling NEURON mod files, you also need a C compiler and the make utility installed on your computer. Additionally, to run parallel simulations the [MPI](#) libraries are also needed. Please see the [NEURON installation documentation](#) for more information on installing NEURON on your computer.

[Brian](#), [NetPyNE](#) can also be similarly installed:

```
pip install netpyne  
pip install brian2
```

For more information, please refer to their respective documentations.

Installation on Fedora Linux

On [Fedora](#) Linux systems, the [NeuroFedora](#) community provides pyNeuroML as a package in their extras repository and can be installed using the following commands:

```
sudo dnf copr enable @neurofedora/neurofedora-extra  
sudo dnf install python3-pyneuroml
```

Optional packages can also be installed using the default package manager:

```
sudo dnf install python3-brian2 python3-neuron neuron-devel python3-netpyne
```

MPI builds of these tools are also available in the NeuroFedora repositories. Please see the [project documentation](#) on installing and using them.

12.3.2 Documentation

pyNeuroML provides a set of command line utilities along with an API to use from within Python scripts:

TODO!

Check that all of these have usage documentation that is viewable using the `-h` flag. Issue filed: <https://github.com/NeuroML/pyNeuroML/issues/87>

- pynml
- pynml-channelanalysis
- pynml-modchananalysis
- pynml-plotspikes
- pynml-povray

- pynml-sonata
- pynml-summary
- pynml-tune

These utilities are self-documented. So, to learn how these utilities are to be used, run them with the `-h` flag. For example:

```
pynml -h
usage: pynml [-h/--help] [<shared options>] <one of the mutually-exclusive options>

pyNeuroML v0.5.9: Python utilities for NeuroML2
  libNeuroML v0.2.54
  jNeuroML v0.10.2

optional arguments:
  -h, --help            show this help message and exit

Shared options:
  These options can be added to any of the mutually-exclusive options

  -verbose              Verbose output
  -java_max_memory MAX Java memory for jNeuroML, e.g. 400M, 2G (used in
                        -Xmx argument to java)
  -nogui                Suppress GUI,
                        i.e. show no plots, just save results
<LEMS/NeuroML 2 file>          LEMS/NeuroML 2 file to process

  ...

  ...
```

API documentation

Detailed API documentation for pyNeuroML can be found [here](#).

The pyNeuroML API is also self documented. You can use Python's in-built documentation viewer `pydoc` to view the documentation for any of the package's modules and their functions:

```
pydoc pyneuroml
Help on package pyneuroml:

NAME
    pyneuroml

PACKAGE CONTENTS
    analysis (package)
    lems (package)
    neuron (package)
    plot (package)
    povray (package)
    pynml
    swc (package)
    tune (package)

DATA
    JNEUROML_VERSION = '0.10.2'
```

(continues on next page)

(continued from previous page)

<i>VERSION</i>	0.5.9
<i>FILE</i>	/usr/lib/python3.9/site-packages/pyneuroml/__init__.py
<pre>pydoc pyneuroml.analysis Help on package pyneuroml.analysis in pyneuroml: NAME pyneuroml.analysis PACKAGE CONTENTS ChannelDensityPlot ChannelHelper NML2ChannelAnalysis FUNCTIONS analyse_spiketime_vs_dt(nml2_file, target, duration, simulator, cell_v_path, dts,_ ↪verbose=False, spike_threshold_mV=0, show_plot_already=True, save_figure_to=None,_ ↪num_of_last_spikes=None) generate_current_vs_frequency_curve(nml2_file, cell_id, start_amp_nA=-0.1, end_amp_nA=0.1, step_nA=0.01, customamps_nA=[], analysis_duration=1000, analysis_delay=0, pre_zero_pulse=0, post_zero_pulse=0, dt=0.05, temperature='32degC', spike_threshold_mV=0.0, plot_voltage_traces=False, plot_if=True, plot_iv=False, xlim_if=None, ylim_if=None, xlim_iv=None, ylim_iv=None, label_xaxis=True, label_yaxis=True, show_volts_label=True, grid=True, font_size=12, if_iv_color='k',_ ↪linewidth=1, bottom_left_spines_only=False, show_plot_already=True, save_voltage_traces_to=None, save_if_figure_to=None, save_iv_figure_to=None, save_if_data_to=None, save_iv_data_to=None, simulator='jNeuroML', num_processors=1, include_included=True, title_above_plot=False, return_axes=False, verbose=False)</pre>	
<i>FILE</i>	/usr/lib/python3.9/site-packages/pyneuroml/analysis/__init__.py

Most IDEs are able to show you this information as you use them in your Python scripts.

12.3.3 Getting help

For any questions regarding pyNeuroML, please open an issue on the GitHub issue tracker [here](#). Any bugs and feature requests can also be filed there.

You can also use any of the *communication channels of the NeuroML community*.

12.3.4 Development

pyNeuroML is developed on GitHub at <https://github.com/NeuroML/pyNeuroML> under the [LGPL-3.0 license](#). The repository contains the complete source code along with instructions on building/installing pyNeuroML. Please follow the instructions there to build pyNeuroML from source.

12.4 libNeuroML

libNeuroML is a Python package for working with models specified in NeuroML version 2. It provides a native Python object model corresponding to the NeuroML schema. This allows users to build their NeuroML models natively in Python without having to work directly with the underlying XML representation. Additionally, libNeuroML includes functions for the conversion of the Python representation of the NeuroML model to and from the XML representation.

Use pyNeuroML

pyNeuroML builds on libNeuroML and includes additional utility functions.

Citation

Please cite Vella et al. ([VCC+14]) if you use libNeuroML.

12.4.1 Quick start

Install Python

Python is generally pre-installed on all computers nowadays. However, if you do not have Python installed on your system, please follow the official [installation instructions](#) to install Python on your computer. A number of Free/Open source Integrated Development Environments (IDEs) are also available that make working with Python (even) easier. An example list is [here](#).

Install libNeuroML with pip

Tip: Use a virtual environment

While using Python packages, it is suggested to use a virtual environment to isolate the software you install from each other. Learn more about using virtual environments in Python [here](#).

The easiest way to install the latest version of libNeuroML is using the default Python package manager, pip:

```
pip install libNeuroML
```

Installation on Fedora Linux

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
```

12.4.2 Documentation

Detailed API documentation for libNeuroML can be found [here](#). For more information on libNeuroML, please see Vella et al. ([VCC+14]) and Cannon et al. ([CGC+14]).

The core classes in NeuroML 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.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
```

12.4.3 Getting help

For any questions regarding libNeuroML, please open an issue on the GitHub issue tracker [here](#). Any bugs and feature requests can also be filed there.

You can also use any of the [communication channels of the NeuroML community](#).

libNeuroML Python API: **modules**, **Classes**, **Classifications** and **usage examples**

neuroml		
<i>AdExlaFCell</i>	<i>IafCell</i>	<i>IF_cond_alpha</i>
<i>IzhikevichCell</i>	<i>IafRefCell</i>	<i>IF_cond_exp</i>
	<i>IafTauCell</i>	<i>IF_curr_alpha</i>
	<i>IafTauRefCell</i>	<i>IF_curr_exp</i>
	<i>FitzHughNagumoCell</i>	<i>EIF_cond_alpha_isfa_ista</i>
		<i>EIF_cond_exp_isfa_ista</i>
<i>CellMorphology</i>	<i>BiophysicalProperties</i>	<i>IntracellularProperties</i>
<i>Segment</i>	<i>MembraneProperties</i>	<i>Resistivity</i>
<i>SegmentGroup</i>	<i>SpecificCapacitance</i>	<i>Species</i>
<i>Point3DWithDiam</i>	<i>ChannelDensity</i>	
	<i>ChannelDensityNernst</i>	
	<i>ChannelDensityGHK</i>	
<i>ExpOneSynapse</i>	<i>BlockingPlasticSynapse</i>	
<i>ExpTwoSynapse</i>	<i>PlasticityMechanism</i>	
	<i>BlockMechanism</i>	
<i>IonChannelHH</i>	<i>HHRate</i>	
<i>GateHHRates</i>	<i>HHTime</i>	
<i>GateHHTauInf</i>	<i>HHVariable</i>	
<i>PulseGenerator</i>	<i>RampGenerator</i>	<i>SpikeGenerator</i>
<i>SineGenerator</i>	<i>VoltageClamp</i>	<i>SpikeGeneratorPoisson</i>
		<i>SpikeGeneratorRandom</i>
		<i>SpikeArray</i>
<i>PopulationInstance</i>	<i>ProjectionConnection</i>	<i>InputList</i>
<i>Location</i>		<i>Input</i>
<i>NeuroMLDocument</i>		
neuroml.loaders		
<i>NeuroMLLoader</i>	<i>XMLLoader</i>	
<i>SWCLoader</i>		<i>ArrayMorphLoader</i>
neuroml.writers		
<i>NeuroMLWriter</i>		<i>ArrayMorphWriter</i>
		<i>JSONWriter</i>
neuroml.utils		
<i>Point neuron models</i>		
<pre>izh = IzhikevichCell(id='izh', a='0.02', b='0.2', c='−65.0', d='6', v0='−70mV', thresh='30mV') iaf = IafTauCell(id='iafTau', leakReversal='−50mV', thresh='−55mV', reset='−70mV', tau='30ms')</pre>		
<i>Multicompartment, conductance based neuron models</i>		
<pre>cell = Cell(id='cell0') cd = ChannelDensity(cond_density='50mS_per_cm2', ion_channel='NaF', erev='55mV', ion='na') cell.membrane_properties = MembraneProperties().channel_densities.append(cd)</pre>		
<i>Synapses</i>		
<pre>e2syn = ExpTwoSynapse(id='gaba', tau_decay='12ms', tau_rise='3ms', gbase='1nS', erev='−70mV')</pre>		
<i>Ion channels</i>		
<pre>ic = IonChannelHH(id='NaF', conductance='10pS', species='na') m = GateHHRates(id='m', instances='3') m.forward_rate = HHRate(type='HHExpRate', rate='0.07per_ms', midpoint='−65mV', scale='−20mV')</pre>		
<i>Inputs</i>		
<pre>p = PulseGenerator(id='p', delay='10ms', duration='100ms', amplitude='50 pA') v = VoltageClamp(id='v', delay='5ms', duration='5ms', target_voltage='0mV', series_resistance='10ohm')</pre>		
<i>Networks</i>		
<pre>net = Network(id='net1') pop = Population(id='p1', component='cell0', size=10) net.populations.append(pop)</pre>		
<i>Top level class for constructing models</i>		
<pre>nml_doc = NeuroMLDocument() nml_doc.append(net)</pre>		
<i>File readers/importers</i>		
<pre>nml_doc2 = NeuroMLLoader.load('file.nml')</pre>		
<i>Export formats</i>		
<pre>JSONWriter.write(nml_doc, 'file.nml')</pre>		
<i>Helper methods</i>		
<pre>validate_neuroml2(nml_file)</pre>		

Fig. 12.3: Examples of mapping between Component names in the NeuroML schema and their corresponding libNeuroML Python classes.

12.4.4 Development

libNeuroML is developed on GitHub at <https://github.com/NeuralEnsemble/libNeuroML> under the [BSD 3 clause license](#). The repository contains the complete source code along with instructions on building/installing libNeuroML. Please follow the instructions there to build libNeuroML from source.

12.5 pyLEMS

pyLEMS is a Python package which provides an API, as well as a simulator for the [LEMS](#) language. It can also be used to run NeuroML2 models.

Use pyNeuroML

pyNeuroML builds on pyLEMS and includes additional functions.

Citation

Please cite Vella et al. ([VCC+14]) if you use pyLEMS.

12.5.1 Quick start

Install Python

Python is generally pre-installed on all computers nowadays. However, if you do not have Python installed on your system, please follow the official [installation instructions](#) to install Python on your computer. A number of Free/Open source Integrated Development Environments (IDEs) are also available that make working with Python (even) easier. An example list is [here](#).

Install pyLEMS with pip

Tip: Use a virtual environment

While using Python packages, it is suggested to use a virtual environment to isolate the software you install from each other. Learn more about using virtual environments in Python [here](#).

The easiest way to install the latest version of pyLEMS is using the default Python package manager, pip:

```
pip install pyLEMS
```

Installation on Fedora Linux

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

```
sudo dnf install python3-pyLEMS
```

12.5.2 Documentation

Detailed API documentation for PyLEMS can be found [here](#). pyLEMS provides the `pylems` command line utility that can be used to simulate LEMS files. `pylems` is self documented, and you can learn about its usage using the `-h` flag:

```
pylems -h
usage: pylems [-h] [-I <Include directory>] [-nogui] [-dlems] <LEMS file>

positional arguments:
  <LEMS file>           LEMS file to be simulated

optional arguments:
  -h, --help             show this help message and exit
  -I <Include directory>
                        Directory to be searched for included files
  -nogui                If this is specified, just parse & simulate the model, but don
                        ↵'t show any plots
  -dlems                If this is specified, export the LEMS file as dLEMS_
                        ↵(distilled LEMS in JSON format, see https://github.com/borismarin/som-codegen)
```

To simulate a LEMS file:

```
pylems lemsexample.xml
```

Please note that if you are simulating a NeuroML file you will have to also specify the location of the [NeuroML 2 LEMS definitions](#) with the `-I` option. We suggest that you use `pyNeuroML` where this is not required:

```
pylems -I <dir of NeuroML2 install>/NeuroML2CoreTypes/ LEMS_NeuroML2_Model.xml
```

For more information on pyLEMS, please see Vella et al. ([VCC+14]) and Cannon et al. ([CGC+14]).

API documentation

Detailed API documentation for pyNeuroML can be found [here](#).

The pyLEMS API is also self documented. You can use Python's in-built documentation viewer `pydoc` to view the documentation for any of the package's modules and their functions:

```
Help on package lems:

NAME
    lems

DESCRIPTION
    @author: Gautham Ganapathy
    @organization: LEMS (http://neuroml.org/lems/, https://github.com/organizations/LEMS)
```

(continues on next page)

(continued from previous page)

```
@contact: gautham@lisphacker.org

PACKAGE CONTENTS
api
base (package)
dlems (package)
model (package)
parser (package)
run
sim (package)

DATA
logger = <Logger LEMS (WARNING)>

VERSION
0.5.2

FILE
/usr/lib/python3.9/site-packages/lems/__init__.py
```

Most IDEs are able to show you this information as you use them in your Python scripts.

12.5.3 Getting help

For any questions regarding pyLEMS, please open an issue on the GitHub issue tracker [here](#). Any bugs and feature requests can also be filed there.

You can also use any of the *communication channels of the NeuroML community*.

12.5.4 Development

pyLEMS is developed on GitHub at <https://github.com/LEMS/pylems> under the [LGPL-3.0 license](#). The repository contains the complete source code along with instructions on building/installing pyLEMS. Please follow the instructions there to build pyLEMS from source.

12.6 NeuroMLlite

NeuroMLlite is a common framework for reading/writing/generating network specifications which builds on NeuroML 2. It is intended to provide a high level specification which can be used to generate networks in NeuroML and many other formats—including graphical and in neuronal simulator formats.

Note: NeuroMLlite is under active development

Please [watch the GitHub repository](#) to receive regular updates on its progress.

12.6.1 Quick start

Install Python

Python is generally pre-installed on all computers nowadays. However, if you do not have Python installed on your system, please follow the official [installation instructions](#) to install Python on your computer. A number of Free/Open source Integrated Development Environments (IDEs) are also available that make working with Python (even) easier. An example list is [here](#).

Install NeuroMLlite with pip

Tip: Use a virtual environment

While using Python packages, it is suggested to use a virtual environment to isolate the software you install from each other. Learn more about using virtual environments in Python [here](#).

The easiest way to install the latest version of libNeuroML is using the default Python package manager, pip:

```
pip install neuromllite
```

Installation on Fedora Linux

On [Fedora](#) Linux systems, the [NeuroFedora](#) community provides pyNeuroML as a package in their [extras repository](#) and can be installed using the following commands:

```
sudo dnf copr enable @neurofedora/neurofedora-extra
sudo dnf install python3-neuromllite
```

12.6.2 Documentation

Along with a Python API, NeuroMLlite also provides a graphical user interface `nmllite-ui` that can be used to create network models and export or simulate them using different simulators supported by NeuroML.

```
nmllite-ui

NMLlite-UI v0.2.4: A GUI for loading NeuroMLlite files

Usage:
    nmllite-ui Sim_xxx.json
        Load a NeuroMLlite file containing a Simulation, which refers to the Network
        ↪ to run
```

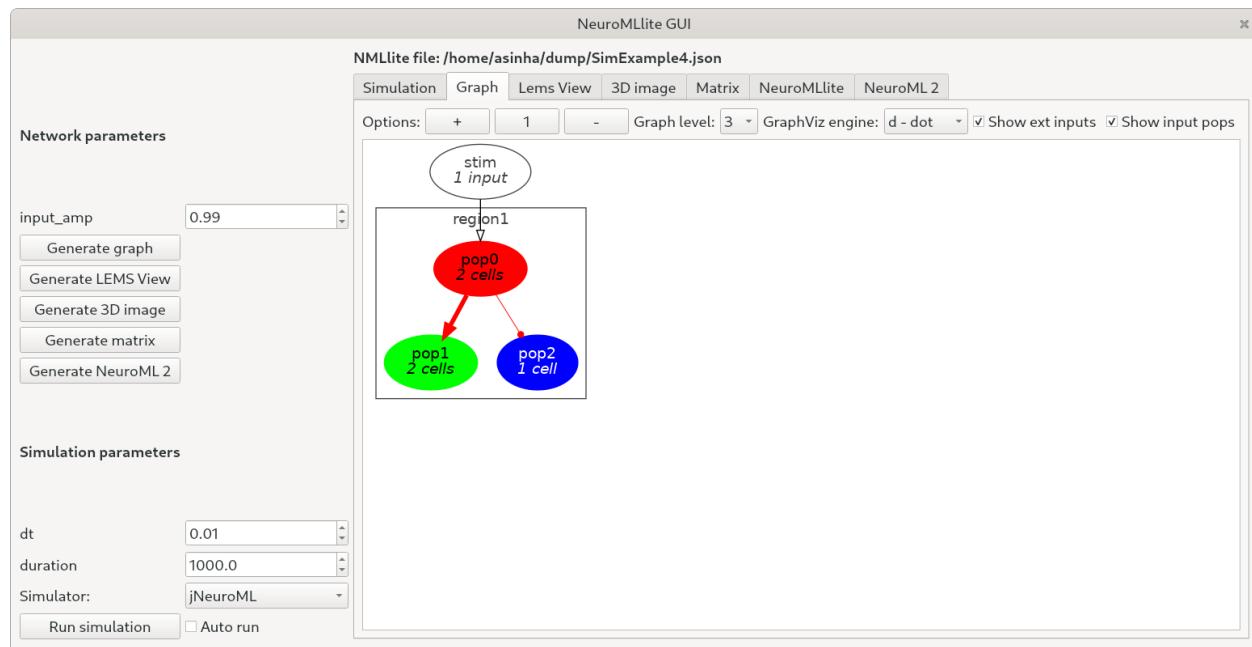


Fig. 12.4: Screenshot of NeuroMLlite UI showing an example simulation

API documentation

TODO!

Generate and publish API documentation for NeuroMLlite. Issue filed: <https://github.com/NeuroML/NeuroMLlite/issues/10>

The NeuroMLlite API is self documented. You can use Python's in-built documentation viewer `pydoc` to view the documentation for any of the package's modules and their functions:

```
Help on package neuromllite:
```

```
NAME
    neuromllite

PACKAGE CONTENTS
    ArborHandler
    BBPConnectomeReader
    BaseType
    ConnectivityHandler
    DefaultNetworkHandler
    GraphVizHandler
    MatrixHandler
    NetworkGenerator
    NeuronHandler
    PsyNeuLinkHandler
    PsyNeuLinkReader
    PyNNHandler
    SonataHandler
    SonataReader
```

(continues on next page)

(continued from previous page)

```

gui (package)
sweep (package)
utils

...

```

Most IDEs are able to show you this information as you use them in your Python scripts.

A number of examples showing how the NeuroMLlite Python API is to be used are also included in the [GitHub repository](#). For instance, `Example4.py` can be run in the following ways to generate different representations of the created network model. Please see the [Readme file](#) included in the repository for more example usage.

```

python Example4.py           # Generate the network in JSON
python Example4.py -nml       # Generate the network in NeuroML2
python Example4.py -jnml      # Generate the network in NeuroML2 & run using
    ↵jNeuroML
python Example4.py -jnmlnetpyne   # Generate the network in NeuroML2 & run using
    ↵NetPyNE
python Example4.py -jnmlnrn     # Generate the network in NeuroML2 & run using
    ↵NEURON
python Example4.py -netpyne     # Generate & run the network directly in NetPyNE
python Example4.py -pynnnest    # Generate & run the network in NEST using PyNN
python Example4.py -pynnnrn     # Generate & run the network in NEURON using PyNN
python Example4.py -pynnbian    # Generate & run the network in Brian using PyNN
...

```

12.6.3 Getting help

For any questions regarding NeuroMLlite, please open an issue on the GitHub issue tracker [here](#). Any bugs and feature requests can also be filed there.

You can also use any of the [communication channels of the NeuroML community](#).

12.6.4 Development

pyNeuroML is developed on GitHub at <https://github.com/NeuroML/NeuroMLlite> under the [LGPL-3.0 license](#). The repository contains the complete source code along with instructions on building/installing pyNeuroML. Please follow the instructions there to build pyNeuroML from source.

12.7 jNeuroML

jNeuroML is a Free/Open Source Java tool for working with LEMS and NeuroML 2. It includes the `jnml` command line application, and can also be used as a Java library.

With jNeuroML you can:

- **Validate** NeuroML v1.8.1 and v2.x files
- **Simulate** NeuroML 2 models
- **Export** NeuroML 2 and LEMS files to many formats such as Neuron, Brian, Matlab, etc.
- **Import** other languages into LEMS (e.g. SBML)

- Visualise NeuroML models and simulations

Use pyNeuroML

pyNeuroML builds on jNeuroML and includes additional functions.

12.7.1 Quick start

Install the Java Runtime Environment

Since jNeuroML is written in Java, you will need a Java Runtime Environment (JRE) installed on your system. On most Linux systems [Free/Open source OpenJDK runtime environment](#) is already pre-installed. You can also install Oracle's proprietary Java platform from their [download page](#) if you prefer. Please refer to your operating system's documentation to install a JRE.

Installation using pre-compiled JAR

jNeuroML is provided as a pre-compiled ready-to-use Java JAR file that can be used on any computer that has Java installed. Please download it from the [GitHub release page](#) and unzip (extract) it in a preferred folder on your computer:

```
cd <folder where you downloaded the jNeuroML zip file>
unzip jNeuroML.zip
```

This will extract the zip file to a new folder which will contain the pre-compiled JAR file and runner scripts:

```
ls jNeuroMLJar/
jNeuroML-0.10.2-jar-with-dependencies.jar  jnml  jnml.bat  README
```

TODO

Add instructions on using the installer script. <https://github.com/NeuroML/jNeuroML/pull/76>

Installation on Fedora Linux

On Fedora Linux systems, the [NeuroFedora](#) community provides jNeuroML as a package in their [extras repository](#) and can be installed using the following commands:

```
sudo dnf copr enable @neurofedora/neurofedora-extra
sudo dnf install jneuroml
```

12.7.2 Documentation

Information on usage of the `jnml` command line application can be found with the `-h` option:

```
jnml -h

jNeuroML v0.10.1
Usage:

jnml LEMSFile.xml
    Load LEMSFile.xml using jLEMS, parse it and validate it as LEMS, and
    execute the model it contains

jnml LEMSFile.xml -nogui
    As above, parse and execute the model and save results, but don't show GUI

...
```

API documentation

The jNeuroML API is self documented. Please refer to the various packages to learn their usage:

- NeuroML/jNeuroML (API Documentation [here](#))
- NeuroML/org.neuroml.model (API Documentation [here](#))
- NeuroML/org.neuroml.model.injectingplugin (API Documentation [here](#))
- NeuroML/org.neuroml.import: Import other formats into LEMS & combine with NeuroML models (API documentation [here](#))
- NeuroML/org.neuroml.export: Export from NeuroML & LEMS (API Documentation [here](#))

12.7.3 Getting help

For any questions regarding jNeuroML, please open an issue on the GitHub issue tracker [here](#). Any bugs and feature requests can also be filed there.

You can also use any of the *communication channels of the NeuroML community*.

12.7.4 Development

jNeuroML is developed on GitHub at <https://github.com/NeuroML/jNeuroML> under the [LGPL-3.0 license](#). The repository contains the complete source code along with instructions on building/installing jNeuroML. Please follow the instructions there to build jNeuroML from source.

Nightly (pre-release) jar builds:

Warning: Please note that these JARs are considered experimental and should only be used for testing purposes.

In case you want to use a development (un-released) version of jNeuroML, you can download a development build following the steps below. You will need to have the [Subversion](#) tool installed on your system.

```
svn checkout svn://svn.code.sf.net/p/neuroml/code/jNeuroMLJar  
cd jNeuroMLJar
```

12.8 jLEMS

jLEMS is an interpreter for the Low Entropy Model Specification language written in Java.

jLEMS is the reference implementation of LEMS

jLEMS was developed by Robert Cannon when the LEMS language was being devised and serves as the key reference for how to implement/interpret the language.

12.8.1 Quick start

Since jLEMS is included in [jNeuroML](#), it does not need to be installed separately. Please follow the instructions on installing jNeuroML provided [here](#).

Please see the *development section below* for information on building the jLEMS interpreter from source.

12.8.2 Documentation

Detailed documentation on LEMS is maintained [here](#). For more information on LEMS, please also see Cannon et al. ([CGC+14])

12.8.3 Getting help

For any questions regarding jLEMS, please open an issue on the GitHub issue tracker [here](#). Any bugs and feature requests can also be filed there.

You can also use any of the *communication channels of the NeuroML community*.

12.8.4 Development

jLEMS is developed on GitHub at <https://github.com/LEMS/jLEMS> under the MIT license. The repository contains the complete source code along with instructions on building/installing jLEMS.

12.9 NeuroML C++ API

A C++ API for NeuroML.

12.9.1 Quick start

The C++ API is generated from the *NeuroML specification* using the [CodeSynthesis XSD XML Schema to C++ data binding compiler](#). The C++ API needs to be compiled from source. Please refer to the instructions in the [Readme](#) document for instructions on building and installing the API.

12.9.2 Documentation

For information on the generated C++ structure, please see the [XSD user manual](#).

API documentation

API documentation for the C++ API can be found [here](#). It can also be generated while building the API from source, as documented in the [Readme](#).

12.9.3 Getting help

For any questions regarding the C++ NeuroML API, please open an issue on the GitHub issue tracker [here](#). Any bugs and feature requests can also be filed there.

You can also use any of the [communication channels of the NeuroML community](#).

12.9.4 Development

The C++ NeuroML API is developed on GitHub at https://github.com/NeuroML/NeuroML_API under the [MIT license](#).

12.10 MatLab NeuroML Toolbox

The NeuroML 2 Toolbox for MATLAB facilitates access to the Java NeuroML 2 API functionality (*jNeuroML*) directly within Matlab.

12.10.1 Quick start

Please install jNeuroML following the instructions provided [here](#). Run Matlab and run the `prefdir` command to find the location of your preferences folder. Create a file `javaclasspath.txt` within that folder containing, on a single line, the full path to the `jNeuroML-<version>-jar-with-dependencies.jar` from jNeuroML.

Restart Matlab, and you will be able to access jNeuroML classes. You can test your setup by validating an example file:

```
import org.neuroml.model.util.NeuroML2Validator
file = java.io.File('/full/path/to/model.nml');
validator = NeuroML2Validator();
validator.validateWithTests(file);
disp(validator.getValidity())
```

12.10.2 Documentation

Please refer to the [jNeuroML documentation](#) for information on the Java NeuroML API. Examples on using the Matlab toolbox are available [here](#).

12.10.3 Getting help

For any questions regarding the NeuroML Matlab toolbox, please open an issue on the GitHub issue tracker [here](#). Any bugs and feature requests can also be filed there.

You can also use any of the [communication channels of the NeuroML community](#).

12.10.4 Development

The NeuroML Matlab toolbox is developed on GitHub at <https://github.com/NeuroML/NeuroMLToolbox>.

12.11 Tools and resources with NeuroML support

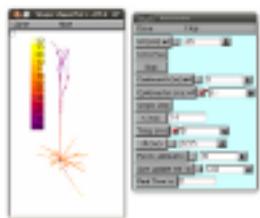
Please help us keep this page up to date.

Tools listed here may no longer be maintained, or may have moved to new locations, and others may be missing. Please [file issues](#) if you can help update this information.

Apart from the core NeuroML tools ([pyNeuroML](#), [jNeuroML](#), etc.) there are many other applications, libraries and databases which support NeuroML 2 and LEMS. These tools take a *number of different approaches* to adding NeuroML support.

12.11.1 Applications with NeuroML support

NEURON



The [NEURON](#) simulation environment is one of the main target platforms for a standard facilitating exchange of neuronal models. [jNeuroML](#) can be used to convert NeuroML2/LEMS models to NEURON. NEURON simulations can also be generated from NeuroML model components by [neuroConstruct](#).

See also [NetPyNE](#), which builds on NEURON.

There is a **dedicated page on NEURON/NeuroML interactions** [here](#).

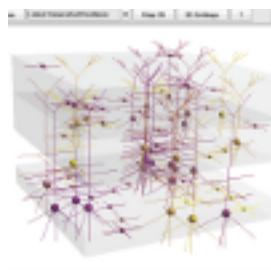
NetPyNE



[NetPyNE](#) is a Python package to facilitate the development, simulation, parallelization, analysis, and optimization of biological neuronal networks using the NEURON simulator. NetPyNE can import from and export to NeuroML. NetPyNE also provides a web based [Graphical User Interface](#).

There is a **dedicated page on NetPyNE/NeuroML interactions** [here](#).

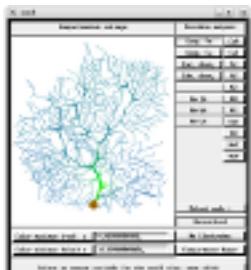
neuroConstruct



[neuroConstruct](#) is a Java based application for constructing 3D networks of biologically realistic neurons. The current version can generate code for the [NEURON](#), [GENESIS](#), [PSICS](#) and [PyNN](#) platforms and also provides import/export support for MorphML, ChannelML and NetworkML (from NeuroML v1) and for NeuroMLv2 cells and networks.

More info on the support for NeuroML in neuroConstruct is available [here](#).

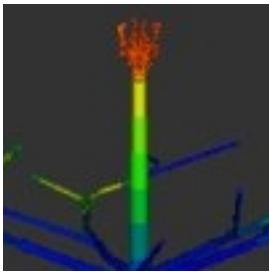
GENESIS



GENESIS is a commonly used neuronal simulation environment and was a main target platform for the NeuroMLv1 specifications. Full GENESIS simulations can be generated from NeuroMLv1 model components by [neuroConstruct](#).

Due to the lack of active development of GENESIS, support for mapping to GENESIS in NeuroMLv2 has been deprecated in favour of [MOOSE](#).

MOOSE



MOOSE is the Multiscale Object-Oriented Simulation Environment. It is the base and numerical core for large, detailed multi-scale simulations that span computational neuroscience and systems biology. It is based on a complete reimplementation of the GENESIS 2 core.

More information on running NeuroML models in MOOSE can be found [here](#).

There is a **dedicated page on MOOSE/NeuroML interactions** [here](#).

BRIAN



Brian is an easy to use, Python based simulator of spiking networks.

There is a **dedicated page on Brian/NeuroML interactions** [here](#).

EDEN

[EDEN](#) is a recently developed simulation engine which incorporates native NeuroML 2 support from the start.

Initial tests of using EDEN with NeuroML models and example code can be found [here](#).

There is a **dedicated page on EDEN/NeuroML interactions** [here](#).

Arbor

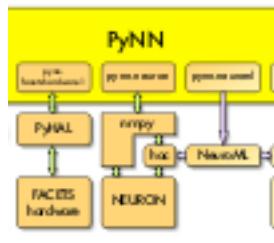


[Arbor](#) is a high performance multicompartmental neural simulation library. Addition of support for NeuroML2 and LEMS is under active development. See [here](#).

Example code for interactions between NeuroML models and Arbor can be found [here](#).

There is a **dedicated page on Arbor/NeuroML interactions** [here](#).

PyNN



[PyNN](#) is a Python package for simulator independent specification of neuronal network models. Model code can be developed using the PyNN API and then run using [NEURON](#), [NEST](#) or [Brian](#). The developed model also can be stored as a NeuroML document. The latest version of [neuroConstruct](#) can be used to generate executable scripts for PyNN based simulators based on NeuroML components, although the majority of multicompartmental conductance based models which are available in [neuroConstruct](#) are outside the current scope of the PyNN API.

More info on the latest support for running NeuroML models in PyNN and vice versa can be found [here](#).

OpenWorm



The [OpenWorm](#) project aims to create a simulation platform to build digital in-silico living systems, starting with a C. elegans virtual organism simulation. The simulations and associated tools are being developed in a fully open source manner. NeuroML is being used for the description of the 302 neurons in the worm's nervous system, both for morphological description of the cells and their electrical properties.

The [c302 subproject](#) in OpenWorm has the latest developments in the NeuroML version of the worm nervous system.

Members of the OpenWorm project are also creating a general purpose neuronal simulator (for both electrical and physical simulations) which will have parallelism and native support for NeuroML built in from the start (see [Gepetto](#)).

Model Description Format (MDF)



ModECI Model Description Format ([MDF](#)) is an open source, community-supported standard and associated library of tools for expressing computational models in a form that allows them to be exchanged between diverse programming languages and execution environments, with a particular focus on machine learning, artificial intelligence and computational neuroscience.

There will be full compatibility between NeuroML and MDF for specifying neuronal models. See [here](#) for ongoing work in this direction.

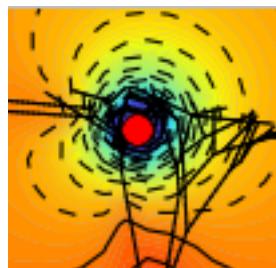
The Virtual Brain



The [Virtual Brain \(TVB\)](#) offers a simulation environment for large-scale brain networks. It allows network properties, in particular the brain's structural connectivity, to be incorporated into models, and so TVB can simulate whole brain behaviour as is commonly observed in clinical scanners (e.g. EEG, MEG, fMRI).

Initial work mapping networks in TVB to/from NeuroML 2 and LEMS can be found [here](#). See also the work of the [INCF Network Specification Working Group](#) in this area.

LFPy



LFPy is a Python package for calculation of extracellular potentials from multicompartment neuron models. It relies on the NEURON simulator and uses the Python interface it provides. LFPy provides a set of easy to use Python classes for setting up the model, running simulations and calculating the extracellular potentials arising from activity in the model neuron. Initial support for loading of NeuroML morphologies has been added.

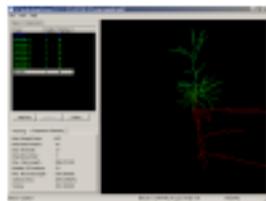
BioSimulators



BioSimulators provides a registry and platform supporting a broad range of modeling frameworks, model formats, simulation algorithms, and simulation tools.

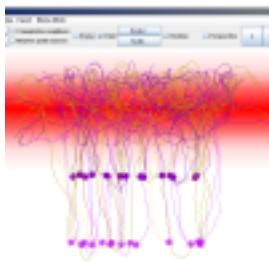
See for example <https://biosimulators.org/simulators/pyneuroml/latest>.

NeuronLand



NeuronLand provides NLMorphologyConverter, which is a command line program for converting between over 20 different 3D neuron morphology formats, and NLMorphologyViewer, which provides a simple interface for viewing these data. Both of these tools provide import and export of MorphML.

CX3D



CX3D is a tool for simulating the growth of cortex in 3D. There was a preliminary implementation of export of generated networks to NeuroML in CX3D.

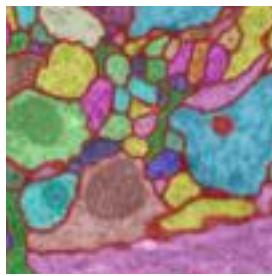
TREES toolbox



The [TREES toolbox](#) is an application in MATLAB which allows: automatic reconstruction of neuronal branching from microscopy image stacks and generation of synthetic axonal and dendritic trees; visualisation, editing and analysis of neuronal trees; comparison of branching patterns between neurons; and investigation of how dendritic and axonal branching depends on local optimization of total wiring and conduction distance.

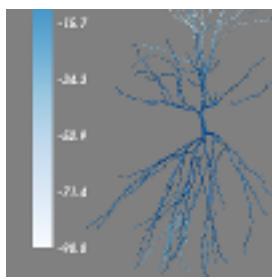
The latest version of the TREES toolbox includes basic functionality for exporting cells in NeuroML v1.x Level 1 (MorphML) or as a NeuroML v2alpha morphology file.

TrakEM2



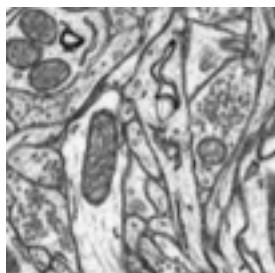
TrakEM2 is an ImageJ plugin for morphological data mining, three-dimensional modelling and image stitching, registration, editing and annotation. As of v0.8n, a menu item “Export - NeuroML...” gives an option to export to MorphML (the anatomy of the arbors only) or NeuroML (the whole network with anatomy and synapses), for the selected trees or all trees.

Neuronvisio



Neuronvisio is a Graphical User Interface for NEURON simulator environment with 3D capabilities. Neuronvisio makes it easy to select and investigate sections' properties, it offers easy integration with matplotlib for plotting the results. It can save the geometry using NeuroML and the simulation results in a customised and extensible HDF5 format; the results can then be reloaded in the software and analysed at a later stage, without re-running the simulation.

CATMAID



CATMAID is the Collaborative Annotation Toolkit for Massive Amounts of Image Data, and is a widely used tool for online reconstruction and annotation of connectomics data. Initial support for export of reconstructed neurons in NeuroML format has been added.

Myokit



Myokit (the Maastricht Myocyte Toolkit) is a Python-based software package created by Michael Clerx to simplify the use of numerical models in the analysis of cardiac myocytes. Initial support for importing ChannelML [has been added](#).

Geppetto



Geppetto is a web-based multi-algorithm, multi-scale simulation platform designed to support the simulation of complex biological systems and their surrounding environment. It is open source and is being developed as part of the [OpenWorm project](#) to create an *in-silico* model of the nematode *C. elegans*. It has had inbuilt support for NeuroML 2/LEMS from the start, and is suitable for many other types of neuronal models.

12.11.2 Other/legacy tools

Older applications

Note: many of the applications listed below are no longer in active development or links no longer work.

PSICS

The latest version of [neuroConstruct](#) can be used to generate executable scripts for **PSICS** based on NeuroML components.

Whole Brain Catalogue

The [Whole Brain Catalog](#) was a graphical interface that allowed multiscale neuroscience data to be visualised relative to a 3D brain atlas.

PCSIM

[PCSIM](#) is a tool in C++ for simulating large scale networks of cells and synapses.

Neuromantic

[Neuromantic](#) is a freeware tool for neuronal reconstruction (similar in some ways to part of Neurolucida's functionality). Neuromantic mainly uses SWC/Cvapp format, but the latest version can import and export MorphML.

Neurospaces/ GENESIS 3

The Neurospaces/ [GENESIS 3](#) project is developing a modular reimplementation of the core of GENESIS 2 along with a number of other components for computational neuroscience as part of the GENESIS 3 initiative. Neurospaces/GENESIS 3 currently supports reading of passive models in NeuroML format (morphology + passive parameters).

SplitNeuron

[SplitNeuron](#) is a library written in C for data structures and functions extending SQLite to simulate large-scale networks of Izhikevich Simple Model compartments. SplitNeuron answers a fundamental issue in large-scale simulation, data transfer between storage and functional software: it uses database not only for data storage but also as simulation engine, moving computation to data rather than using storage systems only for data holding. This choice offers more features with less code to write and a unique way of accessing data for further analysis. Features under development include direct import and cell/network creation from NeuroML.

NeurAnim

[NeurAnim](#) is a research aid for computational neuroscience. It is used to visualise and animate neural network simulations in 3D, and to render movies of these animations for use in presentations. Networks stored in the instance based representation of NetworkML can be loaded and visualised.

CNrun

[CNrun](#) is a neuronal network model simulator, similar in purpose to NEURON except that individual neurons are not compartmentalised. It was built from refactored code written by Thomas Nowotny. It reads in network topology description from a NeuroML file, where the `cell_type` attribute determines the unit class, one of the in-built neuron types of CNrun (e.g. Hodgkin Huxley cell by Traub and Miles (1991), Poisson oscillator, van der Pol oscillator).

NeuGen

[NeuGen](#) is an application in Java which is able to generate networks of synaptically connected morphologically detailed neurons, as in a cortical column. NeuGen generates sets of neurons of the different morphological classes of the cortex, e.g. pyramidal cells and stellate neurons, and connects these networks in 3D. The latest version of NeuGen can export the generated networks to NeuroML. Some manual editing of the generated files is required to make them valid. The developers have been informed of the required updates which will be incorporated soon.

morphforge

[morphforge](#) is a high level, simulator independent, Python library for building simulations of small populations of multi-compartmental neurons. It was built as part of the PhD thesis of Mike Hull (Uni. Edinburgh): Investigating the role of electrical coupling in small populations of interneurons in *Xenopus laevis* tadpoles. Loading of morphologies in MorphML format is supported, and loading of channel descriptions from ChannelML is in progress. Future development of morphforge will be closely aligned with the development of the multicompartmental modelling API in Python (libNeuroML).

NeuroTranslate

[NeuroTranslate](#) is a tool that translates input files between two different languages, the NCS (Neo-Cortical Simulator) input language and NeuroML format. It provides a user-friendly interface, which can be used to both create and edit simulations.

Moogli

[Moogli](#) (a sister project of [MOOSE](#)) is a simulator independent OpenGL based visualization tool for neural simulations. Moogli can visualize morphology of single/multiple neurons or network of neurons, and can also visualize activity in these cells. Loading of morphologies in MorphML and NeuroML formats is supported.

12.11.3 Approaches to adding NeuroML support

There are a number of ways that a neuronal simulator can add “support for NeuroML”, depending on how deeply it embeds/supports the elements of the language.

Commonly used approaches

1) Native support for NeuroML elements

A simulator may have an equivalent internal representation of the core concepts from NeuroML2/LEMS, and so be able to natively read/write these formats.

This is the approach taken in [jNeuroML](#) and [EDEN](#).

2) Native ability to import NeuroML elements

Another approach is for simulators to natively support importing (a subset of) NeuroML models, whereby the NeuroML components are converted to the equivalent entities in the simulator’s internal representation of the model.

This is the approach taken in [MOOSE](#), [Arbor](#) and [NetPyNE](#).

3) Native ability to export NeuroML elements

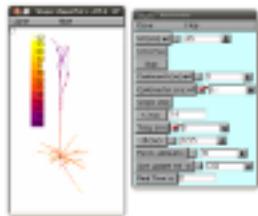
Some simulators allow models to be created with their preferred native model description format, and then exported in valid NeuroML.

This is the approach taken in [NEURON](#) and [NetPyNE](#). It is also possible to export [PyNN](#) models to NeuroML equivalents.

4) 3rd party mapping to simulator’s own format

This is the approach taken in [NEURON](#) via [jNeuroML](#).

12.11.4 NEURON and NeuroML



[NEURON](#) is a widely used simulation environment and is one of the main target platforms for a standard facilitating exchange of neuronal models.

Simulating NeuroML models in NEURON

jNeuroML or *pyNeuroML* can be used to convert NeuroML2/LEMS models to NEURON. This involves pointing at a *LEMS Simulation file* describing what to simulate, and using the `-neuron` option:

```
# Simulate the model using NEURON with python/hoc/mod files generated by jNeuroML
jnml <LEMS simulation file> -neuron -run

# Simulate the model using NEURON with python/hoc/mod files generated by pyNeuroML
pynml <LEMS simulation file> -neuron -run
```

These commands generate a PyNeuron script and run it (a file ending in `_nrn.py`). So you must have NEURON installed on your system, with its Python bindings (PyNeuron). Skipping the `-run` flag will generate the Python script but will not run it: you can run it manually later. Adding `-nogui` will suppress the NEURON graphical elements/menu opening and just run the model in NEURON in the background

You can also run LEMS simulations using the NEURON simulator using the *pyNeuroML* API:

```
from pyneuroml.pynml import run_lems_with_jneuroml_neuron
...
run_lems_with_jneuroml_neuron(lems_file_name)
```

Using neuroConstruct

NEURON simulations can also be generated from NeuroML model components by *neuroConstruct*, but most of this functionality is related to *NeuroML v1*.

12.11.5 NetPyNE and NeuroML



NetPyNE is a Python package to facilitate the development, simulation, parallelization, analysis, and optimization of biological neuronal networks using the NEURON simulator. NetPyNE can import from and export to NeuroML. NetPyNE also provides a web based Graphical User Interface.

Importing NeuroML into NetPyNE

An example of how to import a network in NeuroML into NetPyNE can be found [here](#).

Exporting NeuroML from NetPyNE

An example of how to export a network built using NetPyNE to NeuroML can be found [here](#).

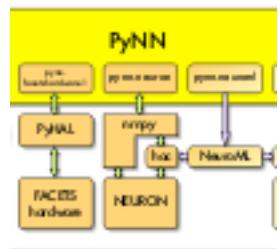
Running NetPyNE on OSBv2

Building and running NetPyNE models will be a core feature of Open Source Brain v2.0. See [here](#) for more details.

NeuroMLlite

NetPyNE is also a key target for cross simulator network creation using [*NeuroMLlite*](#). There are ongoing plans for greater alignment between formats used for network specification in NetPyNE and NeuroMLlite.

12.11.6 PyNN and NeuroML



PyNN is a Python package for simulator independent specification of neuronal network models. Model code can be developed using the PyNN API and then run using NEURON, NEST or Brian. The developed model also can be stored as a NeuroML document.

The latest version of [*neuroConstruct*](#) can be used to generate executable scripts for PyNN based simulators based on NeuroML components, although the majority of multicompartmental conductance based models which are available in neuroConstruct are outside the current scope of the PyNN API.

See <https://github.com/OpenSourceBrain/PyNNShowcase> for examples of usage of NeuroML and PyNN.

More info on the latest support for running NeuroML models in PyNN and vice versa can be found [here](#).

PyNN is also a key target for cross simulator network creation using [*NeuroMLlite*](#).

12.11.7 Brian and NeuroML



Brian is an easy to use, Python based simulator of spiking networks.

Converting NeuroML model to Brian

jNeuroML or *pyNeuroML* can be used to convert NeuroML2/LEMS models to Brian version 2. This involves pointing at a *LEMS Simulation file* describing what to simulate, and using the `-brian2` option:

```
# Using jnml
jnml <LEMS simulation file> -brian2

# Using pynml
pynml <LEMS simulation file> -brian2
```

This command generates a Python script (a file ending in `_brian2.py`) which can be run in Python and will simulate the model and plot/save the results, as outlined in the *LEMS Simulation file*.

Notes:

- Only single compartment cells can be converted to Brian format so far. While there is support in Brian for multi-compartmental cell simulation, this is not yet covered in the jNeuroML based export.
- There has been support for converting NeuroML models to Brian v1 (using `-brian`), but since this version of Brian is deprecated, and only supports Python 2, this export is no longer actively developed.
- There is limited support for executing networks of cells in Brian, and the most likely route for adding this functionality is via *NeuroMLlite*.

Examples

Example code for interactions between NeuroML models and Brian can be found [here](#).

12.11.8 MOOSE and NeuroML

MOOSE is the Multiscale Object-Oriented Simulation Environment. It is the base and numerical core for large, detailed multi-scale simulations that span computational neuroscience and systems biology. It is based on a complete reimplementation of the GENESIS 2 core.

Some tests of using MOOSE with NeuroML models and example code can be found in the MOOSE Showcase repository.

Simulating NeuroML models in MOOSE

You can export NeuroML models to the MOOSE simulator format using *jNeuroML* or *pyNeuroML*, pointing at a *LEMS Simulation file* describing what to simulate, and using the `-moose` option:

```
# Using jnml  
jnml <LEMS simulation file> -moose  
  
# Using pynml  
pynml <LEMS simulation file> -moose
```

12.11.9 EDEN and NeuroML

EDEN is a recently developed simulation engine which incorporates native NeuroML 2 support from the start.

Initial tests of using EDEN with NeuroML models and example code can be found [here](#).

12.11.10 Arbor and NeuroML



Arbor is a high performance multicompartmental neural simulation library. Addition of support for NeuroML2 and LEMS is under active development.

Importing NeuroML into Arbor

The current approach to supporting NeuroML in Arbor involves *importing NeuroML to Arbor's internal format*.

See [here](#) for Arbor's own documentation on this. It involves calling the `neuroml()` method in arbor pointing at the NeuroML file containing the cell you wish to load:

```
nml = arbor.neuroml('mymorphology.cell.nml')
```

See [here](#) for a worked example of this, importing a multicompartmental cell with only a passive membrane conductance.

Support for channels/synapses in LEMS

There is work under way to allow reading of the dynamics of ion channels and synapses which are specified in LEMS into Arbor.

See <https://github.com/thorstenhater/nmlcc> for more details.

Network models in Arbor with NeuroMLlite

There is preliminary support for building network specified in *NeuroMLlite* format directly in Arbor. See [here](#) for an example.

Examples

Example code for interactions between NeuroML models and Arbor can be found in the [Arbor Showcase](#) repository.

CHAPTER
THIRTEEN

CITING NEUROML AND RELATED PUBLICATIONS

This page documents how one can cite NeuroML in their work, and lists publications associated with the NeuroML initiative.

13.1 Citing NeuroML

Please cite NeuroML in your work whenever you have used it. Generally, you should cite the particular paper while discussing NeuroML in the text, and also note and cite the specific version of the NeuroML tool that has been used in the work.

13.1.1 Papers

Please cite the following papers as required:

NeuroML 2 and LEMS

The main citation for NeuroML 2

Please cite the following paper when discussing NeuroML v2.0 or LEMS.

Cannon RC, Gleeson P, Crook S, Ganapathy G, Marin B, Piasini E and Silver RA (2014) LEMS: A language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2, *Frontiers in Neuroinformatics* 8: 79.

```
@Article{Cannon2014,
  author    = {Robert C. Cannon and Padraig Gleeson and Sharon Crook and Gautham
              Ganapathy and Boris Marin and Eugenio Piasini and R. Angus Silver},
  title     = {{LEMS}: a language for expressing complex biological models in concise
              and hierarchical form and its use in underpinning {NeuroML} 2},
  doi       = {10.3389/fninf.2014.00079},
  volume    = {8},
  journal   = {Frontiers in Neuroinformatics},
  publisher = {Frontiers Media {SA}},
  year      = {2014},
}
```

libNeuroML and PyLEMS

Citation for Python & NeuroML

Please cite the following paper when using the Python NeuroML libraries

Vella M, Cannon RC, Crook S, Davison AP, Ganapathy G, Robinson HP, Silver RA and Gleeson P (2014) lib-NeuroML and PyLEMS: using Python to combine procedural and declarative modeling approaches in computational neuroscience. Frontiers in Neuroinformatics 8: 38.

```
@Article{Vella2014,
  author      = {Vella, Michael and Cannon, Robert C. and Crook, Sharon and Davison, Andrew P. and Ganapathy, Gautham and Robinson, Hugh P. C. and Silver, R. Angus and Gleeson, Padraig},
  title       = {libNeuroML and PyLEMS: using Python to combine procedural and declarative modeling approaches in computational neuroscience.},
  doi         = {10.3389/fninf.2014.00038},
  pages       = {38},
  volume      = {8},
  journal     = {Frontiers in neuroinformatics},
  year        = {2014},
}
```

NeuroML v1

Citation for NeuroML v1

Please cite the following paper when discussing NeuroML version 1. (deprecated)

Gleeson, P., S. Crook, R. C. Cannon, M. L. Hines, G. O. Billings, et al. (2010) NeuroML: A Language for Describing Data Driven Models of Neurons and Networks with a High Degree of Biological Detail. PLoS Computational Biology 6(6): e1000815.

```
@Article{Gleeson2010,
  author      = {Padraig Gleeson and Sharon Crook and Robert C. Cannon and Michael L. Hines and Guy O. Billings and Matteo Farinella and Thomas M. Morse and Andrew P. Davison and Subhasis Ray and Upinder S. Bhalla and Simon R. Barnes and Yoana D. Dimitrova and R. Angus Silver},
  title       = {{NeuroML}: A Language for Describing Data Driven Models of Neurons and Networks with a High Degree of Biological Detail},
  doi         = {10.1371/journal.pcbi.1000815},
  editor      = {Karl J. Friston},
  number      = {6},
  pages       = {e1000815},
  volume      = {6},
  journal     = {{PLOS} Computational Biology},
  publisher   = {Public Library of Science ({PLOS})},
  year        = {2010},
}
```

Open Source Brain

This paper describes version 1 of the Open Source Brain platform. Please cite this paper if you have made use of OSB in your work:

Gleeson P, Cantarelli M, Marin B, Quintana A, Earnshaw M, et al. (2019) Open Source Brain: a collaborative resource for visualizing, analyzing, simulating and developing standardized models of neurons and circuits. *Neuron* 103 (3):395–411

```
@Article{Gleeson2019,
  author    = {Padraig Gleeson and Matteo Cantarelli and Boris Marin and Adrian Quintana and Matt Earnshaw and Sadra Sadeh and Eugenio Piasini and Justas Birgiolas and Robert C. Cannon and N. Alex Cayco-Gajic and Sharon Crook and Andrew P. Davison and Salvador Dura-Bernal and Andr\'{e} Ecker and Michael L. Hines and Giovanni Idili and Frederic Lanore and Stephen D. Larson and William W. Lytton and Amitava Majumdar and Robert A. McDougal and Subhashini Sivagnanam and Sergio Solinas and Rokas Stanislovas and Sacha J. van Albada and Werner van Geit and R. Angus Silver},
  title     = {Open Source Brain: A Collaborative Resource for Visualizing, Analyzing, Simulating, and Developing Standardized Models of Neurons and Circuits},
  doi       = {10.1016/j.neuron.2019.05.019},
  number   = {3},
  pages    = {395--411},
  volume   = {103},
  journal  = {Neuron},
  publisher = {Elsevier {BV}},
  year     = {2019},
}
```

13.1.2 Software

It is important to cite software used in scientific work to:

- aid reproducibility of work
- to ensure that the developers of tools receive credit for their work.

You can learn more about Software Citation Principles as set out by the F1000 Software Citation working group in this work [SKNG16].

You can obtain the version of *pyNeuroML* and associated tools using the following command (with example output):

```
$ pynml --version
pyNeuroML v0.5.20 (libNeuroML v0.3.1, jNeuroML v0.11.1)
```

Each NeuroML software tool has a unique DOI and reference associated with each release at the [Zenodo archival facility](#). On each entry, you will be able to find the DOI and citation of the particular version you are using, and you will also be able to download the citation in different formats at the bottom of the right hand side bar.

13.2 Other publications

This section lists other publications related to NeuroML.

Günay, C. et al. (2008) Computational intelligence in electrophysiology: Trends and open problems. In Smolinski, Milanova and Hassanien, eds. Applications of Computational Intelligence in Biology. Springer, Berlin/Heidelberg.

Gleeson, P., V. Steuber, and R. A. Silver (2007) neuroConstruct: A Tool for Modeling Networks of Neurons in 3D Space. *Neuron*. 54(2):219-235.

Cannon R. C., M. O. Gewaltig, P. Gleeson, U. S. Bhalla, H. Cornelis, M. L. Hines, F. W. Howell, E. Muller, J. R. Stiles, S. Wils, E. De Schutter (2007) Interoperability of Neuroscience Modeling Software: Current Status and Future Directions. *Neuroinformatics* Volume 5, 127-138.

Crook, S., P. Gleeson, F. Howell, J. Svitak and R.A. Silver (2007) MorphML: Level 1 of the NeuroML standards for neuronal morphology data and model specification. *Neuroinformatics*. 5(2):96-104.

Crook, S. and F. Howell (2007) XML for data representation and model specification in neuroscience. In Methods in Molecular Biology Book Series: *Neuroinformatics*. ed. C. Crasto, Humana Press.

Crook, S., D. Beeman, P. Gleeson and F. Howell (2005) XML for model specification in neuroscience: An introduction and workshop summary. *Brains, Minds, and Media*. 1:bmm228 (urn:nbn:de:0009-3-2282).

Qi, W. and S. Crook (2004)

Tools for neuroinformatic data exchange: An XML application for neuronal morphology data. *Neurocomputing*. 58-60C:1091-1095.

Goddard, N., M. Hucka, F. Howell, H. Cornelis, K. Shankar and D. Beeman (2001) Towards NeuroML: Model description methods for collaborative modeling in neuroscience. *Philosophical Transactions of the Royal Society B*. 356:1209-1228.

13.2.1 Book Chapters

Crook, SM, HE Plesser, AP Davison (2013) Lessons from the past: approaches for reproducibility in computational neuroscience. In JM Bower, ed. *20 Years of Computational Neuroscience*, Springer

Gleeson, P, V Steuber, RA Silver and S Crook (2012) NeuroML. In Le Novere, ed. *Computational Systems Biology*, Springer.

13.2.2 Abstracts

Cannon, R, P Gleeson, S Crook, RA Silver (2012) A declarative model specification system allowing NeuroML to be extended with user-defined component types. *BMC Neuroscience*. 13(Suppl 1): P42.

Gleeson P, S Crook, A Silver, R Cannon (2011) Development of NeuroML version 2.0: Greater extensibility, support for abstract neuronal models and interaction with Systems Biology languages. *BMC Neuroscience*. 12:P29.

Gleeson, P., S. Crook, S. Barnes and R.A. Silver (2008)

Interoperable model components for biologically realistic single neuron and network models implemented in NeuroML. *Frontiers in Neuroinformatics*. Conference Abstract: *Neuroinformatics 2008*. doi: [10.3389/conf.neuro.11.2008.01.135](https://doi.org/10.3389/conf.neuro.11.2008.01.135).

Larson, S. and M. Martone (2008)

A spatial framework for multi-scale computational neuroanatomy. *Frontiers in Neuroinformatics*. Conference Abstract: *Neuroinformatics 2008*. doi: [10.3389/conf.neuro.11.2008.01.134](https://doi.org/10.3389/conf.neuro.11.2008.01.134).

Crook, S., P. Gleeson and R.A. Silver (2007) NetworkML: Level 3 of the NeuroML standards for multiscale model specification and exchange. Society for Neuroscience Abstracts. 102.28.

Gleeson, P., S. Crook, V. Steuber and R.A. Silver (2007)

Using NeuroML and neuroConstruct to build neuronal network models for multiple simulators. BMC Neuroscience. 8(2):P101.

CHAPTER
FOURTEEN

FREQUENTLY ASKED QUESTIONS (FAQ)

Please help improve the FAQ.

This page lists some commonly asked questions related to NeuroML. Please feel free to [open issues](#) to add more entries to this FAQ.

14.1 Are length 0 segments allowed in NeuroML?

Discussion link: <https://github.com/NeuroML/NeuroML2/issues/115>

There are a lot of SWC reconstructions which have adjacent points, which would get converted to zero length segments. This shouldn't be an issue for most visualisation applications, so no need for them to say that they can't visualise the cell if they see it's invalid.

The `jnml -validate` option could throw a warning when it sees these segments, but currently doesn't (it could be added [here](#)).

For individual simulators, they could have an issue with this, if they map each segment to a compartment (as Moose might), but for Neuron using cables/sections with multiple segments, it shouldn't matter as long as the section doesn't just have one segment.

So ideally it should be the application which loads the NeuroML in (or the conversion/export code) which decides whether this is an issue.

Part II

NeuroML events

JUNE 2022: NEUROML TUTORIAL AT CNS*2022 SATELLITE TUTORIALS

An online NeuroML tutorial will be held at the [CNS*2022 satellite tutorials](#). Registration for the satellite tutorials is free, but required.

This tutorial is intended for members of the research community interested in learning more about how NeuroML and its related technologies facilitates the standardization, sharing, and collaborative development of models.

15.1 Times and dates

- Dates: June 30, 2022
- Time: : 1400–1700 UTC

15.2 Target audience

Anyone who is already familiar with computational modelling, but is keen to standardise, share and collaboratively develop their models.

15.3 Where

The tutorial be done online via Zoom and will make use of the [Open Source Brain v2](#) integrated web research platform. Please register for the [CNS*2022 satellite tutorials](#) to receive the Zoom links.

15.4 Agenda

15.4.1 Part 1: Introduction to NeuroML

- Overview of NeuroML
- Introduce the NeuroML tool chain
- Introduce main documentation
- Related technologies and initiatives

15.4.2 Part 2: Hands on demonstrations of building and using NeuroML models

- Izhikevich neuron hands on tutorial
- Spiking neuron network tutorial
- Single compartment HH neuron tutorial
- Multi compartmental HH neuron tutorial

CHAPTER
SIXTEEN

APRIL 2022: NEUROML DEVELOPMENT WORKSHOP AT HARMONY 2022

Registration for the COMBINE initiative's HARMONY 2022 meeting is free.

Please register for the COMBINE HARMONY 2022 meeting [here](#) if you are coming to our NeuroML workshop. Registration for HARMONY is free.

We will be running a NeuroML development workshop during the upcoming **COMBINE network's HARMONY 2022** meeting on **Thus 28 April 2022**. This will be an opportunity for anyone interested in developing NeuroML or adding support for the format to their application talk about their work and hear about other developments.

16.1 Agenda

The agenda for the meeting can be found [here](#).

16.2 Times and dates

The workshop will take place on **Thus 28 April 2022** at 15:00-18:00 UTC ([converter](#)).

16.3 Registration

To take part in the workshop, please register [here](#) for the HARMONY meeting (registration is free).

You will get sent details to access the HARMONY agenda, which will have links to the **Zoom session for the NeuroML workshop**.

16.4 Open an issue beforehand!

While it will be possible to raise and discuss new issues at the workshop, it will be easier to manage and plan work/discussions if you open an issue with a description of the problem you are trying to address at: <https://github.com/NeuroML/NeuroML2/issues>.

16.5 Slack

To aid communication with the community during (and after) the meeting, we have a **Slack channel** for NeuroML related discussions. Please contact [Padraig Gleeson](#) for an invite.

We look forward to working with the community to drive further uptake of NeuroML compliant models and tools!

CHAPTER
SEVENTEEN

OCTOBER 2021: NEUROML DEVELOPMENT WORKSHOP AT COMBINE MEETING

Registration for the COMBINE 2021 meeting is free.

Register for the COMBINE 2021 meeting [here](#). Registration is free.

A NeuroML development workshop will be held as part of the [annual COMBINE meeting](#) in October 2021.

The general theme of the workshop is to discuss the current status of the NeuroML standard and the complete software ecosystem, and future development plans.

17.1 Times and dates

- 13 October 2021
- 8-11am PDT/11-2pm EST/4-7pm UK/5-8pm CET/8:30-11:30 IST

17.2 Target audience

Everyone that is involved/interested in developing tools that use/integrate with NeuroML is encouraged to join.

Please register for the COMBINE meeting (free of charge) to receive access to the complete schedule of the meeting, including links to the various virtual meetings/sessions.

17.3 Agenda/minutes

The agenda/minutes for the meeting can be found [here](#).

CHAPTER
EIGHTEEN

AUGUST 2021: NEUROML TUTORIAL AT INCF TRAINING WEEKS

A NeuroML tutorial will be held at the [Virtual INCF Neuroinformatics Training Weeks 2021](#).

This tutorial is intended for members of the research community interested in learning more about how NeuroML and its related technologies facilitates the standardization, sharing, and collaborative development of models.

18.1 Times and dates

This tutorial will be offered twice during the Neuroinformatics Training Week: session 1 is targeted to participants residing in Europe, Africa, and the Americas while session 2 is targeted to participants residing in Asia and Australia.

Session 1:

- Dates: 23 Aug 2021
- Time: : 11:00-15:00 EDT / 17:00-21:00 CEST

Session 2

- Dates: 26 Aug 2021
- Time: 09:00-13:00 CEST / 16:00-20:00 JST / 17:00-21:00 AEST

18.2 Target audience

Anyone who is already familiar with computational modelling, but is keen to standardise, share and collaboratively develop their models.

18.3 Agenda

18.3.1 Part 1: Introduction to NeuroML

- Overview of NeuroML
- Introduce the NeuroML tool chain
- Introduce main documentation
- Related technologies and initiatives

18.3.2 Part 2: Hands on demonstrations of building and using NeuroML models

- Izhikevich neuron hands on tutorial
- Spiking neuron network tutorial
- Single compartment HH neuron tutorial
- Multi compartmental HH neuron tutorial

JULY 2021: NEUROML TUTORIAL AT CNS*2021

Register for the 30th Annual meeting of the Organization for Computational Neurosciences (OCNS).

Register for the CNS*2021 [here](#).

We will be running a half day tutorial at the 30th annual meeting of the Organization for Computational Neurosciences (OCNS): [CNS*2021](#).

The goal of the tutorial is to teach users to: **build, visualise, analyse and simulate models using NeuroML**.

19.1 Why take part?

This tutorial is aimed at new and current NeuroML users. We will start with a quick introduction to the NeuroML standard and the associated software ecosystem, after which we will proceed to conduct hands-on sessions to show how one can build computational models with NeuroML.

19.2 Times and dates

- Friday 2nd July 1500UTC.

19.3 Registration

To take part in the tutorial, please register [here](#) for the CNS*2021 meeting.

19.4 Pre-requisites

The sessions will make use of the NeuroML Python tools. Please follow the documentation to install them on your system if you wish to use them locally:

- [PyNeuroML](#)
- [libNeuroML](#)

You can also use the interactive Jupyter notebooks from the documentation if you prefer ([example](#)). These can be run on Binder and Google Collab in your web browser and do not require you to install anything locally on your computer.

19.5 Slack

To aid communication with the community during (and after) the meeting, we have a **Slack channel** for NeuroML related discussions. Please contact [Padraig Gleeson](#) for an invite.

You can also contact the NeuroML community using one of our other [*channels*](#).

We look forward to working with the community to drive further uptake of NeuroML compliant models and tools!

MARCH 2021: NEUROML HACKATHON AT HARMONY 2021

Registration for the COMBINE initiative's HARMONY 2021 meeting is free.

Register for the COMBINE: HARMONY 2021 meeting [here](#). Registration is free.

We will be running 3 online NeuroML hackathon sessions during the upcoming COMBINE: HARMONY 2021 meeting on 23-25th March. The general theme of the sessions will be: **learn to build, visualise, analyse and simulate your models using NeuroML**.

20.1 Why take part?

These hackathons will give members of the neuroscience community the chance to:

- Get high level introductions to the NeuroML language and tool chain
- Meet the NeuroML core development team and editors
- Find out the latest information on which simulators/applications support NeuroML
- Open, discuss and work on issues related to converting your model to NeuroML, or supporting NeuroML in your simulator
- Learn how to share your models with the community

20.2 Times and dates

All sessions will be online and take place over 3 hours (9am-noon Pacific; 12-3pm EST time; 4-7pm UK/UTC; 5-8pm CET, 9:30pm-12:30am IST; note non-standard US/EU time differences that week). The broad focus of each of the sessions (dependent on interests of attendees) is:

- Tues 23rd March: Introduction to NeuroML, general questions about usage
- Wed 24th March: Detailed cell/conductance based models (e.g. converting channels to NeuroML)
- Thus 25th March: Abstract/point neuron networks including PyNN interactions

20.3 Registration

To take part in the hackathon, please register [here](#) for the HARMONY meeting (registration is free). You will get sent details to access the [agenda](#), which will have links to the Zoom sessions for each of the days.

20.4 Open an issue beforehand!

While it will be possible to raise and discuss new issues at the hackathons, it will be easier to manage and plan work/discussions if you open an issue with a description of the problem you are trying to address at: <https://github.com/NeuroML/NeuroML2/issues>.

20.5 Slack

To aid communication with the community during (and after) the meeting, we have a **Slack channel** for NeuroML related discussions. Please contact [Padraig Gleeson](#) for an invite.

We look forward to working with the community to drive further uptake of NeuroML compliant models and tools!

CHAPTER
TWENTYONE

PAST NEUROML EVENTS

A number of developer workshops and editorial board meetings have been held since 2008 to coordinate and promote the work of the NeuroML community. These are listed [here](#).

There has been significant NeuroML involvement also at the meetings organised by the Open Source Brain initiative. See [here](#) for more details.

Part III

The NeuroML Initiative

CHAPTER
TWENTYTWO

GETTING IN TOUCH

We're happy to talk with users, developers and modellers about using NeuroML in their work.

22.1 Mailing list

For general discussion, queries, and troubleshooting related to NeuroML please use the mailing list: <https://lists.sourceforge.net/lists/listinfo/neuroml-technology>.

22.2 Chat channels

Chat channels for quick queries are also available on:

- [Gitter](#)
- [Matrix/Element](#)

The two rooms are bridged, so you can use either. Please note that activity in these rooms depends on time zones and the availability of community members. So, if you do not get a response soon, please post to the mailing list listed above or file an issue on GitHub as noted below.

22.3 Issues related to the libraries or specification

- Please file general issues related to NeuroML at the [NeuroML/NeuroML2](#) repository on GitHub.
- Please file issues related to LEMS and jLEMS at the [LEMS/jLEMS](#) repository on GitHub.
- Additionally, please file issues related to the different NeuroML core tools at their individual [GitHub repositories](#).

22.4 Social media

You can follow NeuroML related updates on Twitter at [@NeuroML](#).

CHAPTER
TWENTYTHREE

OVERVIEW OF STANDARDS IN NEUROSCIENCE

23.1 NeuroML as a standard

NeuroML is an [INCF endorsed standard](#).

NeuroML is a [COMBINE official standard](#).

Work in progress

This page is a work in progress...

CHAPTER
TWENTYFOUR

A BRIEF HISTORY OF NEUROML

24.1 The early days

The concept of NeuroML was first introduced in an article by Goddard et al. (2001) [GHH+01], following meetings at the University of Edinburgh where initial templates and an overall structure for a model description language for computational modelling in neuroscience were discussed. The proposal extended general purpose structures for neuroscience data proposed by Gardner et al. (2001) [GKA+01].

At that time, the design principles for NeuroML were closely linked with a specific software architecture in which a base application loads a range of plug-ins to handle different aspects of a simulation experiment. The simulation platform **Neosim** provided an implementation of this approach (Howell et al. 2003 [HCG+03]), and early NeuroML development was closely aligned to this architecture. Fred Howell and Robert Cannon developed a software library, the NeuroML Development Kit (NDK), to simplify the process of working with XML serializations of models. This library implemented a particular dialect of XML but did not define particular structures at the model description level. Instead, Neosim plug-in developers were free to develop their own structures and serialize them via the NDK, in the hope that some consensus would emerge around the most useful ones.

In practice, few developers beyond the Edinburgh group developed or used such structures and the resulting XML was too application specific to gain wider adoption. The Neosim project was completed in 2005.

24.2 NeuroML v1.x

Based on discussions with Howell and Cannon about the need to develop a consensus for describing widely used model components, Sharon Crook worked with the neuroanatomy community on a language for describing neuronal morphologies in XML, **MorphML** (Qi and Crook 2004 [QC04]). At the same time, Padraig Gleeson, working with Angus Silver, was developing **neuroConstruct**, for generating neuronal simulations for the NEURON and GENESIS simulators (Gleeson et al. 2007 [GSS07]), which had its own internal simulator independent representation for morphologies, channel and networks.

It was agreed that these efforts should be merged under the banner of NeuroML, and the *v1.x structure of NeuroML* was created. A modular approach containing **MorphML**, **ChannelML** and **NetworkML** was adopted to allow application developers to support only those parts of the language needed by their application (Crook et al. 2007 [CGH+07], Gleeson et al. 2010 [GCC+10]). XML schema files for this version of the standard have been available since 2006. The motivation, structure and functionality of this version is described in detail in Gleeson et al. 2010, while the specification of the language is outlined in the **Supporting Information** of that publication.

24.3 NeuroML v2.x - introducing LEMS...

NeuroML2 development was started in 2011. The main motivation for NeuroML2 was the lack of extensibility of NeuroML v1.x; every new model type which was introduced into the language required an update to the Schema, updates to the text documentation and an implementation in each of the native formats of the target simulators. NeuroML2 is built on the **LEMS (Low Entropy Model Specification) language**, which allows machine readable definitions of the cell, channel and synapse models which form the core of the language. This increases transparency of model structure and dynamics and facilitates automatic mapping of the models to multiple simulation formats. More details on the structure of LEMS and how it is used in NeuroML2 can be found in Cannon et al. 2014 [CGC+14] and [here](#).

In parallel with development of NeuroML2 and LEMS, software libraries for reading, writing and running simulations using the languages are under active development in Java ([jNeuroML](#)) and Python ([libNeuroML](#) and [pyLEMS](#) (see Vella et al. 2014 [VCC+14]) and [pyNeuroML](#)).

The NeuroML specifications are currently being developed by the [NeuroML Editorial Board](#), overseen by the NeuroML specifications are developed by the [NeuroML Editorial Board](#) and overseen by its [Scientific Committee](#). NeuroML specifications and the associated libraries are developed on GitHub and an overview of current activities can be found [here](#).

Recent releases of NeuroML2

For full details on the recent releases of NeuroML see: [here](#).

24.4 The future

[NeuroMLlite](#) is under active development, which will significantly enhance the range of network models which can be expressed (in a concise JSON based format) and run in NeuroML supporting simulators. This work will form the basis of NeuroML v3.0.

CHAPTER
TWENTYFIVE

NEUROML EDITORIAL BOARD

An elected board of editors has been formed to manage the NeuroML specification development process. The editorial board consists of five members, elected by the NeuroML community. The editors are responsible for producing and maintaining the official documentation for the NeuroML language, and work in collaboration with the *Scientific Committee* who provide oversight and guidance.

Due to the close link between the development of NeuroML 2 and LEMS, this group is also responsible for producing a stable specification for the subset of LEMS used by NeuroML 2.

25.1 Current Editorial Board

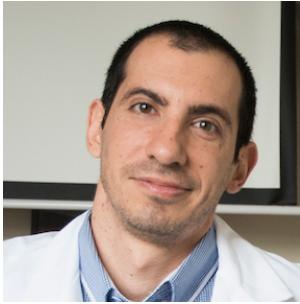
The five current members of the NeuroML Editorial Board are:

- Justas Birgiolas
- Salvador Dura-Bernal
- Padraig Gleeson
- Boris Marin
- Ankur Sinha

Padraig Gleeson, Boris Marin and Justas Birgiolas were elected for three year terms in 2019 (2020–2022) and Salvador Dura-Bernal and Ankur Sinha were elected for three year terms in 2021 (2022–2024).



Justas Birgiolas Independent Researcher Buffalo, NY [Website](#)



Salvador Dura-Bernal SUNY Downstate Brooklyn, USA [Website](#)



Padraig Gleeson University College London UK [Website](#)



Boris Marin Universidade Federal do ABC Brazil [Website](#)



Ankur Sinha University College London UK [Website](#)

Information on past editors and elections can be found [here](#).

25.2 Procedures

The procedures for election of the editorial board, its responsibilities, size and activities were heavily inspired by other initiatives like [SBML](#), that have had successful editorial teams for many years. The [COMBINE initiative](#) seeks to promote community developed standards in computational biology, and the NeuroML editorial board will work with this initiative to ensure best practices in specification preparation.

25.3 Responsibilities of NeuroML Editors

Their main responsibilities are:

- Defining and documenting the procedure for specification production: scope, release frequency, update procedures, form of specification (web based or single pdf, etc.). This should be based on some or all of the recommendations for community based standards development from the COMBINE initiative. These procedures for specification production will have to be agreed with the [Scientific Committee](#).
- Preparing the core specification for the NeuroML language.
- Testing reference implementations of NeuroML compliant applications.
- Preparing a specification for the LEMS language. This can be a subset of the language supported by the reference implementations in Java (jLEMS) and Python (pyLEMS), but will have to cover all of the LEMS elements required to specify the ComponentType definitions for NeuroML 2.
- Responding to community queries about the specification.
- Establishing a procedure for incorporating major changes into the specification (in cooperation with the [Scientific Committee](#)).

Participation in the editorial board will be on a volunteer basis, there is no central funding to support this work.

25.4 History of the NeuroML Editorial Board

This page documents the previous members of the NeuroML Board.

25.4.1 Initial election of editors (2013)

The first election of an editorial board for NeuroML took place in May/June 2013.

- The electorate consisted of the members of the NeuroML mailing lists on 3rd May 2013.
- Anyone on these lists could nominate someone to be an editor. Self nominations were also allowed.
- The three candidates who received the highest number of votes will serve three year terms and the two with the next highest number of votes will serve two year terms.
- Nicolas Le Novère ([lenov -at- babraham.ac.uk](mailto:lenov-at-babraham.ac.uk)) was the returning officer for this initial election.

25.4.2 Election of editors (2015)

The second election of an editorial board for NeuroML took place in June/July 2015.

- The electorate consisted of the members of the NeuroML mailing lists on 18 June 2015.
- Anyone on these lists could nominate someone to be an editor. Self nominations were also allowed.
- The two candidates who received the highest number of votes would serve three year terms.
- Nicolas Le Novère ([lenov -at- babraham.ac.uk](mailto:lenov-at-babraham.ac.uk)) was the returning officer for this election.
- Results were announced [here](#).

25.4.3 Election of editors (2016)

The third election of an editorial board for NeuroML took place in July/August 2016.

- The electorate consisted of the members of the NeuroML mailing lists on 18 July 2016.
- Anyone on these lists could nominate someone to be an editor. Self nominations were also allowed.
- The three candidates who received the highest number of votes would serve three year terms.
- Nicolas Le Novère ([lenov -at- babraham.ac.uk](mailto:lenov-at-babraham.ac.uk)) was the returning officer for this election.

25.4.4 Election of editors (2018)

The fourth election of an editorial board for NeuroML took place in Nov/Dec 2018.

- The electorate consisted of the members of the NeuroML mailing lists as well as anyone who had made significant contributions to any of the NeuroML GitHub repositories in the past 3 years.
- Anyone on the electorate could nominate someone to be an editor. Self nominations were also allowed.
- The two candidates who received the highest number of votes would serve three year terms.
- Salvador Dura-Bernal was elected outright on the first round of voting and Andrew Davison was elected in a run off between the two next highest placed candidates who received the same number of votes.
- Malin Sandstrom at the INCF was the returning officer for this election.

25.4.5 Election of editors (2019)

The fifth election of an editorial board for NeuroML took place in Nov/Dec 2019.

- The electorate consisted of the members of the NeuroML mailing lists as well as anyone who had made significant contributions to any of the NeuroML GitHub repositories in the past 3 years.
- Anyone on the electorate could nominate someone to be an editor. Self nominations were also allowed.
- The three candidates who received the highest number of votes would serve three year terms.
- Padraig Gleeson, Boris Marin and Justas Birgiolas were nominated, and eventually all elected to serve as editors.
- Malin Sandstrom at the INCF was the returning officer for this election.

25.4.6 Election of editors (2022)

The sixth election of an editorial board for NeuroML took place in Nov/Dec 2021.

- The electorate consisted of the members of the NeuroML mailing lists as well as anyone who had made significant contributions to any of the NeuroML GitHub repositories in the past 3 years.
- Anyone on the electorate could nominate someone to be an editor. Self nominations were also allowed.
- The three candidates who received the highest number of votes would serve three year terms.
- Salvador Dura-Bernal and Ankur Sinha were nominated, and eventually all elected to serve as editors.
- Sharon Crook was the returning officer for this election.

25.5 Workshop and Meeting reports

Information and minutes of various NeuroML meetings can be found [here](#).

Meeting	Location	Summary
2021 NeuroML Development workshop & Editorial Board Meeting	Online at COMBINE 2021	A NeuroML development workshop was held as part of the annual COMBINE meeting in October 2021. More info .
2019 NeuroML Editorial Board Meeting	CNS*2019 Meeting, Barcelona	The fifth NeuroML Editorial Board Meeting took place at the CNS meeting in Barcelona, Monday 15th July, 2019. Minutes .
2018 NeuroML Editorial Board Meeting	Online	The fourth NeuroML Editorial Board Meeting took place via video conference on 6th July 2018 between the NeuroML Editorial Board and interested members of the community to get an update on all current NeuroML related activities. Minutes .
2016 NeuroML Editorial Board Meeting	Janelia Research Campus, Virginia, USA	The third NeuroML Editorial Board Meeting took place after the Collaborative Development of Data-Driven Models of Neural Systems conference held at Janelia Research Campus in Sept 2016. More details on the main conference can be found here . Minutes .
2015 NeuroML Editorial Board Meeting @ OSB 2015	Alghero, Sardinia, Italy	The second NeuroML Editorial Board Meeting took place prior to the Open Source Brain 2015 meeting held in Sardinia. More details on the main meeting can be found here . Minutes .
2014 NeuroML Editorial Board Meeting @ OSB 2014	Alghero, Sardinia, Italy	The first official NeuroML Editorial Board Meeting took place prior to the Open Source Brain 2014 meeting held in Sardinia. More details on the main meeting can be found here . Minutes .
2013 NeuroML Meeting Development Workshop @ OSB 2013	Alghero, Sardinia, Italy	The NeuroML Development Workshop was merged into the Open Source Brain kickoff meeting in Alghero, Sardinia. More details on this meeting can be found here . Discussions on the state of NeuroML and future developments took place during the main meeting.
2012 NeuroML Development Workshop	Informatics Forum, Edinburgh, UK	The NeuroML workshop at was combined with the BrainScaleS (previously FACETS) CodeJam meeting. Minutes .
2011 NeuroML Development Workshop	University College London, UK	A key outcome of third NeuroML Development Workshop was the creation of a Scientific Committee for NeuroML. Minutes .
2010 NeuroML Development Workshop	Arizona State University, USA	The second NeuroML Development Workshop was held in Arizona State University to plan for version 2.0 of the NeuroML model description language. There was also a Symposium on Multiscale Approaches to Understanding Neural Plasticity held at ASU before the main meeting and a number of tutorials on software for multiscale modeling given by the meeting participants on the following day. Minutes .
2009 NeuroML Development Workshop	University College London, UK	The focus of the workshop was to refine the specifications for describing models of channel kinetics and the biophysical properties of cells. Special thanks to the Wellcome Trust, the INCF, and the NSF for their generous support of this endeavour. Minutes .
2008 CNS Workshop	Portland, Oregon, USA	Padraig Gleeson and Sharon Crook moderated a workshop on “Interoperability of Software for Computational and Experimental Neuroscience” at the 2008 Computational Neuroscience Meeting.

CHAPTER
TWENTYSIX

NEUROML SCIENTIFIC COMMITTEE

The responsibilities of the NeuroML Scientific Committee are:

- To advise on the scientific focus of the NeuroML initiative; to ensure that the structure of the language is based on the latest knowledge of neuronal anatomy and physiology.
- To agree on the technical implementation for the core specifications (in collaboration with the *NeuroML Editorial Board*) and to ensure that best practices are encouraged in model specification.
- To promote NeuroML internationally, both the core specifications and the tools which support the language.
- To define the governance structure of the NeuroML Initiative and outline a path towards a specification process with dedicated, elected editors.
- To engage with other standardisation and databasing initiatives in the computational neuroscience and wider biology fields.
- To review and agree on extensions to the core specifications and the scope of the initiative; to address issues the community raises regarding the direction of the initiative.

26.1 Current Members



Upi Bhalla NCBS Bangalore, India [Website](#)



Avrama Blackwell Krasnow Institute of Advanced Studies George Mason University, USA [Website](#)



Hugo Cornells K.U. Leuven Belgium [Website](#)



Sharon Crook Arizona State University USA [Website](#)



Andrew Davison CNRS, Gif-sur-Yvette France [Website](#)



Robert McDougal Yale University USA [Website](#)



Lyle Graham Université Paris Descartes Paris, France [Website](#)



Cengiz Gunay Georgia Gwinnett College USA [Website](#)



Michael Hines Yale University USA [Website](#)



Angus Silver University College London London, UK [Website](#)

26.2 Past Members

(Note: past members who are currently members of the *NeuroML Editorial Board* are not listed.)

- Robert Cannon

CHAPTER
TWENTYSEVEN

FUNDING AND ACKNOWLEDGEMENTS

The NeuroML effort has been made possible by funding from research councils in the UK, EU, and the USA.



UK Medical Research Council



UK Biotechnology and Biological Sciences Research Council



National Institutes of Health



EU Synapse Project



National Science Foundation



International Neuroinformatics Coordinating Facility



Wellcome

CHAPTER
TWENTYEIGHT

NEUROML CONTRIBUTORS

This page lists contributors to the various NeuroML and related repositories, listed in no particular order. It is generated periodically, most recently on 01/04/21. See also the current *NeuroML Editorial Board* and the *Scientific Committee*.

- @pgleeson
- @tarelli
- @borismarin
- @mattearnshaw
- @sanjayankur31
- @adrianq
- @epiasini
- @RokasSt
- @dilawar
- @russelljjarvis
- @FinnK
- @kapilkd13
- @gidili
- @wvangeit
- @hugh-osborne
- @lungd
- @orena1
- @JustasB
- @ChihweiLHBird
- @rgerkin
- @ccluri
- @34383c
- @andrisecker
- @jriekе
- @scrook
- @jorc125

- [@lisphacker](#)
- [@dokato](#)
- [@robertcannon](#)
- [@waffle-iron](#)
- [@vellamike](#)
- [@clbarnes](#)
- [@mattions](#)
- [@apdavison](#)
- [@lebedov](#)
- [@baladkb](#)
- [@jefferis](#)
- [@mstimberg](#)
- [@arosh](#)
- [@unidesigner](#)

28.1 Repositories

- [NeuroML/org.neuroml.model](#)
- [NeuroML/org.neuroml.model.injectingplugin](#)
- [NeuroML/org.neuroml1.model](#)
- [NeuroML/org.neuroml.visualiser](#)
- [NeuroML/NeuroML2](#)
- [NeuroML/jNeuroML](#)
- [NeuroML/org.neuroml.export](#)
- [NeuroML/org.neuroml.import](#)
- [NeuroML/NeuroMLToPOV-Ray](#)
- [NeuroML/NML2_LEMS_Examples](#)
- [NeuroML/NeuroMLWebsite](#)
- [NeuroML/pyNeuroML](#)
- [NeuroML/neuroml2model](#)
- [NeuroML/Presentations](#)
- [NeuroML/NeuroMLToolbox](#)
- [NeuroML/NeuroML_API](#)
- [NeuroML/NetworkShorthand](#)
- [NeuroML/mod2neuroml](#)
- [NeuroML/neuroml2modelLite](#)

- NeuroML/NeuroMLlite
- NeuroML/Documentation
- LEMS/pylems
- LEMS/jLEMS
- LEMS/LEMS
- LEMS/org.lemsml.model
- LEMS/expr-parser
- LEMS/lems-domogen-maven-plugin
- NeuralEnsemble/libNeuroML

CHAPTER
TWENTYNINE

CODE OF CONDUCT

Everyone is welcome in the NeuroML community. We request everyone interacting on the NeuroML channels in any capacity to treat each other respectfully. Please:

- act in good faith
- be friendly, welcoming, respectful, and patient
- be mindful and considerate
- be open, use prefer and promote Open Science practices.

If you experience or become aware of behaviour that does not adhere to the Code of Conduct, please contact the moderators of the channel/event you are in.

Part IV

Developer documentation

OVERVIEW

This section will contain information for those who wish to **contribute to the development** of the NeuroML standard and associated tools.

An overview of the NeuroML **release process** can be found [here](#).

The relationship of NeuroML to a number of other tools and standards in computational neuroscience, and the practical steps taken thus far to ensure interoperability, can be found [here](#).

30.1 Contribution guidelines

Thank you for your interest in contributing to NeuroML. Welcome!

This page documents the contribution guidelines for all NeuroML related repositories.

Please do remember that these are *guidelines* but not rules that must be strictly followed. We think these are reasonable ideas to follow and they help us maintain a high code quality while making it easier and more efficient for all of us to work together. However, there may be cases where they can not be followed, and that's fine too.

30.1.1 Code of conduct

All NeuroML projects are governed by the [Code of Conduct](#). By participating, you are expected to uphold this code. Please report unacceptable behaviour to the moderators of the communication channel you are in.

30.1.2 Structure of repositories

- All NeuroML repositories use the [Git](#) version control system.
- Contributions are made using [pull requests](#).
- Each NeuroML software tool resides in its own GitHub repository under the [NeuroML GitHub Organization](#), apart from [libNeuroML](#) which is developed in collaboration with the [NeuralEnsemble](#) community and so lives under their GitHub organization.
- LEMS repositories are housed under the [LEMS GitHub Organization](#).

You can find links to these on the respective pages for each [software tool](#).

The NeuroML standard itself (schema and ComponentType definitions) is housed in its own repository [here](#).

(devdocs:devsop:repos:zenhub)

Kanban board on Zenhub

An overview of the various repositories, tasks, issues, and so on can be seen on the [NeuroML Kanban board on Zenhub](#).

30.1.3 Versioning

All NeuroML repositories (including the standard) follow Semantic versioning. This means that the version string consists of three components: MAJOR.MINOR.PATCH:

- the MAJOR version is incremented when incompatible API changes are made,
- the MINOR version is incremented when functionality is added in a backwards compatible manner, and
- the PATCH version is incremented when backwards compatible bug fixes are made.

30.1.4 Git branches

- Please develop against the development branch in all repositories. This branch is merged into master via a pull request when a new release is made. This ensures that all tests are run at each step to verify correctness. As a result, the master branch of all repositories holds the stable version of the standard and tools, while the development branch holds the next, unstable version that is being worked upon.
- For branch names, please consider using the [Git flow](#) naming convention (not mandatory but strongly suggested):
 - prefix feature branches with feat/ or enh/ (for enhancement)
 - prefix bugfix branches with bugfix/ or fix/
 - pull requests addressing specific tickets may also mention them in the branch name. E.g., bugfix/issue-22.

30.1.5 Git commits

Git commit messages are extremely important because they allow us to nicely track the complete development history of the project. Here are some guidelines on writing good commit messages:

- Each commit should ideally only address one issue. It should be self-contained (should not group together lots of changes). Tip: use git add -p to break your work down into logical, small commits).
- Write good commit messages. Read [this post](#) to see how to write meaningful, useful commit messages and why they are important.
- We strongly suggest using the [Conventional Commit](#) specification. In short:
 - Each commit is of the form <type>[optional scope]: description, followed by the text body of the commit after a blank line, and then any optional references etc. as footer.
 - The type can be one of: fix, feat, build, chore, ci, docs, refactor, perf, test, and so on depending on what the commit is doing.
 - Any backwards incompatible, breaking change must be clearly noted in the commit using the BREAKING CHANGE phrase. This corresponds to a major version update (as noted above in the versioning section).

30.1.6 Code style: Java

TODO

30.1.7 Code style: Python

- While Python 2 is still supported even though it is no longer supported by the Python community, given that most Python modules (numpy/scipy/matplotlib/sphinx) have dropped support for this deprecated Python version, NeuroML will also drop support in the near future. Therefore, we strongly suggest using Python 3.
- For Python repositories, please use [Black](#) to format your code before committing and submitting a pull request.
- We also strongly suggest linting using [flake8](#).
- Please use [type hints](#) in your code and run [mypy](#) to test it for correctness. You can see the [mypy cheatsheet](#) to quickly see how to do this. Since NeuroML is currently still supporting Python 2, we use the Python 2 style to maintain compatibility (this also works with Python 3).
- Deprecations should be clearly noted in the code, and in the commit message. You may use the [Sphinx deprecated](#) directive along with the Python [DeprecationWarning](#), for example.

30.1.8 Documentation

All tools include their own documentation in their repositories. Please feel free to improve this documentation and submit pull requests.

When contributing fixes and enhancements, please remember to document your classes/functions and code in general. Not only does this allow others to understand your code, it also allows us to auto-generate documentation using various tools.

- For the Java repositories, please use the standard [Javadoc](#) syntax.
- For the Python repositories, please document your code using the standard [Sphinx reStructuredText](#) system. For functions and so on, you can use the provided [fields](#).

Where applicable, please add examples and so on to the software documentation to ensure that users can find the information quickly. Additionally, please remember to consider if this primary NeuroML documentation here needs to be updated.

Please use [Semantic Line Breaks](#) wherever possible.

30.1.9 Testing

- Before submitting a pull request, please run the various tests to confirm your changes. You can see how they are run in the various GitHub workflow files (in the `.github/workflows/` folder in each repository). They will be run on all pull requests automatically so you can also verify your changes there.
- For a new feature addition, please remember to include a unit test.
- For a bug fix, please include a regression test.

30.2 Release Process

30.2.1 Overview

In general, work is carried out in the **development** branches of the [main NeuroML repositories](#) and these are merged to **master** branches on a new major release, e.g. move from NeuroML v2.1 to v2.2.

A single page showing the **status of the automated test** as well as any **open Pull Requests** on all of the core NeuroML repositories can be found [here](#).

30.2.2 Steps for new major release

These are the steps required for a new release of the NeuroML development tools.

Task	Version this was last done
Make releases (not just tag - generates DOI) previous development versions of individual repos	v2.2
Increment all version numbers - to distinguish release from previous development version	v2.2
Commit all work in development branches	v2.2
Test all development branches - rerun GitHub Actions at least once	v2.2
Recheck all READMEs & docs	v2.2
Run & check test.py in NeuroML2 repo	v2.2
Check through issues for closed & easily closable ones	v2.2
Update NeuroML milestones	v2.2
Update HISTORY.md in NeuroML2	v2.2
pylems: Update README; Merge to master; Tag release; Release to pip	v2.2
libNeuroML: Update README; Retest; Merge to master; Tag release; Release to pip; Check installation docs	v2.2
pyNeuroML: Update Readme; Tag release; Release to pip	v2.2
NeuroMLlite: Update Readme; Tag release; Release to pip	v2.2
Java repositories (jNeuroML , org.neuroml.* etc.): Merge development to master; Tag releases	v2.2
Rebuild jNeuroML & commit to jNeuroMLJar and use latest for jNeuroML for OMV	v2.2
Regenerate Cells.xml etc. on nml website & commit	v2.2
Update docs on http://www.neuroml.org	v2.2
Add new binary release on https://github.com/NeuroML/jNeuroML/releases	v2.2
ANNOUNCE (mailing list, Twitter)	v2.2
Increment version numbers in all development branches	v2.1
DOI on Zenodo	v2.1
New milestone in issues	v2.1
Update version used in neuroConstruct	v2.2
New release of neuroConstruct	v2.0
Test toolchain on Windows...	v2.0

30.3 Making changes to the NeuroML standard

The NeuroML standard is stored in two sets of files, each serving a specific purpose:

- the NeuroML [XML Schema Definition](#) (XSD) file: this specifies the structure of a valid NeuroML XML file: what XML tags may be used and the how they are related
- the NeuroML [LEMS](#) ComponentType definition XML files: these include the definitions of the NeuroML standard ComponentTypes in LEMS constructs, which include the mathematical details underlying these ComponentTypes

These files are housed in the [NeuroML](#) repository.

The XSD schema file is used to validate NeuroML XML files, as shown in the [page on validating NeuroML files](#). Further, the NeuroML Python model in [libNeuroML](#) is also generated from the XSD file using the `generateDS` utility.

The LEMS ComponentType definition XML files are also used for a series of additional validation tests, and since they include the details of the underlying dynamics for all ComponentTypes, they are also used for the simulation of NeuroML models either using the reference LEMS interpreter, [jLEMS](#), or through automated code generation for supported simulation platforms (via [jNeuroML](#)). Additionally, the LEMS definition files are also used to generate the [human readable schema documentation](#) included in this documentation resource.

The two sets of files are therefore, tightly coupled. Any changes to the XSD file must also be followed by corresponding changes to the LEMS definition files.

30.3.1 Procedure

PR waiting

TODO: A pull request to include the `transfer_docs_to_xsd.py` script in the repository is in review here: <https://github.com/NeuroML/NeuroML2/pull/172>

The suggested way of making changes to these files is via pull requests to the NeuroML repository which will undergo review by the NeuroML editorial board and the development team. As noted in the [general contribution guidelines](#), the development branch tracks the next release of the NeuroML standard. So, all pull request must be made against the development branch.

- New ComponentTypes, and their elements (parameters, variables etc.) that are added in the LEMS definition XML files should be properly documented.
- After both sets of files have been modified, please run the `transfer_docs_to_xsd.py` script in the `scripts` folder to copy documentation over from the XML files to the XSD schema file. This script will also run basic sanity checks to ensure that all ComponentTypes in the LEMS XML definition files are represented in the XSD schema file and vice-versa.
- Please run `xmllint` on the files to ensure they are formatted correctly.
- Please make individual commits for changes to the XSD file, and the XML files. This ensures that their change history is clearly maintained.

30.3.2 Regenerating schema documentation

Once the pull request has been merged in the NeuroML repository, the *human readable schema documentation included in this documentation resource* must be updated. This is done by running the `generate-jupyter-ast.py` script included in the documentation source repository. This will read the LEMS XML definition files and regenerate the corresponding documentation pages. A pull request can then be opened with the updated pages.

30.3.3 Updating the Java API: org.neuroml.model

TODO: Document what needs to be done for <https://github.com/NeuroML/org.neuroml.model>

30.3.4 Updating the Python API: libNeuroML

PR waiting

TODO: A pull request to include the `regenerate-nml.sh` script in the repository is in review here: <https://github.com/NeuralEnsemble/libNeuroML/pull/110>

Any changes to the XSD schema file require regeneration of the `Python` object model in `libNeuroML`:

- copy over the updated XSD schema file to the `neuroml/nml/` directory in the development branch
- commit the new XSD file
- run the `regenerate-nml.sh` script to regenerate and reformat `nml.py`
- build and install `libNeuroML` into a new virtual environment
- run all tests using `pytest`
- run all examples and ensure that they run correctly (please see the [GitHub actions workflow](#) for more information)
- if all checks pass successfully, a pull request can be opened

30.3.5 Updating the C++ API

TODO: Document what needs to be done for https://github.com/NeuroML/NeuroML_API/

INTERACTION WITH OTHER LANGUAGES AND STANDARDS

Needs work

TODO: Add more information to each of these

31.1 PyNN

<https://github.com/NeuroML/NeuroML2/issues/73>

31.2 SBML

<https://github.com/OpenSourceBrain/SBMLShowcase>

31.3 Sonata

<https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1007696>

31.4 NineML & SpineML

<https://github.com/OpenSourceBrain/NineMLShowcase>

31.5 ModECI MDF

<http://www.modeci.org/>

31.6 SWC

<http://www.neuronland.org/NLMorphologyConverter/MorphologyFormats/SWC/Spec.html> <http://www.neuromorpho.org/myfaq.jsp>

Part V

Reference

CHAPTER
THIRTYTWO

GLOSSARY

- XML: Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. (Read full entry on [Wikipedia](#))

CHAPTER
THIRTYTHREE

BIBLIOGRAPHY

BIBLIOGRAPHY

- [BRB+16] Marianne J Bezaire, Ivan Raikov, Kelly Burk, Dhrumil Vyas, and Ivan Soltesz. Interneuronal mechanisms of hippocampal theta oscillations in a full-scale model of the rodent ca1 circuit. *eLife*, 5:e18566, dec 2016. URL: <https://doi.org/10.7554/eLife.18566>, doi:10.7554/eLife.18566.
- [BBC+18] Inga Blundell, Romain Brette, Thomas A. Cleland, Thomas G. Close, Daniel Coca, Andrew P. Davison, Sandra Diaz-Pier, Carlos Fernandez Musoles, Padraig Gleeson, Dan F. M. Goodman, Michael Hines, Michael W. Hopkins, Pramod Kumbhar, David R. Lester, Bóris Marin, Abigail Morrison, Eric Müller, Thomas Nowotny, Alexander Peyser, Dimitri Plotnikov, Paul Richmond, Andrew Rowley, Bernhard Rumpf, Marcel Stimberg, Alan B. Stokes, Adam Tomkins, Guido Treisch, Marmaduke Woodman, and Jochen Martin Eppeler. Code generation in computational neuroscience: a review of tools and techniques. *Frontiers in Neuroinformatics*, 12:68, 2018. doi:10.3389/fninf.2018.00068.
- [CGC+14] Robert C. Cannon, Padraig Gleeson, Sharon Crook, Gautham Ganapathy, Boris Marin, Eugenio Piasini, and R. Angus Silver. LEMS: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2. *Frontiers in Neuroinformatics*, 2014. doi:10.3389/fninf.2014.00079.
- [CGH+07] Sharon Crook, Padraig Gleeson, Fred Howell, Joseph Svitak, and R. Angus Silver. MorphML: level 1 of the NeuroML standards for neuronal morphology data and model specification. *Neuroinformatics*, 5(2):96–104, 5 2007. doi:10.1007/s12021-007-0003-6.
- [FHA+15] KA Ferguson, CYL Huh, B Amilhon, S Williams, and FK Skinner. Data set of CA1 pyramidal cell recordings using an intact whole hippocampus preparation, including recordings of rebound firing (V2). May 2015. URL: <https://doi.org/10.5281/zenodo.17794>, doi:10.5281/zenodo.17794.
- [GKA+01] D. Gardner, K. H. Knuth, M. Abato, S. M. Erde, T. White, R. DeBellis, and E. P. Gardner. Common data model for neuroscience data and data model exchange. *Journal of the American Medical Informatics Association*, 8(1):17–33, 1 2001. doi:10.1136/jamia.2001.0080017.
- [GCM+19] Padraig Gleeson, Matteo Cantarelli, Boris Marin, Adrian Quintana, Matt Earnshaw, Sadra Sadeh, Eugenio Piasini, Justas Birgiolas, Robert C. Cannon, N. Alex Cayco-Gajic, Sharon Crook, Andrew P. Davison, Salvador Dura-Bernal, András Ecker, Michael L. Hines, Giovanni Idili, Frederic Lanore, Stephen D. Larson, William W. Lytton, Amitava Majumdar, Robert A. McDougal, Subhashini Sivagnanam, Sergio Solinas, Rokas Stanislovas, Sacha J. van Albada, Werner van Geit, and R. Angus Silver. Open source brain: a collaborative resource for visualizing, analyzing, simulating, and developing standardized models of neurons and circuits. *Neuron*, 103(3):395–411, 2019. doi:10.1016/j.neuron.2019.05.019.
- [GCC+10] Padraig Gleeson, Sharon Crook, Robert C. Cannon, Michael L. Hines, Guy O. Billings, Matteo Farinella, Thomas M. Morse, Andrew P. Davison, Subhasis Ray, Upinder S. Bhalla, Simon R. Barnes, Yoana D. Dimitrova, and R. Angus Silver. NeuroML: a language for describing data driven models of neurons and networks with a high degree of biological detail. *PLoS Computational Biology*, 6(6):e1000815, 2010. doi:10.1371/journal.pcbi.1000815.
- [GSS07] Padraig Gleeson, Volker Steuber, and R. Angus Silver. neuroConstruct: a tool for modeling networks of neurons in 3d space. *Neuron*, 54(2):219–235, 4 2007. doi:10.1016/j.neuron.2007.03.025.

- [GHH+01] Nigel H. Goddard, Michael Hucka, Fred Howell, Hugo Cornelis, Kavita Shankar, and David Beeman. Towards NeuroML: model description methods for collaborative modelling in neuroscience. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 356(1412):1209–1228, 8 2001. doi:10.1098/rstb.2001.0910.
- [HH52] Alan L. Hodgkin and Andrew F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500, 1952.
- [HCG+03] F. Howell, R. Cannon, N. Goddard, H. Bringmann, P. Rogister, and H. Cornelis. Linking computational neuroscience simulation tools—a pragmatic approach to component-based development. *Neurocomputing*, 52-54:289–294, 6 2003. doi:10.1016/s0925-2312(02)00781-6.
- [Izh07] Eugene M. Izhikevich. *Dynamical systems in neuroscience*. MIT Press, 2007.
- [Lor63] Edward N. Lorenz. Deterministic nonperiodic flow. *Journal of Atmospheric Sciences*, 20(2):130–141, 1963. doi:10.1175/1520-0469(1963)020<0130:dnf>2.0.co;2.
- [PBM04] Astrid A. Prinz, Dirk Bucher, and Eve Marder. Similar network activity from disparate circuit parameters. *Nature Neuroscience*, 7(12):1345–1352, 2004. doi:10.1038/nn1352.
- [QC04] Weihong Qi and Sharon Crook. Tools for neuroinformatic data exchange: an XML application for neuronal morphology data. *Neurocomputing*, 58-60:1091–1095, 6 2004. doi:10.1016/j.neucom.2004.01.171.
- [RGF+11] Cyrille Rossant, Dan F. Goodman, Bertrand Fontaine, Jonathan Platkiewicz, Anna Magnusson, and Romain Brette. Fitting neuron models to spike trains. *Frontiers in Neuroscience*, 5:9, 2011. URL: <https://www.frontiersin.org/article/10.3389/fnins.2011.00009>, doi:10.3389/fnins.2011.00009.
- [SKNG16] Arfon M. Smith, Daniel S. Katz, Kyle E. Niemeyer, and FORCE11 Software Citation Working Group. Software citation principles. *PeerJ Computer Science*, 2:e86, 2016. URL: <https://doi.org/10.7717/peerj-cs.86>, doi:10.7717/peerj-cs.86.
- [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.