NeuroML Documentation

NeuroML contributors

Jul 02, 2025

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 Downloading this documentation as PDF . . . 13
- 2.4 Reporting bugs and issues . . . 13

3 Get NeuroML **15**
- 3.1 What you need . . . 15

4 Getting started with NeuroML **17**
- 4.1 Simulating a regular spiking Izhikevich neuron . . . 18
- 4.2 Interactive single Izhikevich neuron NeuroML example . . . 33
- 4.3 A two population network of regular spiking Izhikevich neurons . . . 38
- 4.4 Interactive two population network example . . . 52
- 4.5 Simulating a single compartment Hodgkin-Huxley neuron . . . 66
- 4.6 Interactive single compartment HH example . . . 91
- 4.7 Simulating a multi compartment OLM neuron . . . 99
- 4.8 Interactive multi-compartment OLM cell example . . . 125
- 4.9 Create novel NeuroML models from components on NeuroML-DB . . . 139

5 Finding and sharing NeuroML models **169**
- 5.1 NeuroML-DB: The NeuroML Database . . . 169
- 5.2 Open Source Brain . . . 170
- 5.3 Other related projects . . . 170
- 5.4 NeuroMorpho.Org . . . 170
- 5.5 OpenWorm . . . 171
- 5.6 Allen Institute . . . 171
- 5.7 Blue Brain Project . . . 171

6 Creating NeuroML models **173**
- 6.1 Converting models to NeuroML and sharing them on Open Source Brain . . . 173
- 6.2 Handling Morphology Files . . . 179
- 6.3 HDF5 support . . . 183
- 6.4 Maintaining provenance in NeuroML models . . . 184

7 Validating NeuroML Models **187**
- 7.1 Using the command line tools . . . 187
- 7.2 Using the Python API . . . 187

7.3 List of validation tests . . . 188

8 Visualising NeuroML Models **191**
8.1 Get a quick summary of your model . . . 191
8.2 View the 3D structure of your model . . . 192
8.3 View the connectivity graph of your model . . . 196
8.4 View the connectivity matrices of the model . . . 198
8.5 View graph of the simulation instance of the model . . . 200
8.6 Viewing/analysing ion channel dynamics . . . 200
8.7 Visualising and analysing ion channel models . . . 200
8.8 Visualising and analysing cell models . . . 208

9 Simulating NeuroML Models **211**
9.1 Using Open Source Brain . . . 211
9.2 Using jNeuroML/pyNeuroML . . . 212
9.3 Using NEURON . . . 213
9.4 Using NetPyNE . . . 213
9.5 Using Brian2 . . . 214
9.6 Using MOOSE . . . 214
9.7 Using EDEN . . . 214
9.8 Using Arbor . . . 214

10 Optimising/fitting NeuroML Models **215**
10.1 Loading data and calculating metrics to use for optimisation . . . 224
10.2 Running the optimisation . . . 236
10.3 Viewing results . . . 246

11 Testing/validating NeuroML Models **255**
11.1 Testing behaviour of NeuroML models across simulators . . . 255
11.2 Validating that NeuroML model reproduce biological activity . . . 255

12 LEMS: Low Entropy Model Specification **257**
12.1 Capabilities . . . 257
12.2 Background . . . 258
12.3 Example . . . 258
12.4 Model structure overview . . . 259
12.5 Example 1: Dimensions, Units, ComponentTypes and Components . . . 263
12.6 Example 2: tidying up example 1 . . . 274
12.7 Main model . . . 274
12.8 Included files . . . 277
12.9 Example 3: Connection dependent synaptic components . . . 282
12.10 Example 4: Kinetic schemes . . . 283
12.11 Example 5: References and paths . . . 287
12.12 Example 6: User defined types for simulation and display . . . 292
12.13 Example 7: User defined types for networks and populations . . . 293
12.14 Example 8: Regimes in Dynamics definitions . . . 298

13 Schema/Specification **303**
13.1 NeuroML v2 . . . 304
13.2 NeuroML v1 . . . 668
13.3 LEMS . . . 668

14 NeuroML 2 and LEMS **723**
14.1 NeuroML 2 Component Type definitions in LEMS . . . 723
14.2 More information . . . 727

14.3 Conventions . . . 727
14.4 Units and dimensions . . . 729
14.5 Paths . . . 729
14.6 Quantities and recording . . . 733
14.7 LEMS Simulation files . . . 733

15 Extending NeuroML **737**
15.1 Creating new ComponentTypes with existing NeuroML ComponentTypes . . . 737
15.2 Creating new ComponentTypes with LEMS elements . . . 738
15.3 Examples . . . 743

16 Software and Tools **745**
16.1 Core NeuroML Tools . . . 745
16.2 Other NeuroML supporting applications . . . 746
16.3 pyNeuroML . . . 746
16.4 libNeuroML . . . 752
16.5 pyLEMS . . . 755
16.6 NeuroMLlite . . . 757
16.7 jNeuroML . . . 760
16.8 jLEMS . . . 763
16.9 NeuroML C++ API . . . 764
16.10 MatLab NeuroML Toolbox . . . 764
16.11 Tools and resources with NeuroML support . . . 765

17 Citing NeuroML and related publications **783**
17.1 Citing NeuroML . . . 783
17.2 Other publications . . . 786

18 Frequently asked questions (FAQ) **789**
18.1 1. Are length 0 segments allowed in NeuroML? . . . 789
18.2 2. What is the difference between reader/writer methods in pyNeuroML and libNeuroML? . . . 789

19 Walk throughs **791**
19.1 Converting Ray et al 2020 to NeuroML . . . 791

II NeuroML events **817**

20 NeuroML outreach and events **819**

21 July 2024: NeuroML tutorial at CNS 2024 **821**
21.1 Agenda . . . 821
21.2 Times and dates . . . 822
21.3 Registration . . . 822

22 April 2024: NeuroML hackathon at HARMONY 2024 **823**
22.1 Agenda . . . 823
22.2 Times and dates . . . 823
22.3 Registration . . . 823
22.4 Open an issue beforehand! . . . 824
22.5 Slack . . . 824

23 June 2022: NeuroML tutorial at CNS*2022 satellite tutorials **825**
23.1 Times and dates . . . 825
23.2 Target audience . . . 825

23.3 Where . . . 825
23.4 Agenda . . . 825

24 April 2022: NeuroML development workshop at HARMONY 2022 827
24.1 Agenda . . . 827
24.2 Times and dates . . . 827
24.3 Registration . . . 827
24.4 Open an issue beforehand! . . . 828
24.5 Slack . . . 828

25 October 2021: NeuroML development workshop at COMBINE meeting 829
25.1 Times and dates . . . 829
25.2 Target audience . . . 829
25.3 Agenda/minutes . . . 829

26 August 2021: NeuroML tutorial at INCF Training Weeks 831
26.1 Times and dates . . . 831
26.2 Target audience . . . 831
26.3 Agenda . . . 831

27 July 2021: NeuroML tutorial at CNS*2021 833
27.1 Why take part? . . . 833
27.2 Times and dates . . . 833
27.3 Registration . . . 833
27.4 Pre-requisites . . . 833
27.5 Slack . . . 834

28 March 2021: NeuroML hackathon at HARMONY 2021 835
28.1 Why take part? . . . 835
28.2 Times and dates . . . 835
28.3 Registration . . . 836
28.4 Open an issue beforehand! . . . 836
28.5 Slack . . . 836

29 March 2012: Fourth NeuroML Development Workshop 837
29.1 Meeting report . . . 837
29.2 Funding . . . 845

30 Past NeuroML Events 847

III The NeuroML Initiative 849

31 Getting in touch 851
31.1 Mailing list . . . 851
31.2 Chat channels . . . 851
31.3 Issues related to the libraries or specification . . . 851
31.4 Social media . . . 851

32 Overview of standards in neuroscience 853
32.1 NeuroML as a standard . . . 853

33 A brief history of NeuroML 855
33.1 The early days . . . 855
33.2 NeuroML v1.x . . . 855
33.3 NeuroML v2.x - introducing LEMS… . . . 856

33.4 The future . . . 856

34 NeuroML Editorial Board **857**
34.1 Current Editorial Board . . . 857
34.2 Procedures . . . 859
34.3 Responsibilities of NeuroML Editors . . . 859
34.4 History of the NeuroML Editorial Board . . . 859
34.5 Workshop and Meeting reports . . . 861

35 NeuroML Scientific Committee **863**
35.1 Current Members . . . 863
35.2 Past Members . . . 866

36 Funding and Acknowledgements **867**

37 Outreach and training **869**
37.1 Google summer of Code . . . 869

38 NeuroML contributors **871**

39 NeuroML repositories **877**

40 Code of Conduct **883**

IV Developer documentation **885**

41 Overview **887**
41.1 Contribution guidelines . . . 887
41.2 Release Process . . . 890
41.3 Making changes to the NeuroML standard . . . 891

42 Interaction with other languages and standards **893**
42.1 PyNN . . . 893
42.2 SBML . . . 893
42.3 Sonata . . . 893
42.4 NineML & SpineML . . . 893
42.5 ModECI MDF . . . 893
42.6 SWC . . . 894

V Reference **895**

43 Glossary **897**

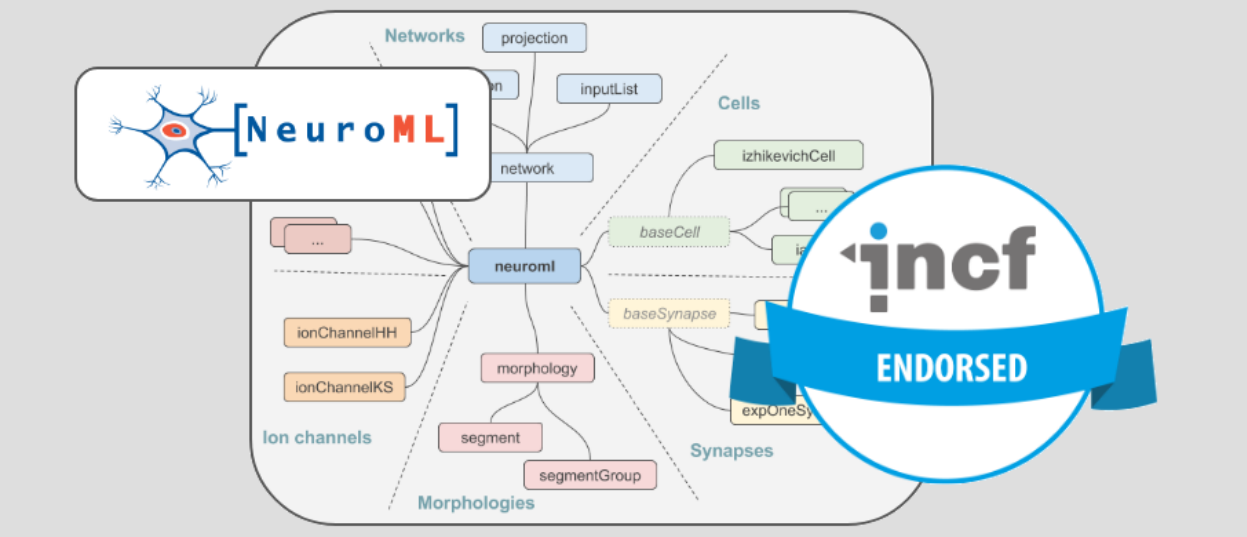
44 Bibliography **899**

Bibliography **901**

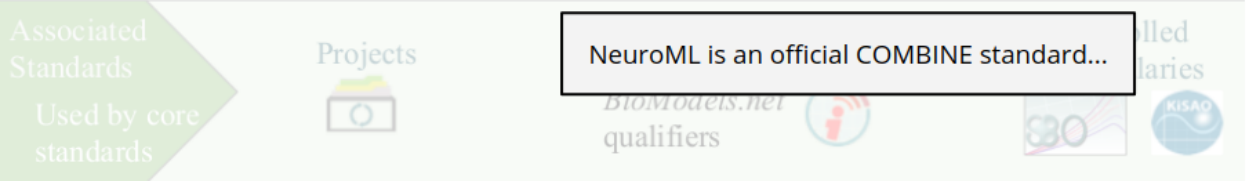
A model description language for computational neuroscience.

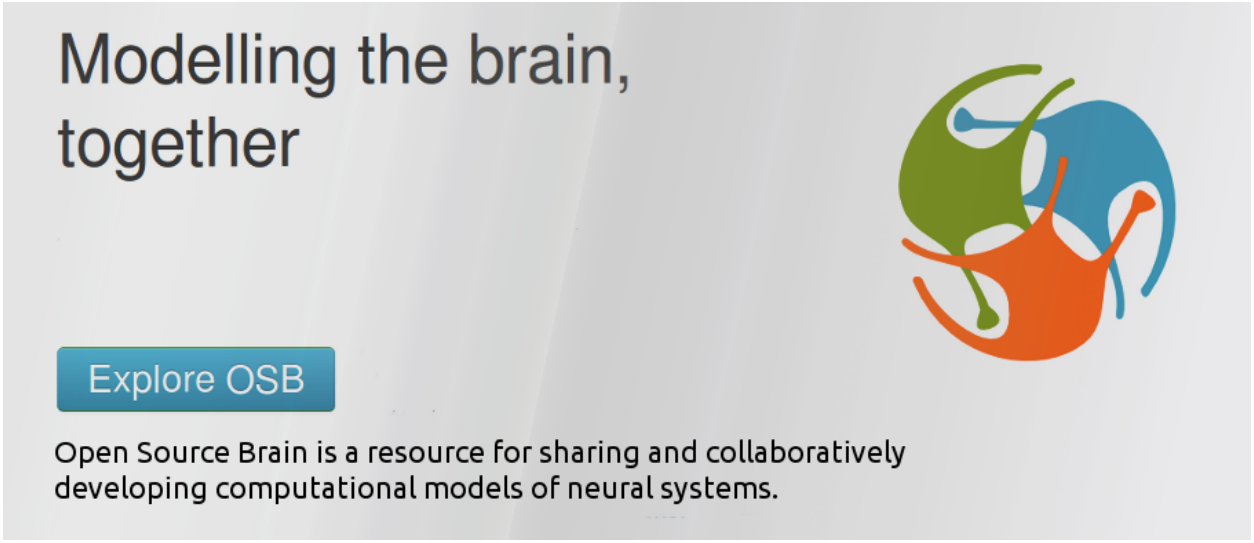
Read the latest NeuroML paper in eLife

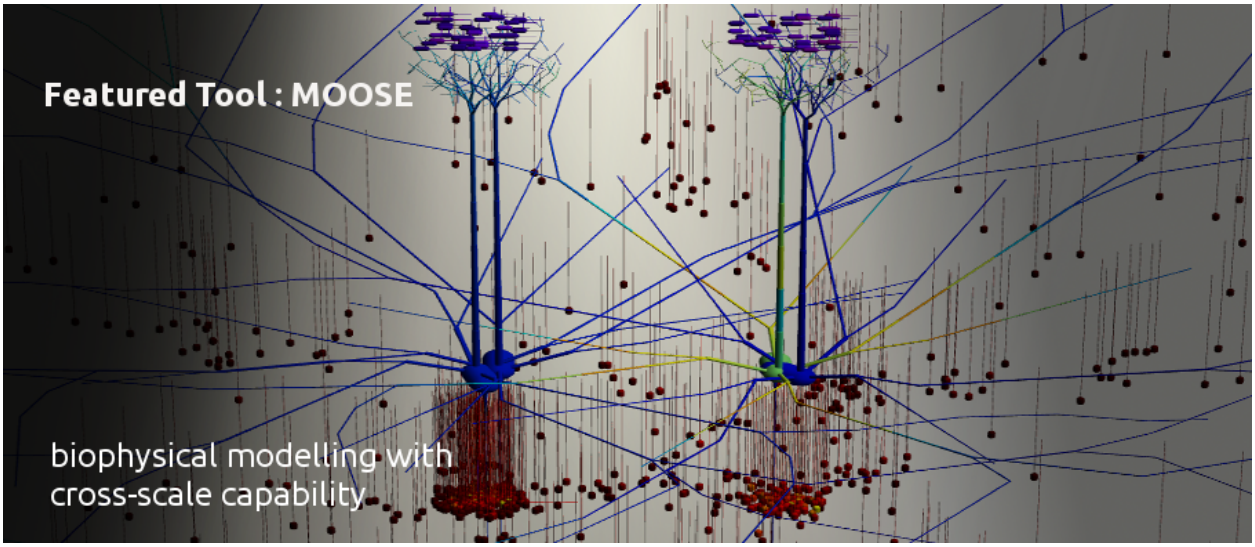
The NeuroML ecosystem for standardized multi-scale modeling in neuroscience ([SGM+25])

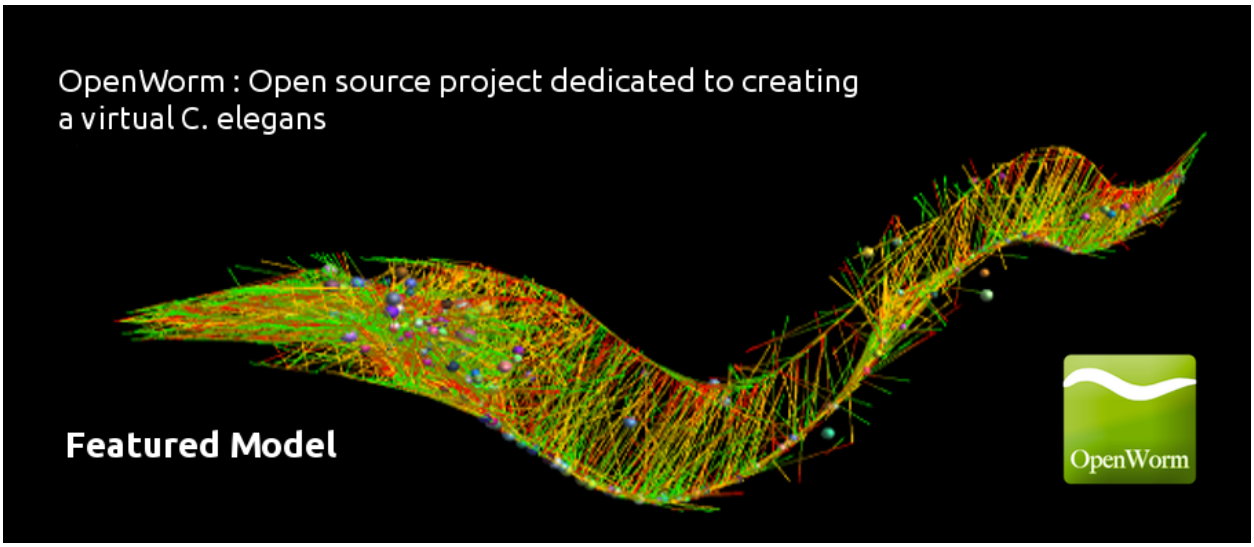












NeuroML is an *international, collaborative initiative* to develop a language for describing detailed models of neural sys-

tems, 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

CHAPTER

TWO

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 using the burger menu on the left side of the top panel.

You can close the left hand side bar by clicking the burger menu on the left side of the top panel. This increases the width of the middle section of the documentation and can be helpful on smaller screens. Clicking the hamburger menu again will re-open it.

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:

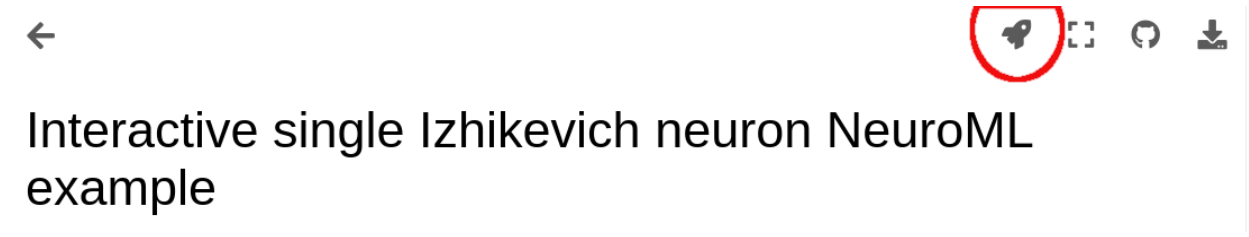


Fig. 2.1: Click the rocket icon in top panel of executable pages to execute them in Binder or Google Collaboratory.

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

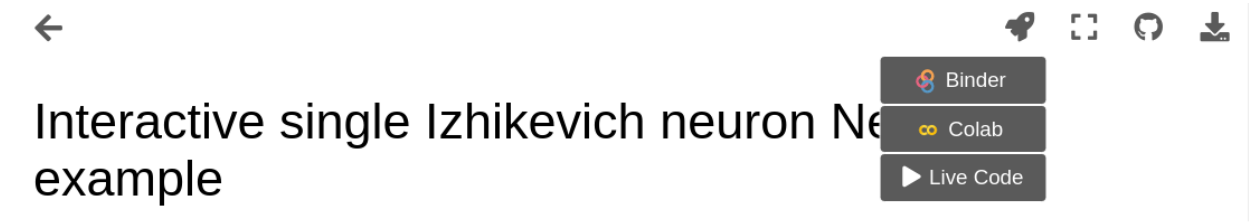


Fig. 2.2: You can run the Jupyter Notebook on Binder or Google Colaboratory.

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:

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

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.

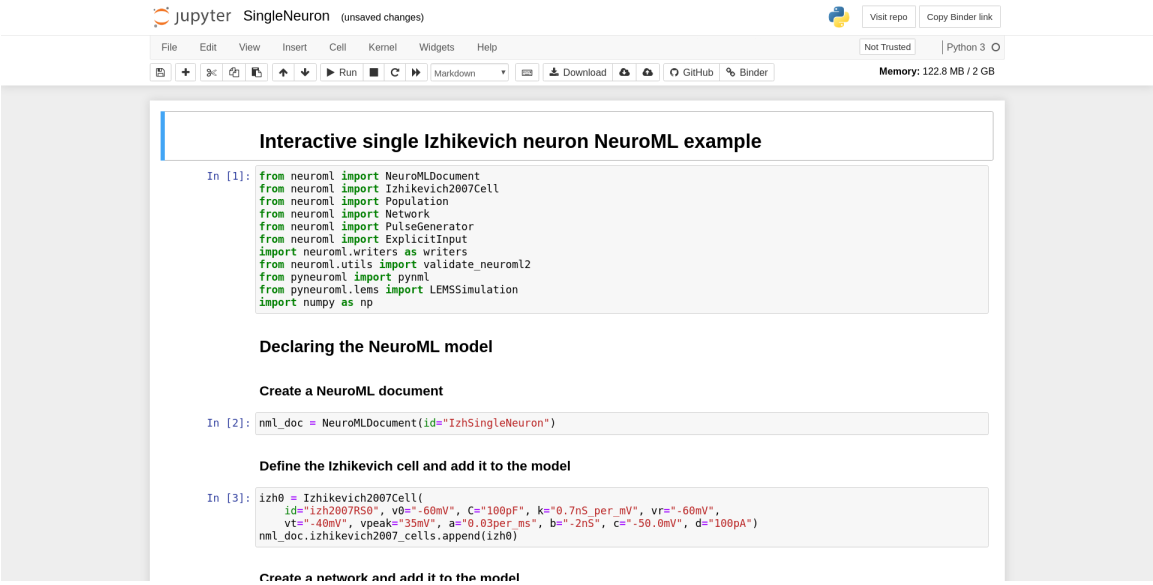


Fig. 2.3: Izhikevich example running in Binder

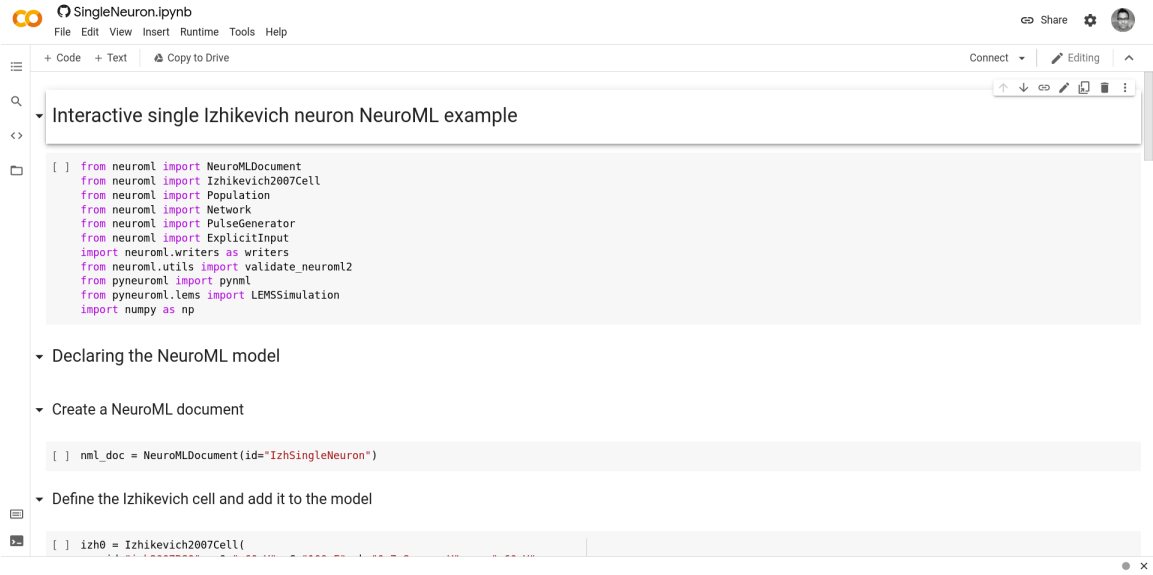


Fig. 2.4: Izhikevich example running in Google Colaboratory.

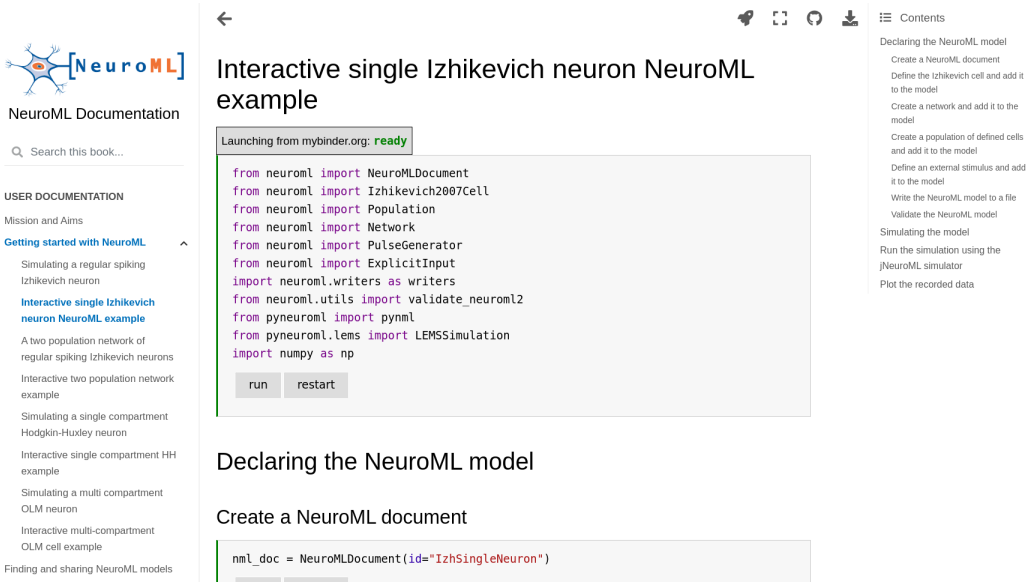


Fig. 2.5: Izhikevich example running in Binder but using the Live Code option.

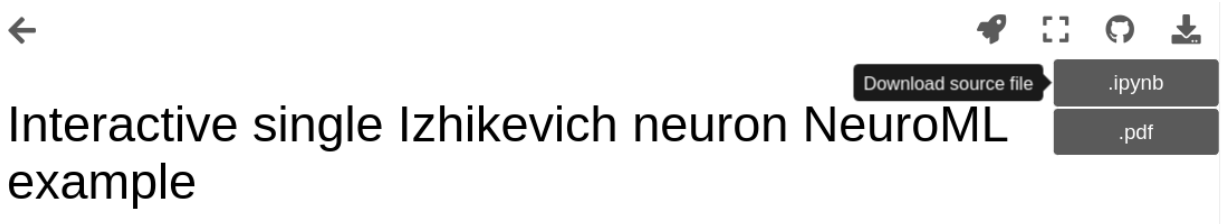


Fig. 2.6: Jupyter notebooks can be downloaded using the Download link in the top panel.

2.3 Downloading this documentation as PDF

You can download this documentation as PDF pages for offline use.

To download individual pages, use the download icon in the top bar. This will generate a PDF page of the current page for you, using your browser's "print to file" functionality.

You can also download the complete book as a PDF here.

2.4 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.

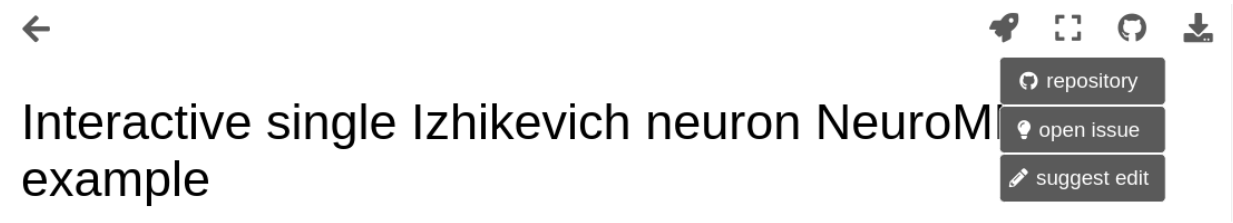


Fig. 2.7: You can report issues and suggest edits to the documentation to help us improve it using the options in the GitHub icon in the top panel.

You can also always contact the NeuroML community using our *communication channels* if required.

CHAPTER

THREE

GET NEUROML

While one can use Jupyter Notebooks on different platforms (Binder/Open Source Brain v2/Google Colab) to work with NeuroML models (for example, the tutorials in this documentation can mostly be run on Jupyter Notebooks), for certain use cases, it may be preferable/necessary to install the stack on our own computers. One such use case, for example, is when one needs to run large scale simulations that require supercomputers/clusters to simulate.

3.1 What you need

The NeuroML stack is written primarily in Python and Java, and so requires:

- a supported, working Python installation
- a working Java Runtime Environment (JRE)

The software stack is currently tested on:

- Python versions: 3.9–3.13 (3.12 is preferred)
- Java versions 8, 11, 16, 17, 19 on these operating systems (OS): Ubuntu 22.04 ("ubuntu-latest"), MacOS 14 Arm 64 ("macos-latest"), Windows 2019 ("windows-2019")

Once you have these programming languages installed, all you need to do is install *pyNeuroML*, and that will install the other parts of the NeuroML software stack for you too. Please see the *pyNeuroML* page for more details.

CHAPTER

FOUR

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. We present below a set of step-by-step guides to illustrate how models are written and simulated using NeuroML.

Link to guide	Description	Model life cycle stages
	Introductory guides	
Guide 1	Create and simulate a simple regular spiking Izhikevich neuron in NeuroML	Create, Validate, Simulate
Guide 2	Create a network of two synaptically connected populations of Izhikevich neurons	Create, Validate, Visualise, Simulate
Guide 3	Build and simulate a single compartment Hodgkin-Huxley neuron	Create, Validate, Visualise, Simulate
Guide 4	Create and simulate a multi compartment hippocampal OLM neuron	Create, Validate, Visualise, Simulate
	Advanced guides	
Guide 5	Create novel NeuroML models from components on NeuroML-DB	Reuse, Create, Validate, Simulate
Guide 6	Optimise/fit NeuroML models to experimental data	Create, Validate, Simulate, Fit
Guide 7	Extend NeuroML by creating a novel model type in LEMS	Create, Simulate
	Step by step walkthroughs	
Guide 8	Guide to converting cell models to NeuroML and sharing them on Open Source Brain	Create, Validate, Simulate, Share
Guide 9	Conversion of Ray et al 2020 [RAS20] to NeuroML	Create, Validate, Visualise, Simulate, Extend using LEMS

You do not need to install any software on your computers to run many of the examples above. These examples are followed by a Jupyter notebook for you to experiment with inside your browser (*more info*).

4.1 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 Juptyer 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):

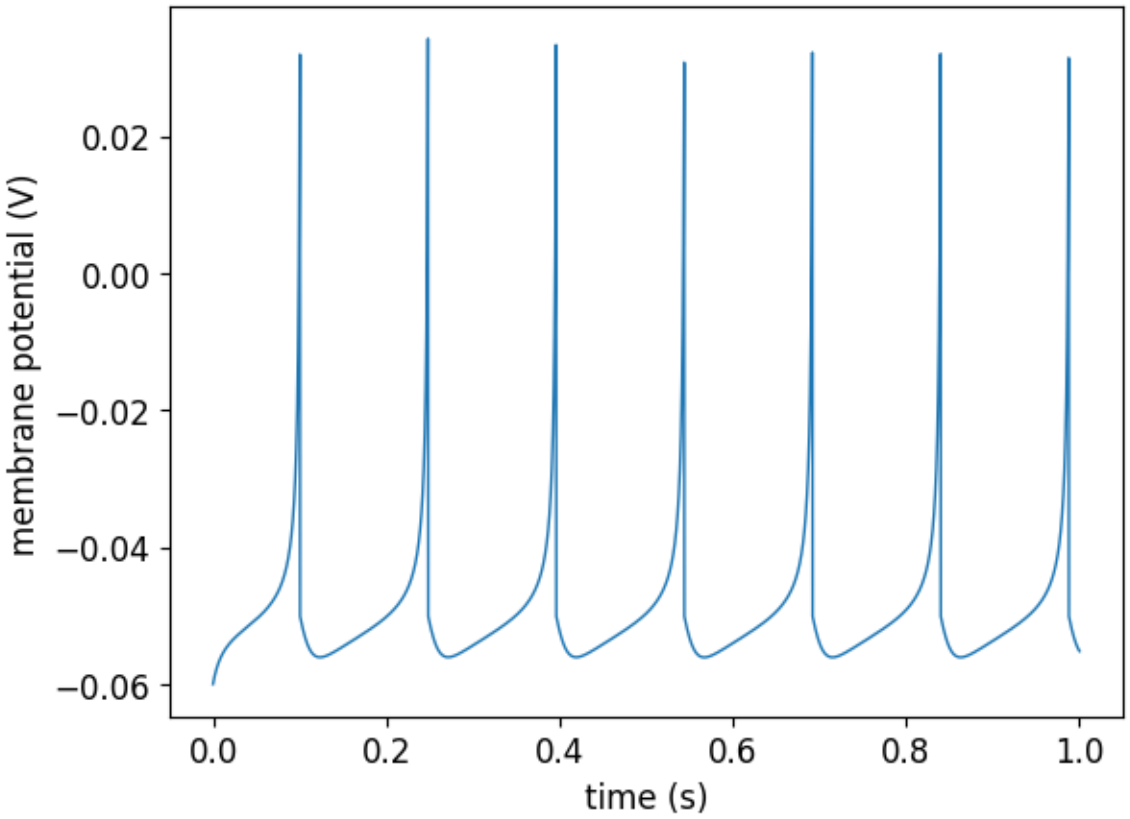


Fig. 4.1: Membrane potential of the simulated regular spiking Izhikevich neuron.

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
import neuroml.writers as writers
from neuroml.utils import component_factory
```

(continues on next page)

(continued from previous page)

```
from neuroml.utils import validate_neuroml2
from pyneuroml import pynml
from pyneuroml.lems import LEMSSimulation
import numpy as np


# Create a new NeuroML model document
# component_factory: form one: provide name of NeuroML class as string
# advantage of this form: do not need to import all the ComponentType classes
# before using them
nml_doc = component_factory("NeuroMLDocument", id="IzhSingleNeuron")
# component_factory: form two: provide class as argument
# nml_doc = component_factory(NeuroMLDocument, id="IzhSingleNeuron")

# Inspect it:
nml_doc.info()

# Also see contents:
nml_doc.info(show_contents=True)

# Define the Izhikevich cell and add it to the model in the document
# the `add` will create and validate the new component, and add it to the
# parent (nml_doc)
izh0 = nml_doc.add(
    "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")

# Exercise 1: give wrong units of a parameter/parameters
# Exercise 2: skip out a few parameters

# Inspect the component
izh0.info()

# Inspect the component, also show all members:
izh0.info(True)

# inspect the document
nml_doc.info(show_contents=True)

# Create a network and add it to the model
# net = component_factory("Network", id="IzNet")
# Throws an error: why?
# Because a Population is necessary in a Network, but we have not provided one.
# Two workarounds:
# - create population first, and pass that to component_factory here
# - disable validation
net = nml_doc.add("Network", id="IzNet", validate=False)

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

# Define an external stimulus and add it to the model
pg = nml_doc.add(
    "PulseGenerator",
    id="pulseGen_%i" % 0, delay="0ms", duration="1000ms",
```

(continues on next page)

(continued from previous page)

```
    amplitude="0.07 nA"
)
exp_input = net.add("ExplicitInput", target="%s[%i]" % (pop0.id, 0), input=pg.id)

# 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 an instance of 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
)
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)"
)
```

4.1.1 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" represents the complete model and is the top level container for everything that the model should contain.

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

Let us define an Izhikevich cell that we will use to simulate a neuron. The Izhikevich neuron model can take sets of parameters to exhibit different types of spiking behaviour. Here, we define a component (object) of the general Izhikevich cell using parameters to show regular spiking.

```
izh0 = nml_doc.add(
    "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")
```

Now that the neuron has been defined and added to the document, we declare a *network* with a *population* of these neurons to create a network in a similar way. 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 add them to our model:

```
net = nml_doc.add("Network", id="IzNet", validate=False)

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

Question: why did we disable validation when we created the new network component?

```
net = nml_doc.add("Network", id="IzNet", validate=False)
```

Let us try creating a network without disabling validation:

```
net = nml_doc.add("Network", id="IzNet")
```

It will throw a validation error:

```
ValueError: Validation failed:
- Number of values for populations is below the minimum allowed, expected at least 1,␣
 ↪found 0
```

This is because a network must have at least one population for it to be valid. To fix this, we can either create the population before the network, or we can disable validation. Here we chose to disable validation because we knew we were immediately creating our population and adding it to our network.

Moving on, since we are providing a single input to the single cell in our network, we can add an `ExplicitInput` to our network. See the supplementary section on the `info` function below to learn how you can find out that `ExplicitInput` could be used here.

The list of inputs included in the NeuroML specification can be found on the *inputs* page. We use a *pulse generator* here, creating a new component and adding it to our NeuroML document. To connect it to our neuron, we specify the neuron as the `target` using an *explicit input*.

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

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 again to check if it conforms to the NeuroML schema (more on this *later*).

```
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)
```

Note that the validation here will re-run the tests our component factory and other methods use, but it also runs a series of additional tests that can only be run on the complete model. So, it is necessary to validate the model after it has been fully constructed.

4.1.2 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.

```
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')

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

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

```
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*.

4.1.3 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.

```
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)"
)
```

On the next page, you will find 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.*

4.1.4 Supplementary information

The sections here explain concepts that have been used above. These will help give you a deeper understanding of NeuroML, so we do suggest you go through them also.

The generated NeuroML model XML

Let us investigate the generated NeuroML XML file:

```
<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="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="IzNet">
        <population id="IzhPop0" component="izh2007RS0" size="1"/>
        <explicitInput target="IzhPop0[0]" input="pulseGen_0"/>
    </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 in 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: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="IzhSingleNeuron
↪">
```

The first attribute, `xmlns` 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="IzNet">
        <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 as we saw before (see *here* for the schema definition):

- `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.

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.

The schema

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:

```
    <xs:complexType name="Izhikevich2007Cell">
        <xs:complexContent>
            <xs:extension base="BaseCellMembPotCap">
                <xs:attribute name="v0" type="Nml2Quantity_voltage" use="required"/>
                <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>
```

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.

The generated LEMS XML

The generated LEMS simulation file is shown below:

```
<Lems>

    <!--

        This LEMS file has been automatically generated using PyNeuroML v1.1.13␣
↪(libNeuroML v0.5.8)

     -->

    <!-- 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"/>
```

(continues on next page)

(continued from previous page)

```
    <Simulation id="example-single-izhikevich2007cell-sim" length="1000ms" step="0.1ms
↪" target="IzNet" 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.

The component_factory() function

In the code above, we've used the `component_factory` utility function that is included in the `neuroml.utils` module. This is, as the name notes, a "factory function". When we provide the name of a NeuroML component type (the Python class) to it as the first argument along with any parameters, it will create a new component (Python object) and return it to us to use, after running a few checks under the hood:

- is the created component valid?
- are all the necessary parameters set?
- are any extra parameters given?

We will see some of these checks in action later as we create more components for our model.

The `component_factory` can accept two forms. We can either pass the component type (class) to the function, or we can pass its name as a string. The difference is that we do not need to `import` the class in our script before using it if we specify its name as a string. The component factory function will import the class for us for us internally. Either form works, so you can choose which you prefer. It is important to only remain consistent and use one form to aid readability.

The add() function

We've used another utility method in the code above: `add`. The `add` method calls the `component_factory` for us internally to create a new object of the required component.

We could also use the `component_factory`, followed by `add`, which would result in the same thing:

```
izh0 = component_factory(
    "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.add(izh0)
```

In fact, we could do it all without using either method:

```
# from neuroml import Izhikevich2007Cell
izh0 = neuroml.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)
```

This last form is not suggested because here, the extra checks that the `component_factory` and `add` methods run are not carried out. You also need to know the name of the variable in the `nml_doc` object to be able to append to it. The output of the `info` method will list all the member names, but the `add` method inspects the parent component and places the child in the right place for us.

An exercise here would be to try providing invalid arguments to the `add` or `component_factory` methods. For example:

- try giving the wrong units for a parameter
- try leaving out a parameter

What happens?

For example, I have used the wrong units for the `d` parameter here, `ms` instead of `pA`:

```
# or
# izh0 = component_factory(
izh0 = nml_doc.add(
    "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="100ms")
```

and it will throw a `ValueError` telling us that this does not match the expected string for `d`:

```
ValueError: Validation failed:
- Value "100ms" does not match xsd pattern restrictions: [['^(-?([0-9]*(\\.[0-9]+)?
 ↪)([eE]-?[0-9]+)?[\\s]*(A|uA|nA|pA))$']]
```

The specific error here includes the "pattern restrictions" (regular expression) for valid values of the `d` parameter. There are a number of tutorials on regular expressions on the internet that you can use to learn more about the meaning of the provided pattern restriction. The one restriction that we are interested in here is that the value of `d` must end in one of `A`, `uA`, `nA`, or `pA`. Anything else will result in an invalid value, and the factory will throw a `ValueError`.

The NeuroML specification declares valid units for all its components. This allows us to validate models and components while building the model—even before we have a complete model that we want to simulate. In fact, NeuroML also defines a list of units and dimensions that can be used.

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*.

The info() function

Now that we have a document, what if we want to inspect it to see what components it can hold, and what its current contents are? Each NeuroML component type includes the `info` function that gives us a quick summary of information about the component:

```
nml_doc.info()

# Also see contents:
nml_doc.info(show_contents=True)
```

The output will be of this form:

```
Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
 ↪Userdocs/NeuroMLv2.html for more information.

Valid members for NeuroMLDocument are:
* poisson_firing_synapses (class: PoissonFiringSynapse, Optional)
* fixed_factor_concentration_models (class: FixedFactorConcentrationModel, Optional)
* transient_poisson_firing_synapses (class: TransientPoissonFiringSynapse, Optional)
* alpha_current_synapses (class: AlphaCurrentSynapse, Optional)
* IF_curr_alpha (class: IF_curr_alpha, Optional)
* alpha_synapses (class: AlphaSynapse, Optional)
...
```

This shows all the valid NeuroML components that the top level `NeuroMLDocument` component can directly contain. It also tells us the component type (class) corresponding to the component (object). It also tells us whether this component is optional or required.

In the second form, where we also pass `show_contents=True`, it will also show the contents of each member if any. We can use this to inspect our created Izhikevich cell component:

```
izh0.info(True)
```

The output will be:

```
Izhikevich2007Cell -- Cell based on the modified Izhikevich model in Izhikevich 2007,␣
 ↪Dynamical systems in neuroscience, MIT Press

Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
 ↪Userdocs/NeuroMLv2.html for more information.

Valid members for Izhikevich2007Cell are:
* annotation (class: Annotation, Optional)
* b (class: Nml2Quantity_conductance, Required)
        * Contents ('ids'/<objects>): -2nS

* c (class: Nml2Quantity_voltage, Required)
        * Contents ('ids'/<objects>): -50.0mV

* d (class: Nml2Quantity_current, Required)
        * Contents ('ids'/<objects>): 100pA

* C (class: Nml2Quantity_capacitance, Required)
        * Contents ('ids'/<objects>): 100pF

* v0 (class: Nml2Quantity_voltage, Required)
        * Contents ('ids'/<objects>): -60mV
```

(continues on next page)

(continued from previous page)

```
* k (class: Nml2Quantity_conductancePerVoltage, Required)
        * Contents ('ids'/<objects>): 0.7nS_per_mV

* vr (class: Nml2Quantity_voltage, Required)
        * Contents ('ids'/<objects>): -60mV

* neuro_lex_id (class: NeuroLexId, Optional)
* metaid (class: MetaId, Optional)
* vt (class: Nml2Quantity_voltage, Required)
        * Contents ('ids'/<objects>): -40mV

* id (class: NmlId, Required)
        * Contents ('ids'/<objects>): izh2007RS0

* notes (class: xs:string, Optional)
* vpeak (class: Nml2Quantity_voltage, Required)
        * Contents ('ids'/<objects>): 35mV

* properties (class: Property, Optional)
* a (class: Nml2Quantity_pertime, Required)
        * Contents ('ids'/<objects>): 0.03per_ms
```

We can see that all the required parameters are correctly set for this component.

We can also inspect the full document:

```
nml_doc.info(show_contents=True)
```

Try running this at the beginning of the script right after creating the document, and at the end when the model has been completed. You should notice a major change, that our cell has been correctly added to the document.

```
...
* izhikevich2007_cells (class: Izhikevich2007Cell, Optional)
*         * Contents ('ids'/<objects>): ['izh2007RS0']
*
...
```

The `info()` function is very useful to see what components can belong to another. For example, to see what components can be added to our `net` network, we can run this:

```
net.info()

Network -- Network containing:  **population** s ( potentially of type␣
 ↪**populationList** , and so specifying a list of cell  **location** s ); ␣
 ↪**projection** s ( with lists of  **connection** s ) and/or  **explicitConnection**␣
 ↪s; and  **inputList** s ( with lists of  **input** s ) and/or  **explicitInput** s.␣
 ↪Note: often in NeuroML this will be of type  **networkWithTemperature**  if there␣
 ↪are temperature dependent elements ( e. g. ion channels ).

Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
 ↪Userdocs/NeuroMLv2.html for more information.

Valid members for Network are:
* metaid (class: MetaId, Optional)
* notes (class: xs:string, Optional)
* properties (class: Property, Optional)
```

(continues on next page)

(continued from previous page)

```
* annotation (class: Annotation, Optional)
* type (class: networkTypes, Optional)
* temperature (class: Nml2Quantity_temperature, Optional)
* neuro_lex_id (class: NeuroLexId, Optional)
* spaces (class: Space, Optional)
* regions (class: Region, Optional)
* extracellular_properties (class: ExtracellularPropertiesLocal, Optional)
* populations (class: Population, Required)
* cell_sets (class: CellSet, Optional)
* id (class: NmlId, Required)
* synaptic_connections (class: SynapticConnection, Optional)
* projections (class: Projection, Optional)
* electrical_projections (class: ElectricalProjection, Optional)
* continuous_projections (class: ContinuousProjection, Optional)
* explicit_inputs (class: ExplicitInput, Optional)
* input_lists (class: InputList, Optional)
```

This tells us what `net` can contain. For setting the input, for example, it would seem that we should use one of either `ExplicitInput` or `InputList` here. The `ctinfo` function can be used to get more information about these (next).

The ctinfo() function

There are multiple ways of getting information on a component type. The first, of course, is to look at the *schema* documentation online. The documentation for ExplicitInput is *here*, and for InputList is *here*. The schema documentation will also include examples of usage for most component types under the "Usage:Python" tab.

`neuroml` includes the `ctinfo()` utility function, that like the `info()` method, provides information about component types (`ct` in `ctinfo` stands for `component type`). Note that component types are classes and the `info()` method cannot be used on them. It can only be used once objects have been created from the component type classes.

So, we could do (create a new dummy object of the class and call `info()` on it):

```
neuroml.ExplicitInput().info()
```

but `ctinfo` will do this for us:

```
from neuroml.utils import ctinfo
ctinfo("ExplicitInput")
# or the second form:
# ctinfo(neuroml.ExplicitInput)
ExplicitInput -- An explicit input ( anything which extends  **basePointCurrent**  )␣
 ↪to a target cell in a population

Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
 ↪Userdocs/NeuroMLv2.html for more information.

Valid members for ExplicitInput are:
* destination (class: xs:string, Optional)
* target (class: xs:string, Required)
* input (class: xs:string, Required)


ctinfo("InputList")
InputList -- An explicit list of  **input** s to a **population.**

Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
```

(continues on next page)

(continued from previous page)

```
 ↪Userdocs/NeuroMLv2.html for more information.

Valid members for InputList are:
* populations (class: NmlId, Required)
* component (class: NmlId, Required)
* input (class: Input, Optional)
* input_ws (class: InputW, Optional)
* id (class: NmlId, Required)
```

Finally, for completeness, we can also get information from the API documentation for libNeuroML here. Since this is documentation that is "embedded" in the Python classes, we can also use the Python in-built help function to see it:

```
help(neuroml.ExplicitInput)
Help on class ExplicitInput in module neuroml.nml.nml:

class ExplicitInput(BaseWithoutId)
 |  ExplicitInput(target: 'one str (required)' = None, input: 'one str (required)' =␣
 ↪None, destination: 'one str (optional)' = None, gds_collector_=None, **kwargs_)
 |
 |  ExplicitInput -- An explicit input ( anything which extends  **basePointCurrent**␣
 ↪ ) to a target cell in a population
 |
 ...



help(neuroml.InputList)
Help on class InputList in module neuroml.nml.nml:

class InputList(Base)
 |  InputList(id: 'one NonNegativeInteger (required)' = None, populations: 'one NmlId␣
 ↪(required)' = None, component: 'one NmlId (required)' = None, input: 'list of␣
 ↪Input(s) (optional)' = None, input_ws: 'list of InputW(s) (optional)' = None, gds_
 ↪collector_=None, **kwargs_)
 |
 |  InputList -- An explicit list of  **input** s to a **population.**
 ...
```

The information provided by the different sources will be similar, but `ctinfo()` is perhaps the most NeuroML specific (whereas the Python `help()` function provides Python language related information also.)

Use an integrated development environment (IDE)

IDEs make programming easier. For example, a good IDE will show you the documentation that the `help` Python function shows.

Another useful function is the `ctparentinfo()` function. Like `info()` it provides some information about the component/object:

```
ctparentinfo("InputList")
InputList -- An explicit list of  **input** s to a **population.**

Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
 ↪Userdocs/NeuroMLv2.html for more information.
```

(continues on next page)

(continued from previous page)

```
Valid parents for InputList are:
* Network
        * input_lists (class: InputList, Optional)
```

This tells us that components of type `InputList` can be added to components of the `Network` type, in the `input_list` member. Of course, we will use the `add` function in our network object `net`, and that will add the component to the correct member.

The validate() function

We can check whether each component is valid using the `validate` function that each component has. For example:

```
net.validate()
```

This function does not return anything if the component is valid. (Technically, if a function does not return anything in Python, it returns `None` by default, so this returns `None` if the component is valid.) However, if it is not valid, it will throw a `ValueError`.

4.2 Interactive single Izhikevich neuron NeuroML example

To run this interactive Jupyter Notebook when viewing the online NeuroML documentation (e.g. via Binder or Google Colab), please click on the rocket icon in the top panel. For more information, please see *how to use this documentation*.

This notebook creates a simple model in NeuroML version 2. It adds a simple spiking neuron model to a population and then the population to a network. Then we create a LEMS simulation file to specify how to to a simulate the model, and finally we execute it using jNeuroML. The results of that simulation are plotted below.

See also a *more detailed introduction to NeuroML and LEMS using this example*.

4.2.1 1) Initial setup and library installs

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
import neuroml.writers as writers
from neuroml.utils import component_factory
from neuroml.utils import validate_neuroml2
from pyneuroml import pynml
from pyneuroml.lems import LEMSSimulation
import numpy as np
```

4.2.2 2) Declaring the NeuroML model

Create a NeuroML document

This is the container document to which the cells and the network will be added.

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

Define the Izhikevich cell and add it to the model

The Izhikevich model is a simple, 2 variable neuron model exhibiting a range of neurophysiologically realistic spiking behaviours depending on the parameters given. We use the izhikevich2007cell version here.

```
izh0 = nml_doc.add(
    "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")
```

Create a network and add it to the model

We add a network to the document created above.

```
net = nml_doc.add("Network", id="IzNet", validate=False)
```

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

A population of size 1 of these cells is created and then added to the network.

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

Define an external stimulus and add it to the model

On its own the cell will not spike, so we add a small current to it in the form of a pulse generator which will apply a square pulse of current.

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

4.2.3 3) Write, print and validate the generated file

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
```

Print out the file

Here we print the XML for the saved NeuroML file.

```
with open(nml_file) as f:
   print(f.read())
```

```
<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=
↪"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="IzNet">
        <population id="IzhPop0" component="izh2007RS0" size="1"/>
        <explicitInput target="IzhPop0[0]" input="pulseGen_0"/>
    </network>
</neuroml>
```

Validate the NeuroML model

```
validate_neuroml2(nml_file)
```

```
It's valid!
```

4.2.4 4) Simulating the model

Create a simulation instance of the model

The NeuroML file does not contain any information on how long to simulate the model for or what to save etc. For this we will need a LEMS simulation file.

```
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)
```

```
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
↪source/Userdocs/NML2_examples/izhikevich2007_single_cell_network.nml
```

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()
with open(lems_simulation_file) as f:
   print(f.read())
```

```
<Lems>

    <!--

        This LEMS file has been automatically generated using PyNeuroML v1.1.10␣
↪(libNeuroML v0.5.8)

     -->

    <!-- 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="IzNet" 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>
```

4.2.5 Run the simulation using the jNeuroML simulator

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

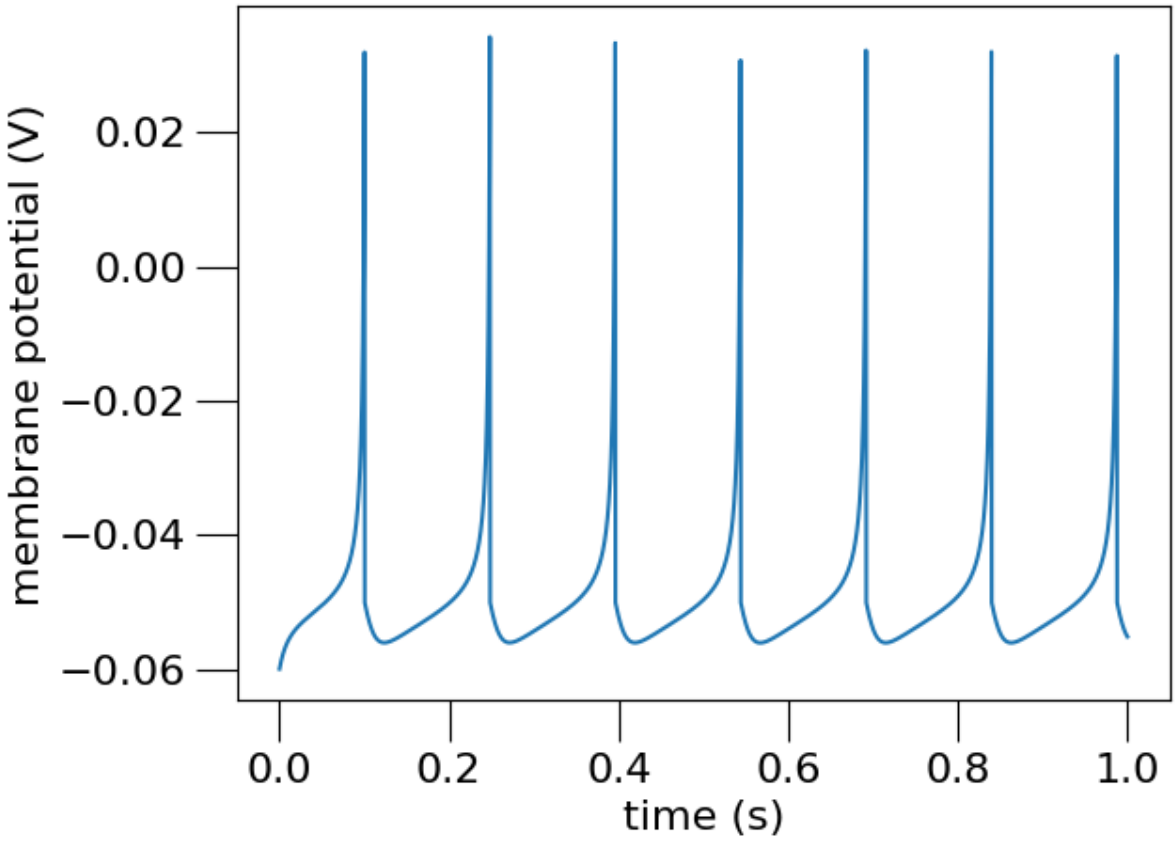
```
pyNeuroML >>> INFO - Loading LEMS file: LEMS_example-single-izhikevich2007cell-sim.
 ↪xml and running with jNeuroML
pyNeuroML >>> INFO - Executing: (java -Xmx2G  -Djava.awt.headless=true -jar  "/opt/
 ↪homebrew/anaconda3/envs/py39n/lib/python3.9/site-packages/pyneuroml/lib/jNeuroML-
 ↪0.13.0-jar-with-dependencies.jar"  LEMS_example-single-izhikevich2007cell-sim.
 ↪xml  -nogui -I '') in directory: .
pyNeuroML >>> INFO - Command completed successfully!
```

```
True
```

4.2.6 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)",
    save_figure_to="SingleNeuron.png"

)
```

```
pyNeuroML >>> INFO - Generating plot: Membrane potential
pyNeuroML >>> INFO - Saving image to /Users/padraig/git/Documentation/source/
 ↪Userdocs/NML2_examples/SingleNeuron.png of plot: Membrane potential
pyNeuroML >>> INFO - Saved image to SingleNeuron.png of plot: Membrane potential
```




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

4.3 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. Please note that this example uses the *NEURON* simulator to simulate the model. Please ensure that the `NEURON_HOME` environment variable is correctly set as noted *here*.

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

import random
import numpy as np

from neuroml.utils import component_factory
from pyneuroml import pynml
from pyneuroml.lems import LEMSSimulation
import neuroml.writers as writers
```

(continues on next page)

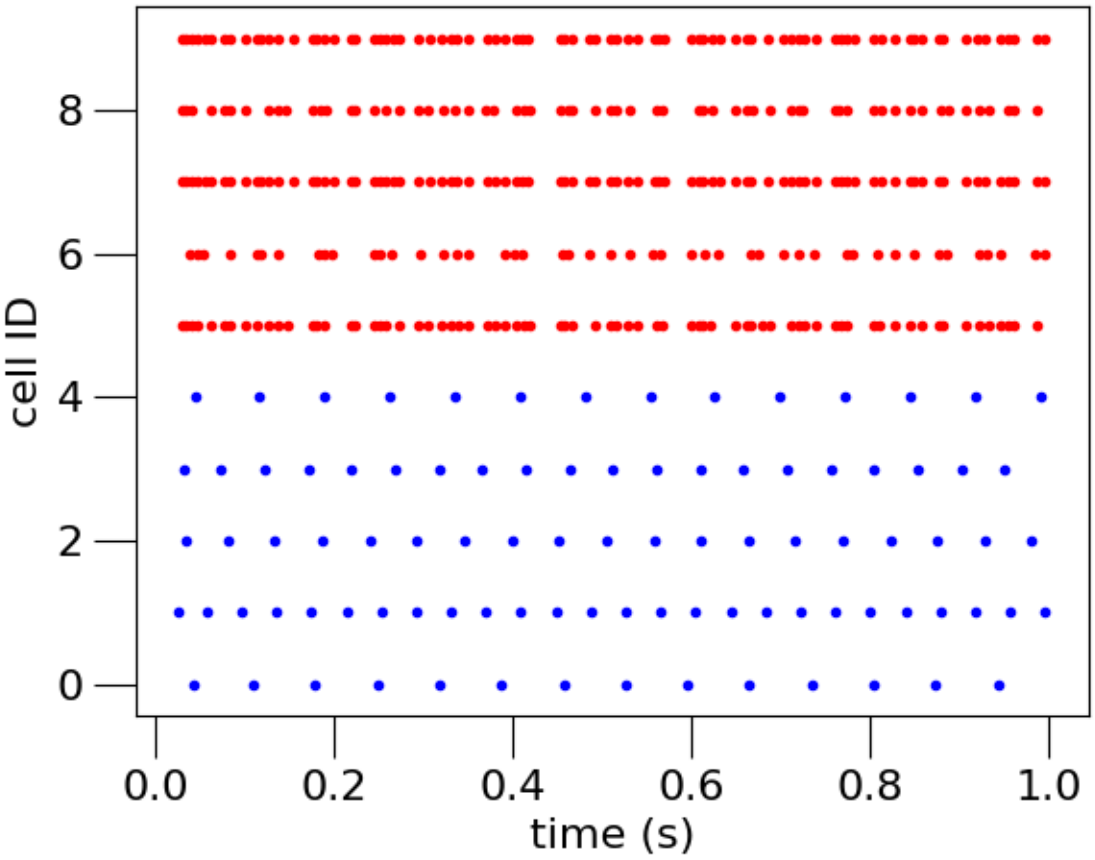


Fig. 4.2: Spike times of neurons in 2 populations recorded from the simulation.

(continued from previous page)

```
nml_doc = component_factory("NeuroMLDocument", id="IzNet")
iz0 = nml_doc.add(
    "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",
)

# Inspect the component, also show all members:
iz0.info(True)

# Create a component of type ExpOneSynapse, and add it to the document
syn0 = nml_doc.add(
    "ExpOneSynapse", id="syn0", gbase="65nS", erev="0mV", tau_decay="3ms"
)
# Check what we have so far:
nml_doc.info(True)
# Also try:
print(nml_doc.summary())

# create the network: turned of validation because we will add populations next
net = nml_doc.add("Network", id="IzNet", validate=False)

# create the first population
size0 = 5
pop0 = component_factory("Population", id="IzPop0", component=iz0.id, size=size0)
# Set optional color property. Note: used later when generating plots
pop0.add("Property", tag="color", value="0 0 .8")
net.add(pop0)

# create the second population
size1 = 5
pop1 = component_factory("Population", id="IzPop1", component=iz0.id, size=size1)
pop1.add("Property", tag="color", value=".8 0 0")
net.add(pop1)

# network should be valid now that it contains populations
net.validate()

# create a projection from one population to another
proj = net.add(
    "Projection",
    id="proj",
    presynaptic_population=pop0.id,
    postsynaptic_population=pop1.id,
    synapse=syn0.id,
)

# We do two things in the loop:
```

(continues on next page)

(continued from previous page)

```
# - add pulse generator inputs to population 1 to make neurons spike
# - create synapses between the two populations with a particular probability
random.seed(123)
prob_connection = 0.8
count = 0
for pre in range(0, size0):
    # pulse generator as explicit stimulus
    pg = nml_doc.add(
        "PulseGenerator",
        id="pg_%i" % pre,
        delay="0ms",
        duration="10000ms",
        amplitude="%f nA" % (0.1 + 0.1 * random.random()),
    )

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

    # synapses between populations
    for post in range(0, size1):
        if random.random() <= prob_connection:
            syn = proj.add(
                "Connection",
                id=count,
                pre_cell_id="../%s[%i]" % (pop0.id, pre),
                post_cell_id="../%s[%i]" % (pop1.id, post),
            )
            count += 1

nml_doc.info(True)
print(nml_doc.summary())

# write model to file and validate
nml_file = "izhikevich2007_network.nml"
writers.NeuroMLWriter.write(nml_doc, nml_file)

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

# Create simulation, and record data
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(
```

(continues on next page)

(continued from previous page)

```
    "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()

# Run the simulation
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. We will use the same helper functions to inspect the model as we build it: `component_factory`, `add`, `info`, `summary`.

4.3.1 Declaring the model in NeuroML

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

```
nml_doc = component_factory("NeuroMLDocument", id="IzNet")
iz0 = nml_doc.add(
    "Izhikevich2007Cell",
    id="iz2007RS0",
    v0="-60mV",
    C="100pF",
    k="0.7nS_per_mV",
    vr="-60mV",
    vt="-40mV",
    vpeak="35mV",
    a="0.03per_ms",
    b="-2nS",
```

(continues on next page)

(continued from previous page)

```
    c="-50.0mV",
    d="100pA",
)

# Inspect the component, also show all members:
iz0.info(True)

# Create a component of type ExpOneSynapse, and add it to the document
syn0 = nml_doc.add(
    "ExpOneSynapse", id="syn0", gbase="65nS", erev="0mV", tau_decay="3ms"
)
```

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. We use the *ExpOne Synapse* here, 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`.

Let's inspect our model document so far:

```
nml_doc.info(True)
Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
 ↪Userdocs/NeuroMLv2.html for more information.

Valid members for NeuroMLDocument are:
...
* izhikevich2007_cells (class: Izhikevich2007Cell, Optional)
        * Contents ('ids'/<objects>): ['iz2007RS0']

* ad_ex_ia_f_cells (class: AdExIaFCell, Optional)
* fitz_hugh_nagumo_cells (class: FitzHughNagumoCell, Optional)
* fitz_hugh_nagumo1969_cells (class: FitzHughNagumo1969Cell, Optional)
* pinsky_rinzel_ca3_cells (class: PinskyRinzelCA3Cell, Optional)
* pulse_generators (class: PulseGenerator, Optional)
* pulse_generator_dls (class: PulseGeneratorDL, Optional)
* id (class: NmlId, Required)
        * Contents ('ids'/<objects>): IzNet

* sine_generators (class: SineGenerator, Optional)
...
* IF_curr_exp (class: IF_curr_exp, Optional)
* exp_one_synapses (class: ExpOneSynapse, Optional)
        * Contents ('ids'/<objects>): ['syn0']

* IF_cond_alpha (class: IF_cond_alpha, Optional)
* exp_two_synapses (class: ExpTwoSynapse, Optional)
...
```

Let's also get a summary:

```
print(nml_doc.summary())
*******************************************************
* NeuroMLDocument: IzNet
*
*  ExpOneSynapse: ['syn0']
*  Izhikevich2007Cell: ['iz2007RS0']
*
*******************************************************
```

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 when we generate our plots below.

```
# create the network: turned of validation because we will add populations next
net = nml_doc.add("Network", id="IzNet", validate=False)

# create the first population
size0 = 5
pop0 = component_factory("Population", id="IzPop0", component=iz0.id, size=size0)
# Set optional color property. Note: used later when generating plots
pop0.add("Property", tag="color", value="0 0 .8")
net.add(pop0)

# create the second population
size1 = 5
pop1 = component_factory("Population", id="IzPop1", component=iz0.id, size=size1)
pop1.add("Property", tag="color", value=".8 0 0")
net.add(pop1)
```

We can test to see if the network is now valid, since we have added the required populations to it:

```
net.validate()
```

This function does not return anything if the component is valid. If it is invalid, however, it will throw a `ValueError`.

We can now 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).

```
# create a projection from one population to another
proj = net.add(
    "Projection",
    id="proj",
    presynaptic_population=pop0.id,
    postsynaptic_population=pop1.id,
    synapse=syn0.id,
)

# We do two things in the loop:
# - add pulse generator inputs to population 1 to make neurons spike
# - create synapses between the two populations with a particular probability
random.seed(123)
prob_connection = 0.8
count = 0
for pre in range(0, size0):
    # pulse generator as explicit stimulus
    pg = nml_doc.add(
        "PulseGenerator",
        id="pg_%i" % pre,
        delay="0ms",
        duration="10000ms",
        amplitude="%f nA" % (0.1 + 0.1 * random.random()),
    )

    exp_input = net.add(
```

(continues on next page)

(continued from previous page)

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

    # synapses between populations
    for post in range(0, size1):
        if random.random() <= prob_connection:
            syn = proj.add(
                "Connection",
                id=count,
                pre_cell_id="../%s[%i]" % (pop0.id, pre),
                post_cell_id="../%s[%i]" % (pop1.id, post),
            )
            count += 1
```

Let us inspect our model again to confirm that we have it set up correctly.

```
nml_doc.info(True)
Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
 ↪Userdocs/NeuroMLv2.html for more information.

Valid members for NeuroMLDocument are:
* biophysical_properties (class: BiophysicalProperties, Optional)
* SpikeSourcePoisson (class: SpikeSourcePoisson, Optional)
* cells (class: Cell, Optional)
* networks (class: Network, Optional)
        * Contents ('ids'/<objects>): ['IzNet']

* cell2_ca_poolses (class: Cell2CaPools, Optional)
...
* izhikevich2007_cells (class: Izhikevich2007Cell, Optional)
        * Contents ('ids'/<objects>): ['iz2007RS0']

* ad_ex_ia_f_cells (class: AdExIaFCell, Optional)
* fitz_hugh_nagumo_cells (class: FitzHughNagumoCell, Optional)
* fitz_hugh_nagumo1969_cells (class: FitzHughNagumo1969Cell, Optional)
* pinsky_rinzel_ca3_cells (class: PinskyRinzelCA3Cell, Optional)
* pulse_generators (class: PulseGenerator, Optional)
        * Contents ('ids'/<objects>): ['pg_0', 'pg_1', 'pg_2', 'pg_3', 'pg_4']

* pulse_generator_dls (class: PulseGeneratorDL, Optional)
* id (class: NmlId, Required)
        * Contents ('ids'/<objects>): IzNet

...
* exp_one_synapses (class: ExpOneSynapse, Optional)
        * Contents ('ids'/<objects>): ['syn0']

* IF_cond_alpha (class: IF_cond_alpha, Optional)
..


print(nml_doc.summary())
******************************************************
* NeuroMLDocument: IzNet
*
*  ExpOneSynapse: ['syn0']
```

(continues on next page)

(continued from previous page)

```
*  Izhikevich2007Cell: ['iz2007RS0']
*  PulseGenerator: ['pg_0', 'pg_1', 'pg_2', 'pg_3', 'pg_4']
*
*  Network: IzNet
*
*   10 cells in 2 populations
*     Population: IzPop0 with 5 components of type iz2007RS0
*       Properties: color=0 0 .8;
*     Population: IzPop1 with 5 components of type iz2007RS0
*       Properties: color=.8 0 0;
*
*   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 pg_0 to IzPop0(cell 0), destination: unspecified
*     Explicit Input of type pg_1 to IzPop0(cell 1), destination: unspecified
*     Explicit Input of type pg_2 to IzPop0(cell 2), destination: unspecified
*     Explicit Input of type pg_3 to IzPop0(cell 3), destination: unspecified
*     Explicit Input of type pg_4 to IzPop0(cell 4), destination: unspecified
*
*********************************************************
```

We can now save and validate our model.

```
# write model to file and validate
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" type="population">
            <property tag="color" value="0 0 .8"/>
```

(continues on next page)

(continued from previous page)

```
        </population>
        <population id="IzPop1" component="iz2007RS0" size="5" type="population">
            <property tag="color" value=".8 0 0"/>
        </population>
        <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 should now be 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 have 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. Similar to the `summary()` function that we have used so far, *pyNeuroML* also includes the helper `pynml-summary` script that can be used to get summaries of NeuroML models from their NeuroML files:

```
$ pynml-summary izhikevich2007_network.nml
******************************************************
* NeuroMLDocument: IzNet
*
*   ExpOneSynapse: ['syn0']
*   Izhikevich2007Cell: ['iz2007RS0']
*   PulseGenerator: ['pulseGen_0', 'pulseGen_1', 'pulseGen_2', 'pulseGen_3',
↪'pulseGen_4']
```

(continues on next page)

(continued from previous page)

```
*
*  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:

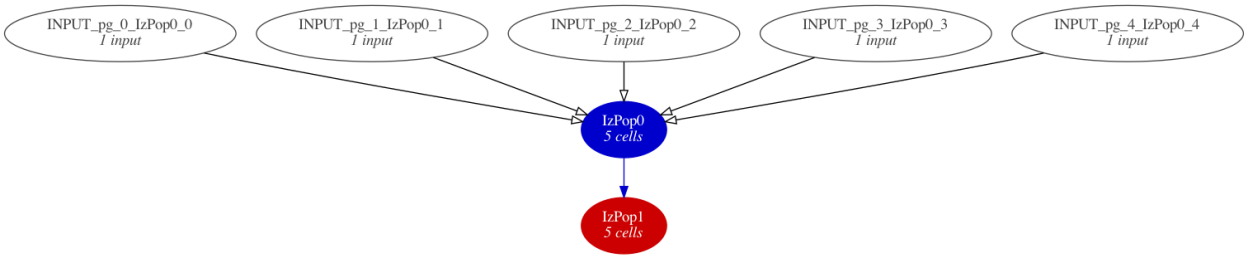


Fig. 4.3: A summary graph of the model generated using pynml and the dot tool.

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

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

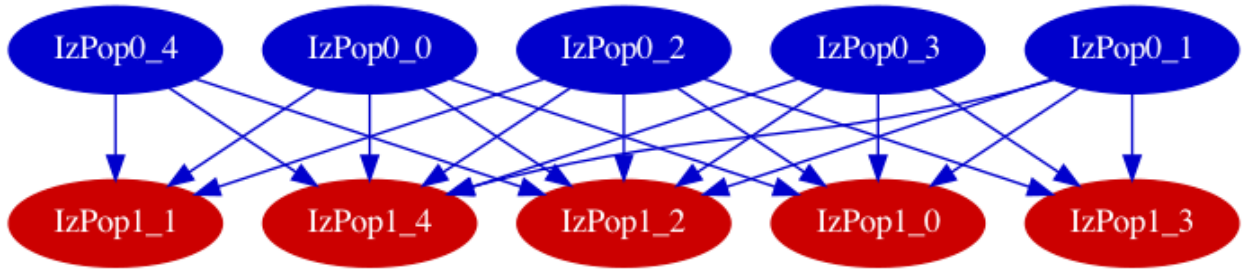


Fig. 4.4: Model summary graph showing individual connections between cells in the populations.

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

4.3.2 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.

```
# Create simulation, and record data
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 v1.1.13␣
↪(libNeuroML v0.5.8)

     -->

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

(continues on next page)

(continued from previous page)

```
    <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:

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

4.3.3 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]
```

(continues on next page)

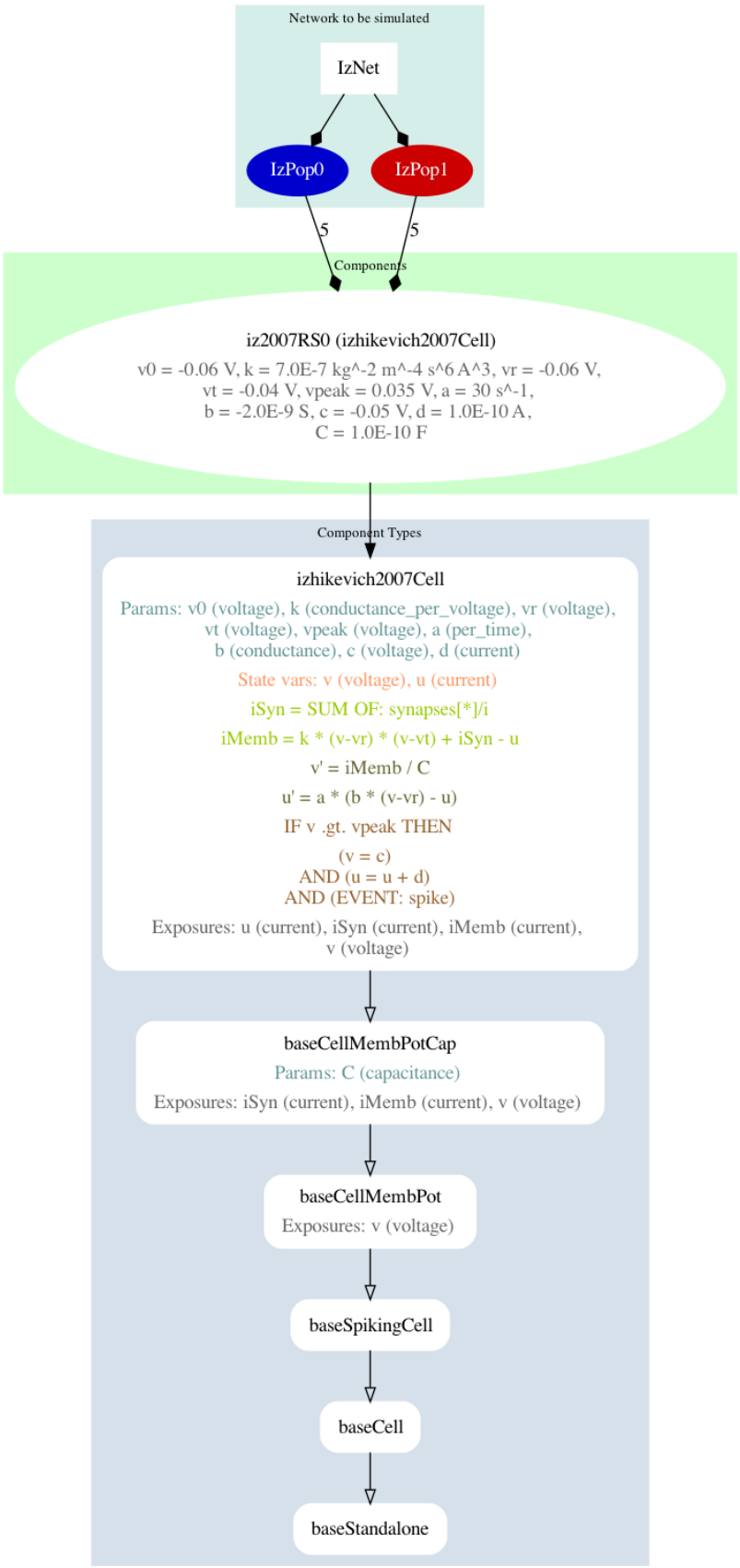


Fig. 4.5: A summary graph of the model generated using pynml -lems-graph.

(continued from previous page)

```
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=[".", "."],
)
```

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 raster plot 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.

4.4 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.
"""

import random
import numpy as np

from neuroml.utils import component_factory
from pyneuroml import pynml
from pyneuroml.lems import LEMSSimulation
import neuroml.writers as writers
```

4.4.1 Declaring the NeuroML model

Create a NeuroML document

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

Declare the Izhikevich cell and add it to the model document

```
iz0 = nml_doc.add(
    "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",
)
# Inspect the component, also show all members:
iz0.info(True)
```

```
Izhikevich2007Cell -- Cell based on the modified Izhikevich model in Izhikevich␣
 ↪2007, Dynamical systems in neuroscience, MIT Press

Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
 ↪Userdocs/NeuroMLv2.html for more information.

Valid members for Izhikevich2007Cell are:
* a (class: Nml2Quantity_pertime, Required)
        * Contents ('ids'/<objects>): 0.03per_ms

* C (class: Nml2Quantity_capacitance, Required)
        * Contents ('ids'/<objects>): 100pF

* annotation (class: Annotation, Optional)
* b (class: Nml2Quantity_conductance, Required)
        * Contents ('ids'/<objects>): -2nS

* metaid (class: MetaId, Optional)
* c (class: Nml2Quantity_voltage, Required)
        * Contents ('ids'/<objects>): -50.0mV

* d (class: Nml2Quantity_current, Required)
        * Contents ('ids'/<objects>): 100pA

* neuro_lex_id (class: NeuroLexId, Optional)
* v0 (class: Nml2Quantity_voltage, Required)
        * Contents ('ids'/<objects>): -60mV

* properties (class: Property, Optional)
* k (class: Nml2Quantity_conductancePerVoltage, Required)
```

(continues on next page)

(continued from previous page)

```
        * Contents ('ids'/<objects>): 0.7nS_per_mV

* notes (class: xs:string, Optional)
* vr (class: Nml2Quantity_voltage, Required)
        * Contents ('ids'/<objects>): -60mV

* vt (class: Nml2Quantity_voltage, Required)
        * Contents ('ids'/<objects>): -40mV

* vpeak (class: Nml2Quantity_voltage, Required)
        * Contents ('ids'/<objects>): 35mV

* id (class: NmlId, Required)
        * Contents ('ids'/<objects>): iz2007RS0
```

```
"Izhikevich2007Cell -- Cell based on the modified Izhikevich model in Izhikevich␣
↪2007, Dynamical systems in neuroscience, MIT Press\n\nPlease see the NeuroML␣
↪standard schema documentation at https://docs.neuroml.org/Userdocs/NeuroMLv2.
↪html for more information.\n\nValid members for Izhikevich2007Cell are:\n* a␣
↪(class: Nml2Quantity_pertime, Required)\n\t* Contents ('ids'/<objects>): 0.03per_
↪ms\n\n* C (class: Nml2Quantity_capacitance, Required)\n\t* Contents ('ids'/
↪<objects>): 100pF\n\n* annotation (class: Annotation, Optional)\n* b (class:␣
↪Nml2Quantity_conductance, Required)\n\t* Contents ('ids'/<objects>): -2nS\n\n*␣
↪metaid (class: MetaId, Optional)\n* c (class: Nml2Quantity_voltage, Required)\n\
↪t* Contents ('ids'/<objects>): -50.0mV\n\n* d (class: Nml2Quantity_current,␣
↪Required)\n\t* Contents ('ids'/<objects>): 100pA\n\n* neuro_lex_id (class:␣
↪NeuroLexId, Optional)\n* v0 (class: Nml2Quantity_voltage, Required)\n\t*␣
↪Contents ('ids'/<objects>): -60mV\n\n* properties (class: Property, Optional)\n*␣
↪k (class: Nml2Quantity_conductancePerVoltage, Required)\n\t* Contents ('ids'/
↪<objects>): 0.7nS_per_mV\n\n* notes (class: xs:string, Optional)\n* vr (class:␣
↪Nml2Quantity_voltage, Required)\n\t* Contents ('ids'/<objects>): -60mV\n\n* vt␣
↪(class: Nml2Quantity_voltage, Required)\n\t* Contents ('ids'/<objects>): -40mV\n\
↪n* vpeak (class: Nml2Quantity_voltage, Required)\n\t* Contents ('ids'/<objects>
↪): 35mV\n\n* id (class: NmlId, Required)\n\t* Contents ('ids'/<objects>):␣
↪iz2007RS0\n\n"
```

Declare the Synapse and add it to the model document

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

```
nml_doc.info(True)
```

```
Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
↪Userdocs/NeuroMLv2.html for more information.

Valid members for NeuroMLDocument are:
* compound_inputs (class: CompoundInput, Optional)
* compound_input_dls (class: CompoundInputDL, Optional)
* includes (class: IncludeType, Optional)
* voltage_clamps (class: VoltageClamp, Optional)
* extracellular_properties (class: ExtracellularProperties, Optional)
* voltage_clamp_triples (class: VoltageClampTriple, Optional)
```

(continues on next page)

(continued from previous page)

```
* intracellular_properties (class: IntracellularProperties, Optional)
* spike_arrays (class: SpikeArray, Optional)
* morphology (class: Morphology, Optional)
* timed_synaptic_inputs (class: TimedSynapticInput, Optional)
* ion_channel (class: IonChannel, Optional)
* spike_generators (class: SpikeGenerator, Optional)
* ion_channel_hhs (class: IonChannelHH, Optional)
* spike_generator_randoms (class: SpikeGeneratorRandom, Optional)
* ion_channel_v_shifts (class: IonChannelVShift, Optional)
* spike_generator_poissons (class: SpikeGeneratorPoisson, Optional)
* ion_channel_kses (class: IonChannelKS, Optional)
* spike_generator_ref_poissons (class: SpikeGeneratorRefPoisson, Optional)
* decaying_pool_concentration_models (class: DecayingPoolConcentrationModel,␣
 ↪Optional)
* poisson_firing_synapses (class: PoissonFiringSynapse, Optional)
* fixed_factor_concentration_models (class: FixedFactorConcentrationModel,␣
 ↪Optional)
* transient_poisson_firing_synapses (class: TransientPoissonFiringSynapse,␣
 ↪Optional)
* alpha_current_synapses (class: AlphaCurrentSynapse, Optional)
* IF_curr_alpha (class: IF_curr_alpha, Optional)
* alpha_synapses (class: AlphaSynapse, Optional)
* IF_curr_exp (class: IF_curr_exp, Optional)
* exp_one_synapses (class: ExpOneSynapse, Optional)
        * Contents ('ids'/<objects>): ['syn0']

* IF_cond_alpha (class: IF_cond_alpha, Optional)
* exp_two_synapses (class: ExpTwoSynapse, Optional)
* IF_cond_exp (class: IF_cond_exp, Optional)
* exp_three_synapses (class: ExpThreeSynapse, Optional)
* EIF_cond_exp_isfa_ista (class: EIF_cond_exp_isfa_ista, Optional)
* blocking_plastic_synapses (class: BlockingPlasticSynapse, Optional)
* EIF_cond_alpha_isfa_ista (class: EIF_cond_alpha_isfa_ista, Optional)
* double_synapses (class: DoubleSynapse, Optional)
* HH_cond_exp (class: HH_cond_exp, Optional)
* gap_junctions (class: GapJunction, Optional)
* exp_cond_synapses (class: ExpCondSynapse, Optional)
* silent_synapses (class: SilentSynapse, Optional)
* alpha_cond_synapses (class: AlphaCondSynapse, Optional)
* linear_graded_synapses (class: LinearGradedSynapse, Optional)
* exp_curr_synapses (class: ExpCurrSynapse, Optional)
* graded_synapses (class: GradedSynapse, Optional)
* alpha_curr_synapses (class: AlphaCurrSynapse, Optional)
* annotation (class: Annotation, Optional)
* biophysical_properties (class: BiophysicalProperties, Optional)
* SpikeSourcePoisson (class: SpikeSourcePoisson, Optional)
* cells (class: Cell, Optional)
* networks (class: Network, Optional)
* cell2_ca_poolses (class: Cell2CaPools, Optional)
* ComponentType (class: ComponentType, Optional)
* base_cells (class: BaseCell, Optional)
* iaf_tau_cells (class: IafTauCell, Optional)
* properties (class: Property, Optional)
* iaf_tau_ref_cells (class: IafTauRefCell, Optional)
* notes (class: xs:string, Optional)
* iaf_cells (class: IafCell, Optional)
* metaid (class: MetaId, Optional)
```

(continues on next page)

(continued from previous page)

```
* iaf_ref_cells (class: IafRefCell, Optional)
* izhikevich_cells (class: IzhikevichCell, Optional)
* izhikevich2007_cells (class: Izhikevich2007Cell, Optional)
        * Contents ('ids'/<objects>): ['iz2007RS0']

* ad_ex_ia_f_cells (class: AdExIaFCell, Optional)
* fitz_hugh_nagumo_cells (class: FitzHughNagumoCell, Optional)
* fitz_hugh_nagumo1969_cells (class: FitzHughNagumo1969Cell, Optional)
* pinsky_rinzel_ca3_cells (class: PinskyRinzelCA3Cell, Optional)
* pulse_generators (class: PulseGenerator, Optional)
* pulse_generator_dls (class: PulseGeneratorDL, Optional)
* sine_generators (class: SineGenerator, Optional)
* sine_generator_dls (class: SineGeneratorDL, Optional)
* ramp_generators (class: RampGenerator, Optional)
* ramp_generator_dls (class: RampGeneratorDL, Optional)
* id (class: NmlId, Required)
        * Contents ('ids'/<objects>): IzNet
```

```
"Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
 ↪Userdocs/NeuroMLv2.html for more information.\n\nValid members for␣
 ↪NeuroMLDocument are:\n* compound_inputs (class: CompoundInput, Optional)\n*␣
 ↪compound_input_dls (class: CompoundInputDL, Optional)\n* includes (class:␣
 ↪IncludeType, Optional)\n* voltage_clamps (class: VoltageClamp, Optional)\n*␣
 ↪extracellular_properties (class: ExtracellularProperties, Optional)\n* voltage_
 ↪clamp_triples (class: VoltageClampTriple, Optional)\n* intracellular_properties␣
 ↪(class: IntracellularProperties, Optional)\n* spike_arrays (class: SpikeArray,␣
 ↪Optional)\n* morphology (class: Morphology, Optional)\n* timed_synaptic_inputs␣
 ↪(class: TimedSynapticInput, Optional)\n* ion_channel (class: IonChannel,␣
 ↪Optional)\n* spike_generators (class: SpikeGenerator, Optional)\n* ion_channel_
 ↪hhs (class: IonChannelHH, Optional)\n* spike_generator_randoms (class:␣
 ↪SpikeGeneratorRandom, Optional)\n* ion_channel_v_shifts (class: IonChannelVShift,
 ↪ Optional)\n* spike_generator_poissons (class: SpikeGeneratorPoisson, Optional)\
 ↪n* ion_channel_kses (class: IonChannelKS, Optional)\n* spike_generator_ref_
 ↪poissons (class: SpikeGeneratorRefPoisson, Optional)\n* decaying_pool_
 ↪concentration_models (class: DecayingPoolConcentrationModel, Optional)\n*␣
 ↪poisson_firing_synapses (class: PoissonFiringSynapse, Optional)\n* fixed_factor_
 ↪concentration_models (class: FixedFactorConcentrationModel, Optional)\n*␣
 ↪transient_poisson_firing_synapses (class: TransientPoissonFiringSynapse,␣
 ↪Optional)\n* alpha_current_synapses (class: AlphaCurrentSynapse, Optional)\n* IF_
 ↪curr_alpha (class: IF_curr_alpha, Optional)\n* alpha_synapses (class:␣
 ↪AlphaSynapse, Optional)\n* IF_curr_exp (class: IF_curr_exp, Optional)\n* exp_one_
 ↪synapses (class: ExpOneSynapse, Optional)\n\t* Contents ('ids'/<objects>): ['syn0
 ↪']\n\n* IF_cond_alpha (class: IF_cond_alpha, Optional)\n* exp_two_synapses␣
 ↪(class: ExpTwoSynapse, Optional)\n* IF_cond_exp (class: IF_cond_exp, Optional)\
 ↪n* exp_three_synapses (class: ExpThreeSynapse, Optional)\n* EIF_cond_exp_isfa_
 ↪ista (class: EIF_cond_exp_isfa_ista, Optional)\n* blocking_plastic_synapses␣
 ↪(class: BlockingPlasticSynapse, Optional)\n* EIF_cond_alpha_isfa_ista (class:␣
 ↪EIF_cond_alpha_isfa_ista, Optional)\n* double_synapses (class: DoubleSynapse,␣
 ↪Optional)\n* HH_cond_exp (class: HH_cond_exp, Optional)\n* gap_junctions (class:␣
 ↪GapJunction, Optional)\n* exp_cond_synapses (class: ExpCondSynapse, Optional)\n*␣
 ↪silent_synapses (class: SilentSynapse, Optional)\n* alpha_cond_synapses (class:␣
 ↪AlphaCondSynapse, Optional)\n* linear_graded_synapses (class:␣
 ↪LinearGradedSynapse, Optional)\n* exp_curr_synapses (class: ExpCurrSynapse,␣
 ↪Optional)\n* graded_synapses (class: GradedSynapse, Optional)\n* alpha_curr_
 ↪synapses (class: AlphaCurrSynapse, Optional)\n* annotation (class: Annotation,␣
 ↪Optional)\n* biophysical_properties (class: BiophysicalProperties, Optional)\n*␣
```

(continues on next page)

(continued from previous page)

```
↪SpikeSourcePoisson (class: SpikeSourcePoisson, Optional)\n* cells (class: Cell,␣
↪Optional)\n* networks (class: Network, Optional)\n* cell2_ca_poolses (class:␣
↪Cell2CaPools, Optional)\n* ComponentType (class: ComponentType, Optional)\n*␣
↪base_cells (class: BaseCell, Optional)\n* iaf_tau_cells (class: IafTauCell,␣
↪Optional)\n* properties (class: Property, Optional)\n* iaf_tau_ref_cells (class:␣
↪IafTauRefCell, Optional)\n* notes (class: xs:string, Optional)\n* iaf_cells␣
↪(class: IafCell, Optional)\n* metaid (class: MetaId, Optional)\n* iaf_ref_cells␣
↪(class: IafRefCell, Optional)\n* izhikevich_cells (class: IzhikevichCell,␣
↪Optional)\n* izhikevich2007_cells (class: Izhikevich2007Cell, Optional)\n\t*␣
↪Contents ('ids'/<objects>): ['iz2007RS0']\n\n* ad_ex_ia_f_cells (class:␣
↪AdExIaFCell, Optional)\n* fitz_hugh_nagumo_cells (class: FitzHughNagumoCell,␣
↪Optional)\n* fitz_hugh_nagumo1969_cells (class: FitzHughNagumo1969Cell,␣
↪Optional)\n* pinsky_rinzel_ca3_cells (class: PinskyRinzelCA3Cell, Optional)\n*␣
↪pulse_generators (class: PulseGenerator, Optional)\n* pulse_generator_dls␣
↪(class: PulseGeneratorDL, Optional)\n* sine_generators (class: SineGenerator,␣
↪Optional)\n* sine_generator_dls (class: SineGeneratorDL, Optional)\n* ramp_
↪generators (class: RampGenerator, Optional)\n* ramp_generator_dls (class:␣
↪RampGeneratorDL, Optional)\n* id (class: NmlId, Required)\n\t* Contents ('ids'/
↪<objects>): IzNet\n\n"
```

```
print(nml_doc.summary())
```

```
******************************************************
* NeuroMLDocument: IzNet
*
*  ExpOneSynapse: ['syn0']
*  Izhikevich2007Cell: ['iz2007RS0']
*
******************************************************
```

Declare a Network and add it to the model document

```
net = nml_doc.add("Network", id="IzNet", validate=False)
```

Create two populations

```
# create the first population
size0 = 5
pop0 = component_factory("Population", id="IzPop0", component=iz0.id, size=size0,␣
 ↪type="population")
# Set optional color property. Note: used later when generating plots
pop0.add("Property", tag="color", value="0 0 .8")
net.add(pop0)

# create the second population
size1 = 5
pop1 = component_factory("Population", id="IzPop1", component=iz0.id, size=size1,␣
 ↪type="population")
pop1.add("Property", tag="color", value=".8 0 0")
net.add(pop1)
```

```
<neuroml.nml.nml.Population at 0x7f53ace425c0>
```

```
net.validate()
```

Declare projections

```
# create a projection from one population to another
proj = net.add(
    "Projection",
    id="proj",
    presynaptic_population=pop0.id,
    postsynaptic_population=pop1.id,
    synapse=syn0.id,
)
```

Add the projections between populations and the external inputs

```
random.seed(123)
prob_connection = 0.8
count = 0
for pre in range(0, size0):
    # pulse generator as explicit stimulus
    pg = nml_doc.add(
        "PulseGenerator",
        id="pg_%i" % pre,
        delay="0ms",
        duration="10000ms",
        amplitude="%f nA" % (0.1 + 0.1 * random.random()),
    )

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

    # synapses between populations
    for post in range(0, size1):
        if random.random() <= prob_connection:
            syn = proj.add(
                "Connection",
                id=count,
                pre_cell_id="../%s[%i]" % (pop0.id, pre),
                post_cell_id="../%s[%i]" % (pop1.id, post),
            )
            count += 1
```

```
nml_doc.info(True)
```

```
Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
 ↪Userdocs/NeuroMLv2.html for more information.

Valid members for NeuroMLDocument are:
* compound_inputs (class: CompoundInput, Optional)
```

(continues on next page)

(continued from previous page)

```
* compound_input_dls (class: CompoundInputDL, Optional)
* includes (class: IncludeType, Optional)
* voltage_clamps (class: VoltageClamp, Optional)
* extracellular_properties (class: ExtracellularProperties, Optional)
* voltage_clamp_triples (class: VoltageClampTriple, Optional)
* intracellular_properties (class: IntracellularProperties, Optional)
* spike_arrays (class: SpikeArray, Optional)
* morphology (class: Morphology, Optional)
* timed_synaptic_inputs (class: TimedSynapticInput, Optional)
* ion_channel (class: IonChannel, Optional)
* spike_generators (class: SpikeGenerator, Optional)
* ion_channel_hhs (class: IonChannelHH, Optional)
* spike_generator_randoms (class: SpikeGeneratorRandom, Optional)
* ion_channel_v_shifts (class: IonChannelVShift, Optional)
* spike_generator_poissons (class: SpikeGeneratorPoisson, Optional)
* ion_channel_kses (class: IonChannelKS, Optional)
* spike_generator_ref_poissons (class: SpikeGeneratorRefPoisson, Optional)
* decaying_pool_concentration_models (class: DecayingPoolConcentrationModel,␣
 ↪Optional)
* poisson_firing_synapses (class: PoissonFiringSynapse, Optional)
* fixed_factor_concentration_models (class: FixedFactorConcentrationModel,␣
 ↪Optional)
* transient_poisson_firing_synapses (class: TransientPoissonFiringSynapse,␣
 ↪Optional)
* alpha_current_synapses (class: AlphaCurrentSynapse, Optional)
* IF_curr_alpha (class: IF_curr_alpha, Optional)
* alpha_synapses (class: AlphaSynapse, Optional)
* IF_curr_exp (class: IF_curr_exp, Optional)
* exp_one_synapses (class: ExpOneSynapse, Optional)
        * Contents ('ids'/<objects>): ['syn0']

* IF_cond_alpha (class: IF_cond_alpha, Optional)
* exp_two_synapses (class: ExpTwoSynapse, Optional)
* IF_cond_exp (class: IF_cond_exp, Optional)
* exp_three_synapses (class: ExpThreeSynapse, Optional)
* EIF_cond_exp_isfa_ista (class: EIF_cond_exp_isfa_ista, Optional)
* blocking_plastic_synapses (class: BlockingPlasticSynapse, Optional)
* EIF_cond_alpha_isfa_ista (class: EIF_cond_alpha_isfa_ista, Optional)
* double_synapses (class: DoubleSynapse, Optional)
* HH_cond_exp (class: HH_cond_exp, Optional)
* gap_junctions (class: GapJunction, Optional)
* exp_cond_synapses (class: ExpCondSynapse, Optional)
* silent_synapses (class: SilentSynapse, Optional)
* alpha_cond_synapses (class: AlphaCondSynapse, Optional)
* linear_graded_synapses (class: LinearGradedSynapse, Optional)
* exp_curr_synapses (class: ExpCurrSynapse, Optional)
* graded_synapses (class: GradedSynapse, Optional)
* alpha_curr_synapses (class: AlphaCurrSynapse, Optional)
* annotation (class: Annotation, Optional)
* biophysical_properties (class: BiophysicalProperties, Optional)
* SpikeSourcePoisson (class: SpikeSourcePoisson, Optional)
* cells (class: Cell, Optional)
* networks (class: Network, Optional)
        * Contents ('ids'/<objects>): ['IzNet']

* cell2_ca_poolses (class: Cell2CaPools, Optional)
* ComponentType (class: ComponentType, Optional)
```

(continues on next page)

(continued from previous page)

```
* base_cells (class: BaseCell, Optional)
* iaf_tau_cells (class: IafTauCell, Optional)
* properties (class: Property, Optional)
* iaf_tau_ref_cells (class: IafTauRefCell, Optional)
* notes (class: xs:string, Optional)
* iaf_cells (class: IafCell, Optional)
* metaid (class: MetaId, Optional)
* iaf_ref_cells (class: IafRefCell, Optional)
* izhikevich_cells (class: IzhikevichCell, Optional)
* izhikevich2007_cells (class: Izhikevich2007Cell, Optional)
        * Contents ('ids'/<objects>): ['iz2007RS0']

* ad_ex_ia_f_cells (class: AdExIaFCell, Optional)
* fitz_hugh_nagumo_cells (class: FitzHughNagumoCell, Optional)
* fitz_hugh_nagumo1969_cells (class: FitzHughNagumo1969Cell, Optional)
* pinsky_rinzel_ca3_cells (class: PinskyRinzelCA3Cell, Optional)
* pulse_generators (class: PulseGenerator, Optional)
        * Contents ('ids'/<objects>): ['pg_0', 'pg_1', 'pg_2', 'pg_3', 'pg_4']

* pulse_generator_dls (class: PulseGeneratorDL, Optional)
* sine_generators (class: SineGenerator, Optional)
* sine_generator_dls (class: SineGeneratorDL, Optional)
* ramp_generators (class: RampGenerator, Optional)
* ramp_generator_dls (class: RampGeneratorDL, Optional)
* id (class: NmlId, Required)
        * Contents ('ids'/<objects>): IzNet
```

```
"Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
 ↪Userdocs/NeuroMLv2.html for more information.\n\nValid members for␣
 ↪NeuroMLDocument are:\n* compound_inputs (class: CompoundInput, Optional)\n*␣
 ↪compound_input_dls (class: CompoundInputDL, Optional)\n* includes (class:␣
 ↪IncludeType, Optional)\n* voltage_clamps (class: VoltageClamp, Optional)\n*␣
 ↪extracellular_properties (class: ExtracellularProperties, Optional)\n* voltage_
 ↪clamp_triples (class: VoltageClampTriple, Optional)\n* intracellular_properties␣
 ↪(class: IntracellularProperties, Optional)\n* spike_arrays (class: SpikeArray,␣
 ↪Optional)\n* morphology (class: Morphology, Optional)\n* timed_synaptic_inputs␣
 ↪(class: TimedSynapticInput, Optional)\n* ion_channel (class: IonChannel,␣
 ↪Optional)\n* spike_generators (class: SpikeGenerator, Optional)\n* ion_channel_
 ↪hhs (class: IonChannelHH, Optional)\n* spike_generator_randoms (class:␣
 ↪SpikeGeneratorRandom, Optional)\n* ion_channel_v_shifts (class: IonChannelVShift,
 ↪ Optional)\n* spike_generator_poissons (class: SpikeGeneratorPoisson, Optional)\
 ↪n* ion_channel_kses (class: IonChannelKS, Optional)\n* spike_generator_ref_
 ↪poissons (class: SpikeGeneratorRefPoisson, Optional)\n* decaying_pool_
 ↪concentration_models (class: DecayingPoolConcentrationModel, Optional)\n*␣
 ↪poisson_firing_synapses (class: PoissonFiringSynapse, Optional)\n* fixed_factor_
 ↪concentration_models (class: FixedFactorConcentrationModel, Optional)\n*␣
 ↪transient_poisson_firing_synapses (class: TransientPoissonFiringSynapse,␣
 ↪Optional)\n* alpha_current_synapses (class: AlphaCurrentSynapse, Optional)\n* IF_
 ↪curr_alpha (class: IF_curr_alpha, Optional)\n* alpha_synapses (class:␣
 ↪AlphaSynapse, Optional)\n* IF_curr_exp (class: IF_curr_exp, Optional)\n* exp_one_
 ↪synapses (class: ExpOneSynapse, Optional)\n\t* Contents ('ids'/<objects>): ['syn0
 ↪']\n\n* IF_cond_alpha (class: IF_cond_alpha, Optional)\n* exp_two_synapses␣
 ↪(class: ExpTwoSynapse, Optional)\n* IF_cond_exp (class: IF_cond_exp, Optional)\
 ↪n* exp_three_synapses (class: ExpThreeSynapse, Optional)\n* EIF_cond_exp_isfa_
 ↪ista (class: EIF_cond_exp_isfa_ista, Optional)\n* blocking_plastic_synapses␣
 ↪(class: BlockingPlasticSynapse, Optional)\n* EIF_cond_alpha_isfa_ista (class:␣
```

(continues on next page)

(continued from previous page)

```
↪EIF_cond_alpha_isfa_ista, Optional)\n* double_synapses (class: DoubleSynapse,␣
↪Optional)\n* HH_cond_exp (class: HH_cond_exp, Optional)\n* gap_junctions (class:␣
↪GapJunction, Optional)\n* exp_cond_synapses (class: ExpCondSynapse, Optional)\n*␣
↪silent_synapses (class: SilentSynapse, Optional)\n* alpha_cond_synapses (class:␣
↪AlphaCondSynapse, Optional)\n* linear_graded_synapses (class:␣
↪LinearGradedSynapse, Optional)\n* exp_curr_synapses (class: ExpCurrSynapse,␣
↪Optional)\n* graded_synapses (class: GradedSynapse, Optional)\n* alpha_curr_
↪synapses (class: AlphaCurrSynapse, Optional)\n* annotation (class: Annotation,␣
↪Optional)\n* biophysical_properties (class: BiophysicalProperties, Optional)\n*␣
↪SpikeSourcePoisson (class: SpikeSourcePoisson, Optional)\n* cells (class: Cell,␣
↪Optional)\n* networks (class: Network, Optional)\n\t* Contents ('ids'/<objects>
↪): ['IzNet']\n\n* cell2_ca_poolses (class: Cell2CaPools, Optional)\n*␣
↪ComponentType (class: ComponentType, Optional)\n* base_cells (class: BaseCell,␣
↪Optional)\n* iaf_tau_cells (class: IafTauCell, Optional)\n* properties (class:␣
↪Property, Optional)\n* iaf_tau_ref_cells (class: IafTauRefCell, Optional)\n*␣
↪notes (class: xs:string, Optional)\n* iaf_cells (class: IafCell, Optional)\n*␣
↪metaid (class: MetaId, Optional)\n* iaf_ref_cells (class: IafRefCell, Optional)\
↪n* izhikevich_cells (class: IzhikevichCell, Optional)\n* izhikevich2007_cells␣
↪(class: Izhikevich2007Cell, Optional)\n\t* Contents ('ids'/<objects>): [
↪'iz2007RS0']\n\n* ad_ex_ia_f_cells (class: AdExIaFCell, Optional)\n* fitz_hugh_
↪nagumo_cells (class: FitzHughNagumoCell, Optional)\n* fitz_hugh_nagumo1969_cells␣
↪(class: FitzHughNagumo1969Cell, Optional)\n* pinsky_rinzel_ca3_cells (class:␣
↪PinskyRinzelCA3Cell, Optional)\n* pulse_generators (class: PulseGenerator,␣
↪Optional)\n\t* Contents ('ids'/<objects>): ['pg_0', 'pg_1', 'pg_2', 'pg_3', 'pg_4
↪']\n\n* pulse_generator_dls (class: PulseGeneratorDL, Optional)\n* sine_
↪generators (class: SineGenerator, Optional)\n* sine_generator_dls (class:␣
↪SineGeneratorDL, Optional)\n* ramp_generators (class: RampGenerator, Optional)\
↪n* ramp_generator_dls (class: RampGeneratorDL, Optional)\n* id (class: NmlId,␣
↪Required)\n\t* Contents ('ids'/<objects>): IzNet\n\n"
```

```
print(nml_doc.summary())
```

```
******************************************************
* NeuroMLDocument: IzNet
*
*  ExpOneSynapse: ['syn0']
*  Izhikevich2007Cell: ['iz2007RS0']
*  PulseGenerator: ['pg_0', 'pg_1', 'pg_2', 'pg_3', 'pg_4']
*
*  Network: IzNet
*
*   10 cells in 2 populations
*     Population: IzPop0 with 5 components of type iz2007RS0
*       Properties: color=0 0 .8;
*     Population: IzPop1 with 5 components of type iz2007RS0
*       Properties: color=.8 0 0;
*
*   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 pg_0 to IzPop0(cell 0), destination: unspecified
*     Explicit Input of type pg_1 to IzPop0(cell 1), destination: unspecified
```

(continues on next page)

(continued from previous page)

```
*     Explicit Input of type pg_2 to IzPop0(cell 2), destination: unspecified
*     Explicit Input of type pg_3 to IzPop0(cell 3), destination: unspecified
*     Explicit Input of type pg_4 to IzPop0(cell 4), destination: unspecified
*
*******************************************************
```

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)
```

```
pyNeuroML >>> INFO - Executing: (java -Xmx400M  -jar  "/home/asinha/Documents/02_
 ↪Code/00_mine/NeuroML/documentation/.venv/lib/python3.10/site-packages/pyneuroml/
 ↪lib/jNeuroML-0.12.0-jar-with-dependencies.jar" -validate "izhikevich2007_network.
 ↪nml" ) in directory: .
```

```
Written network file to: izhikevich2007_network.nml
```

```
pyNeuroML >>> INFO - Command completed. Output:
  jNeuroML >>   jNeuroML v0.12.0
  jNeuroML >>  Validating: /home/asinha/Documents/02_Code/00_mine/NeuroML/
 ↪documentation/source/Userdocs/NML2_examples/izhikevich2007_network.nml
  jNeuroML >>  Valid against schema and all tests
  jNeuroML >>  No warnings
  jNeuroML >>
  jNeuroML >>  Validated 1 files: All valid and no warnings
  jNeuroML >>
  jNeuroML >>
```

```
True
```

4.4.2 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)
```

```
pyNeuroML >>> INFO - Loading NeuroML2 file: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/izhikevich2007_network.nml
```

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')
```

4.4.3 Save the simulation to a file

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

4.4.4 Run the simulation using jNeuroML

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

```
pyNeuroML >>> INFO - Loading LEMS file: LEMS_example_izhikevich2007network_sim.xml␣
↪and running with jNeuroML_NEURON
pyNeuroML >>> INFO - Executing: (java -Xmx2G  -Djava.awt.headless=true -jar  "/
↪home/asinha/Documents/02_Code/00_mine/NeuroML/documentation/.venv/lib/python3.10/
↪site-packages/pyneuroml/lib/jNeuroML-0.12.0-jar-with-dependencies.jar"  "LEMS_
↪example_izhikevich2007network_sim.xml"  -neuron -run -compile -nogui -I '') in␣
↪directory: .
pyNeuroML >>> INFO - Command completed. Output:
  jNeuroML >>   jNeuroML v0.12.0
  jNeuroML >>  (INFO) Reading from: /home/asinha/Documents/02_Code/00_mine/NeuroML/
↪documentation/source/Userdocs/NML2_examples/LEMS_example_izhikevich2007network_
↪sim.xml
  jNeuroML >>  (INFO) Creating NeuronWriter to output files to /home/asinha/
↪Documents/02_Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_examples
  jNeuroML >>  (INFO) Adding simulation Component(id=example_izhikevich2007network_
↪sim type=Simulation) of network/component: IzNet (Type: network)
  jNeuroML >>  (INFO) Adding population: IzPop0
  jNeuroML >>  (INFO) -- Writing to mod: /home/asinha/Documents/02_Code/00_mine/
↪NeuroML/documentation/source/Userdocs/NML2_examples/iz2007RS0.mod
  jNeuroML >>  (INFO) File /home/asinha/Documents/02_Code/00_mine/NeuroML/
↪documentation/source/Userdocs/NML2_examples/iz2007RS0.mod exists and is identical
  jNeuroML >>  (INFO) Adding population: IzPop1
  jNeuroML >>  (INFO) -- Mod file for: iz2007RS0 has already been created
  jNeuroML >>  (INFO) Adding projections/connections...
  jNeuroML >>  (INFO) -- Writing to mod: /home/asinha/Documents/02_Code/00_mine/
↪NeuroML/documentation/source/Userdocs/NML2_examples/syn0.mod
  jNeuroML >>  (INFO) File /home/asinha/Documents/02_Code/00_mine/NeuroML/
```

(continues on next page)

(continued from previous page)

```
↪documentation/source/Userdocs/NML2_examples/syn0.mod exists and is identical
 jNeuroML >>  (INFO) -- Writing to mod: /home/asinha/Documents/02_Code/00_mine/
↪NeuroML/documentation/source/Userdocs/NML2_examples/pg_0.mod
 jNeuroML >>  (INFO) File /home/asinha/Documents/02_Code/00_mine/NeuroML/
↪documentation/source/Userdocs/NML2_examples/pg_0.mod exists and is identical
 jNeuroML >>  (INFO) -- Writing to mod: /home/asinha/Documents/02_Code/00_mine/
↪NeuroML/documentation/source/Userdocs/NML2_examples/pg_1.mod
 jNeuroML >>  (INFO) File /home/asinha/Documents/02_Code/00_mine/NeuroML/
↪documentation/source/Userdocs/NML2_examples/pg_1.mod exists and is identical
 jNeuroML >>  (INFO) -- Writing to mod: /home/asinha/Documents/02_Code/00_mine/
↪NeuroML/documentation/source/Userdocs/NML2_examples/pg_2.mod
 jNeuroML >>  (INFO) File /home/asinha/Documents/02_Code/00_mine/NeuroML/
↪documentation/source/Userdocs/NML2_examples/pg_2.mod exists and is identical
 jNeuroML >>  (INFO) -- Writing to mod: /home/asinha/Documents/02_Code/00_mine/
↪NeuroML/documentation/source/Userdocs/NML2_examples/pg_3.mod
 jNeuroML >>  (INFO) File /home/asinha/Documents/02_Code/00_mine/NeuroML/
↪documentation/source/Userdocs/NML2_examples/pg_3.mod exists and is identical
 jNeuroML >>  (INFO) -- Writing to mod: /home/asinha/Documents/02_Code/00_mine/
↪NeuroML/documentation/source/Userdocs/NML2_examples/pg_4.mod
 jNeuroML >>  (INFO) File /home/asinha/Documents/02_Code/00_mine/NeuroML/
↪documentation/source/Userdocs/NML2_examples/pg_4.mod exists and is identical
 jNeuroML >>  (INFO) Trying to compile mods in: /home/asinha/Documents/02_Code/00_
↪mine/NeuroML/documentation/source/Userdocs/NML2_examples
 jNeuroML >>  (INFO) Going to compile the mod files in: /home/asinha/Documents/02_
↪Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_examples, forcing␣
↪recompile: false
 jNeuroML >>  (INFO) Parent dir: /home/asinha/Documents/02_Code/00_mine/NeuroML/
↪documentation/source/Userdocs/NML2_examples
 jNeuroML >>  (INFO) Assuming *nix environment...
 jNeuroML >>  (INFO) Name of file to be created: /home/asinha/Documents/02_Code/
↪00_mine/NeuroML/documentation/source/Userdocs/NML2_examples/x86_64/libnrnmech.la
 jNeuroML >>  (INFO) Name of file to be created: /home/asinha/Documents/02_Code/
↪00_mine/NeuroML/documentation/source/Userdocs/NML2_examples/x86_64/libnrnmech.so
 jNeuroML >>  (INFO) Name of file to be created: /home/asinha/Documents/02_Code/
↪00_mine/NeuroML/documentation/source/Userdocs/NML2_examples/x86_64/.libs/
↪libnrnmech.so
 jNeuroML >>  (INFO) commandToExecute: /usr/bin/nrnivmodl
 jNeuroML >>  (INFO) Found previously compiled file: /home/asinha/Documents/02_
↪Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_examples/x86_64/
↪libnrnmech.so
 jNeuroML >>  (INFO) Going to check if mods in /home/asinha/Documents/02_Code/00_
↪mine/NeuroML/documentation/source/Userdocs/NML2_examples are newer than /home/
↪asinha/Documents/02_Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_
↪examples/x86_64/libnrnmech.so
 jNeuroML >>  (INFO) Going to check if mods in /home/asinha/Documents/02_Code/00_
↪mine/NeuroML/documentation/source/Userdocs/NML2_examples are newer than /home/
↪asinha/Documents/02_Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_
↪examples/x86_64/.libs/libnrnmech.so
 jNeuroML >>  (INFO) Not being asked to recompile, and no mod files exist in /
↪home/asinha/Documents/02_Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_
↪examples which are newer than /home/asinha/Documents/02_Code/00_mine/NeuroML/
↪documentation/source/Userdocs/NML2_examples/x86_64/.libs/libnrnmech.so
 jNeuroML >>  (INFO) Success in compiling mods: true
 jNeuroML >>  (INFO) Have successfully executed command: python /home/asinha/
↪Documents/02_Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_examples/
↪LEMS_example_izhikevich2007network_sim_nrn.py
 jNeuroML >>  (INFO) NRN Output >>>
```

(continues on next page)

(continued from previous page)

```
 jNeuroML >>  (INFO) NRN Output >>>     Starting simulation in NEURON of 1000ms␣
↪generated from NeuroML2 model...
 jNeuroML >>  (INFO) NRN Output >>>
 jNeuroML >>  (INFO) NRN Output >>> Population IzPop0 contains 5 instance(s) of␣
↪component: iz2007RS0 of type: izhikevich2007Cell
 jNeuroML >>  (INFO) NRN Output >>> Population IzPop1 contains 5 instance(s) of␣
↪component: iz2007RS0 of type: izhikevich2007Cell
 jNeuroML >>  (INFO) NRN Output >>> Adding projection: proj, from IzPop0 to␣
↪IzPop1 with synapse syn0, 20 connection(s)
 jNeuroML >>  (INFO) NRN Output >>> Setting up the network to simulate took 0.
↪000990 seconds
 jNeuroML >>  (INFO) NRN Output >>> Running a simulation of 1000.0ms (dt = 0.1ms;␣
↪seed=123)
 jNeuroML >>  (INFO) NRN Output >>> Finished NEURON simulation in 0.022036␣
↪seconds (0.000367 mins)...
 jNeuroML >>  (INFO) NRN Output >>> Saving results at t=999.9999999996382...
 jNeuroML >>  (INFO) NRN Output >>> Saved data to: time.dat
 jNeuroML >>  (INFO) NRN Output >>> Saved data to: example_izhikevich2007network_
↪sim.1.spikes.dat
 jNeuroML >>  (INFO) NRN Output >>> Saved data to: example_izhikevich2007network_
↪sim.0.spikes.dat
 jNeuroML >>  (INFO) NRN Output >>> Finished saving results in 0.002883 seconds
 jNeuroML >>  (INFO) NRN Output >>> Done
 jNeuroML >>  (INFO) Exit value for running NEURON: 0
 jNeuroML >>
```

```
True
```

4.4.5 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=['.','.'],
)
```

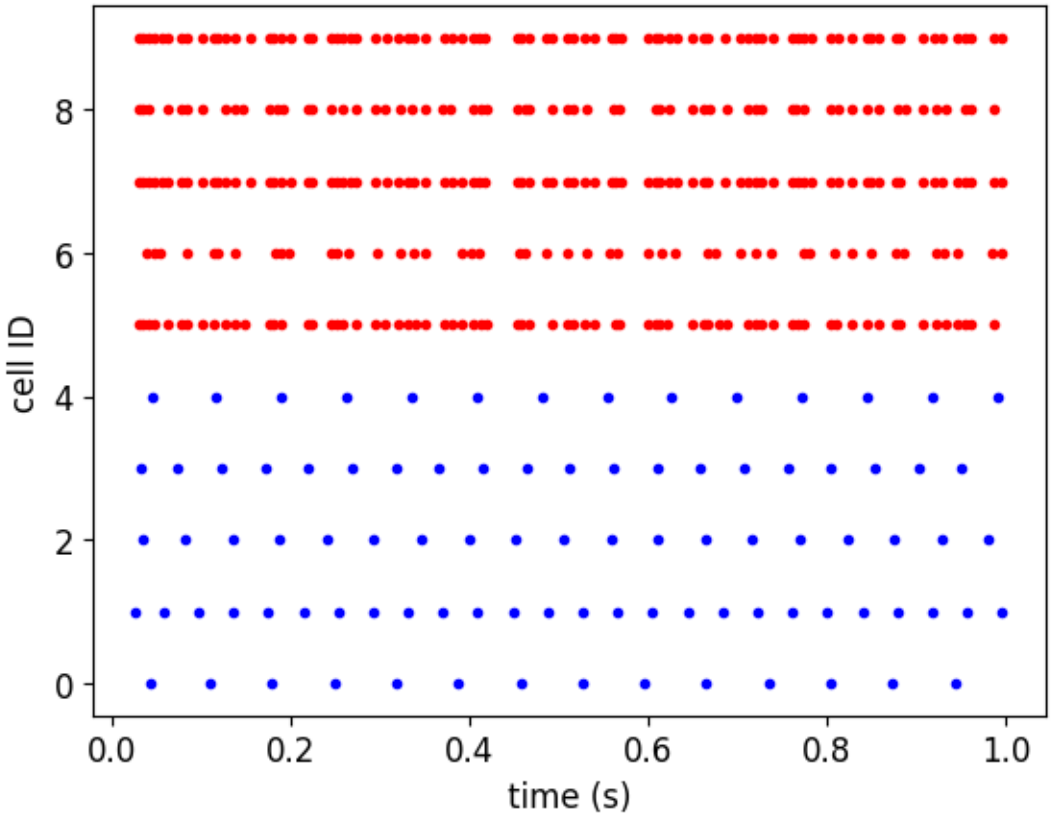
```
pyNeuroML >>> INFO - Generating plot: Spike times
/home/asinha/Documents/02_Code/00_mine/NeuroML/documentation/.venv/lib64/python3.
 ↪10/site-packages/pyneuroml/plot/Plot.py:174: UserWarning: marker is redundantly␣
 ↪defined by the 'marker' keyword argument and the fmt string "o" (-> marker='o').␣
 ↪The keyword argument will take precedence.
  plt.plot(
pyNeuroML >>> INFO - Saving image to /home/asinha/Documents/02_Code/00_mine/
```

(continues on next page)

(continued from previous page)

```
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/example_
 ↪izhikevich2007network_sim-spikes.png of plot: Spike times
pyNeuroML >>> INFO - Saved image to example_izhikevich2007network_sim-spikes.png␣
 ↪of plot: Spike times
```

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



4.5 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 2023 NeuroML contributors
Author: Ankur Sinha <sanjay DOT ankur AT gmail DOT com>
"""
```

(continues on next page)

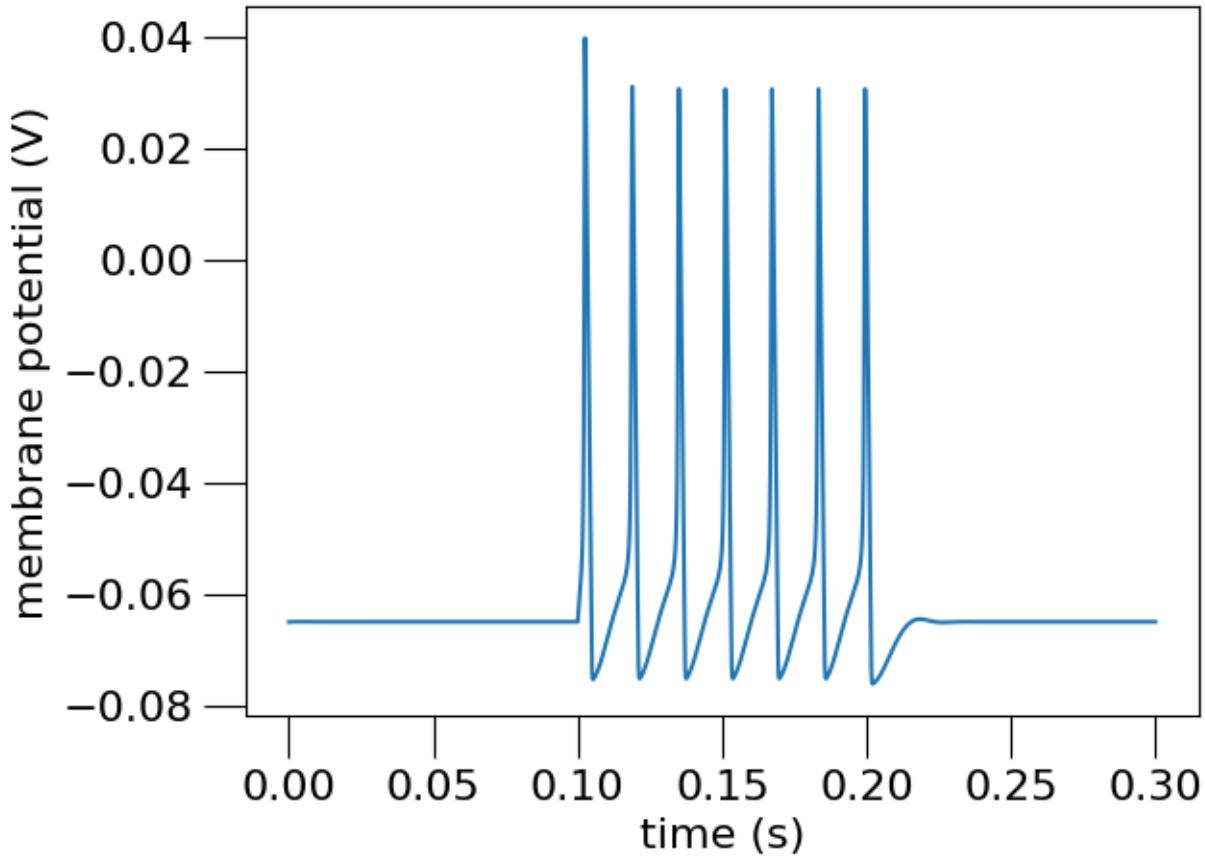


Fig. 4.6: Membrane potential of the simulated Hodgkin-Huxley neuron.

(continued from previous page)

```
import math
import neuroml
from neuroml import NeuroMLDocument
from neuroml import Network, Population
from neuroml import PulseGenerator, ExplicitInput
import numpy as np
from pyneuroml import pynml
from pyneuroml.lems import LEMSSimulation
from neuroml.utils import component_factory


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]/biophys/membraneProperties/na_channels/iDensity/",
    )
    simulation.add_column_to_output_file(
        "output0",
        column_id="pop0[0]/k/iDensity",
        quantity="pop0[0]/biophys/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):
```

(continues on next page)

(continued from previous page)

```
    """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 simulaton

    """
    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 = component_factory(
        "IonChannelHH",
        id="na_channel",
        notes="Sodium channel for HH cell",
        conductance="10pS",
        species="na",
        validate=False,
    )
    gate_m = component_factory(
        "GateHHRates",
```

(continues on next page)

(continued from previous page)

```
        id="m",
        instances="3",
        notes="m gate for na channel",
        validate=False,
    )
    m_forward_rate = component_factory(
        "HHRate", type="HHExpLinearRate", rate="1per_ms", midpoint="-40mV", scale=
↪"10mV"
    )
    m_reverse_rate = component_factory(
        "HHRate", type="HHExpRate", rate="4per_ms", midpoint="-65mV", scale="-18mV"
    )

    gate_m.add(m_forward_rate, hint="forward_rate", validate=False)
    gate_m.add(m_reverse_rate, hint="reverse_rate")
    na_channel.add(gate_m)

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

    na_channel_doc = component_factory(
        "NeuroMLDocument", id="na_channel", notes="Na channel for HH neuron"
    )
    na_channel_fn = "HH_example_na_channel.nml"
    na_channel_doc.add(na_channel)
    na_channel_doc.validate(recursive=True)

    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 = component_factory(
```

(continues on next page)

(continued from previous page)

```
        "IonChannelHH",
        id="k_channel",
        notes="Potassium channel for HH cell",
        conductance="10pS",
        species="k",
        validate=False,
    )
    gate_n = component_factory(
        "GateHHRates",
        id="n",
        instances="4",
        notes="n gate for k channel",
        validate=False,
    )
    n_forward_rate = component_factory(
        "HHRate",
        type="HHExpLinearRate",
        rate="0.1per_ms",
        midpoint="-55mV",
        scale="10mV",
    )
    n_reverse_rate = component_factory(
        "HHRate", type="HHExpRate", rate="0.125per_ms", midpoint="-65mV", scale="-80mV
↪"
    )
    gate_n.add(n_forward_rate, hint="forward_rate", validate=False)
    gate_n.add(n_reverse_rate, hint="reverse_rate")
    k_channel.add(gate_n)

    k_channel_doc = component_factory(
        "NeuroMLDocument", id="k_channel", notes="k channel for HH neuron"
    )
    k_channel_fn = "HH_example_k_channel.nml"
    k_channel_doc.add(k_channel)
    k_channel_doc.validate(recursive=True)

    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 = component_factory(
        "IonChannelHH", id="leak_channel", conductance="10pS", notes="Leak conductance
↪"
    )
    leak_channel_doc = component_factory(
        "NeuroMLDocument", id="leak_channel", notes="leak channel for HH neuron"
```

(continues on next page)

(continued from previous page)

```
    )
    leak_channel_fn = "HH_example_leak_channel.nml"
    leak_channel_doc.add(leak_channel)
    leak_channel_doc.validate(recursive=True)

    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"

    # Define a cell
    hh_cell = hh_cell_doc.add(
        "Cell", id="hh_cell", notes="A single compartment HH cell"
    )  # type: neuroml.Cell
    hh_cell.info(show_contents=True)

    # Channel density for Na channel
    hh_cell.add_channel_density(
        hh_cell_doc,
        cd_id="na_channels",
        cond_density="120.0 mS_per_cm2",
        erev="50.0 mV",
        ion="na",
        ion_channel="na_channel",
        ion_chan_def_file=create_na_channel(),
    )

    # Channel density for k channel
    hh_cell.add_channel_density(
        hh_cell_doc,
        cd_id="k_channels",
        cond_density="360 S_per_m2",
        erev="-77mV",
        ion="k",
        ion_channel="k_channel",
        ion_chan_def_file=create_k_channel(),
    )
    # Leak channel
    hh_cell.add_channel_density(
        hh_cell_doc,
        cd_id="leak_channels",
        cond_density="3.0 S_per_m2",
        erev="-54.3mV",
        ion="non_specific",
        ion_channel="leak_channel",
        ion_chan_def_file=create_leak_channel(),
```

(continues on next page)

(continued from previous page)

```
    )

    # Other membrane properties
    hh_cell.add_membrane_property("SpikeThresh", value="-20mV")
    hh_cell.set_specific_capacitance("1.0 uF_per_cm2")
    hh_cell.set_init_memb_potential("-65mV")

    hh_cell.set_resistivity("0.03 kohm_cm")

    # 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)
    hh_cell.add_segment(
        prox=[0, 0, 0, diam],
        dist=[0, 0, 0, diam],
        name="soma",
        parent=None,
        fraction_along=1.0,
        group="soma_0",
    )

    hh_cell_doc.validate(recursive=True)
    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 = component_factory(
        "NeuroMLDocument", id="network", notes="HH cell network"
    )
    net_doc_fn = "HH_example_net.nml"
    net_doc.add("IncludeType", href=create_cell())
    net = net_doc.add("Network", id="single_hh_cell_network", validate=False)

    # Create a population: convenient to create many cells of the same type
    pop = net.add(
        "Population",
        id="pop0",
        notes="A population for our cell",
        component="hh_cell",
        size=1,
    )

    # Input
    pulsegen = net_doc.add(
        "PulseGenerator",
        id="pg",
        notes="Simple pulse generator",
        delay="100ms",
        duration="100ms",
        amplitude="0.08nA",
```

(continues on next page)

(continued from previous page)

```
    )

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

    net_doc.validate(recursive=True)

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


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

4.5.1 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 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 = component_factory(
        "IonChannelHH",
        id="na_channel",
        notes="Sodium channel for HH cell",
        conductance="10pS",
        species="na",
```

(continues on next page)

(continued from previous page)

```
        validate=False,
    )
    gate_m = component_factory(
        "GateHHRates",
        id="m",
        instances="3",
        notes="m gate for na channel",
        validate=False,
    )
    m_forward_rate = component_factory(
        "HHRate", type="HHExpLinearRate", rate="1per_ms", midpoint="-40mV", scale=
↪"10mV"
    )
    m_reverse_rate = component_factory(
        "HHRate", type="HHExpRate", rate="4per_ms", midpoint="-65mV", scale="-18mV"
    )

    gate_m.add(m_forward_rate, hint="forward_rate", validate=False)
    gate_m.add(m_reverse_rate, hint="reverse_rate")
    na_channel.add(gate_m)

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

    na_channel_doc = component_factory(
        "NeuroMLDocument", id="na_channel", notes="Na channel for HH neuron"
    )
    na_channel_fn = "HH_example_na_channel.nml"
    na_channel_doc.add(na_channel)
    na_channel_doc.validate(recursive=True)

    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/
```

(continues on next page)

(continued from previous page)

```
↪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="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="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 = component_factory(
        "IonChannelHH",
        id="k_channel",
        notes="Potassium channel for HH cell",
        conductance="10pS",
        species="k",
        validate=False,
    )
    gate_n = component_factory(
        "GateHHRates",
        id="n",
        instances="4",
        notes="n gate for k channel",
        validate=False,
    )
```

(continues on next page)

(continued from previous page)

```
    n_forward_rate = component_factory(
        "HHRate",
        type="HHExpLinearRate",
        rate="0.1per_ms",
        midpoint="-55mV",
        scale="10mV",
    )
    n_reverse_rate = component_factory(
        "HHRate", type="HHExpRate", rate="0.125per_ms", midpoint="-65mV", scale="-80mV
↪"
    )
    gate_n.add(n_forward_rate, hint="forward_rate", validate=False)
    gate_n.add(n_reverse_rate, hint="reverse_rate")
    k_channel.add(gate_n)

    k_channel_doc = component_factory(
        "NeuroMLDocument", id="k_channel", notes="k channel for HH neuron"
    )
    k_channel_fn = "HH_example_k_channel.nml"
    k_channel_doc.add(k_channel)
    k_channel_doc.validate(recursive=True)

    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 = component_factory(
        "IonChannelHH", id="leak_channel", conductance="10pS", notes="Leak conductance
↪"
    )
    leak_channel_doc = component_factory(
        "NeuroMLDocument", id="leak_channel", notes="leak channel for HH neuron"
    )
    leak_channel_fn = "HH_example_leak_channel.nml"
    leak_channel_doc.add(leak_channel)
    leak_channel_doc.validate(recursive=True)

    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">
    <notes>k channel for HH neuron</notes>
    <ionChannelHH id="k_channel" species="k" conductance="10pS">
        <notes>Potassium channel for HH cell</notes>
        <gateHHrates id="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"

    # Define a cell
    hh_cell = hh_cell_doc.add(
        "Cell", id="hh_cell", notes="A single compartment HH cell"
    )  # type: neuroml.Cell
    hh_cell.info(show_contents=True)

    # Channel density for Na channel
    hh_cell.add_channel_density(
        hh_cell_doc,
        cd_id="na_channels",
        cond_density="120.0 mS_per_cm2",
        erev="50.0 mV",
        ion="na",
        ion_channel="na_channel",
```

(continues on next page)

(continued from previous page)

```
        ion_chan_def_file=create_na_channel(),
    )

    # Channel density for k channel
    hh_cell.add_channel_density(
        hh_cell_doc,
        cd_id="k_channels",
        cond_density="360 S_per_m2",
        erev="-77mV",
        ion="k",
        ion_channel="k_channel",
        ion_chan_def_file=create_k_channel(),
    )
    # Leak channel
    hh_cell.add_channel_density(
        hh_cell_doc,
        cd_id="leak_channels",
        cond_density="3.0 S_per_m2",
        erev="-54.3mV",
        ion="non_specific",
        ion_channel="leak_channel",
        ion_chan_def_file=create_leak_channel(),
    )

    # Other membrane properties
    hh_cell.add_membrane_property("SpikeThresh", value="-20mV")
    hh_cell.set_specific_capacitance("1.0 uF_per_cm2")
    hh_cell.set_init_memb_potential("-65mV")

    hh_cell.set_resistivity("0.03 kohm_cm")

    # 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)
    hh_cell.add_segment(
        prox=[0, 0, 0, diam],
        dist=[0, 0, 0, diam],
        name="soma",
        parent=None,
        fraction_along=1.0,
        group="soma_0",
    )

    hh_cell_doc.validate(recursive=True)
    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:

```
    na_channel_doc.add(na_channel)
    na_channel_doc.validate(recursive=True)

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

(continues on next page)

(continued from previous page)

```
    )

    return na_channel_fn
```

We start by creating a new NeuroML document that we will use to save this cell, and adding the cell to it.

A *Cell* component has a number of child/children components that we need to now populate:

```
Cell -- Cell with  **segment** s 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.

Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
↪Userdocs/NeuroMLv2.html for more information.

Valid members for Cell are:
* morphology_attr (class: NmlId, Optional)
* biophysical_properties_attr (class: NmlId, Optional)
* morphology (class: Morphology, Optional)
        * Contents ('ids'/<objects>): 'morphology'

* neuro_lex_id (class: NeuroLexId, Optional)
* metaid (class: MetaId, Optional)
* biophysical_properties (class: BiophysicalProperties, Optional)
        * Contents ('ids'/<objects>): 'biophys'

* id (class: NmlId, Required)
        * Contents ('ids'/<objects>): hh_cell

* notes (class: xs:string, Optional)
        * Contents ('ids'/<objects>): A single compartment HH cell

* properties (class: Property, Optional)
* annotation (class: Annotation, Optional)
```

We can see that the *morphology* and *biophysical properties* components have already been initialised for us. We now need to add the required components to them.

We begin with the biophysical properties. 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. A number of helpful functions are available to us: `add_channel_density`, `add_membrane_property`, `set_specific_capacitance`, `set_init_memb_potential`: For example, for the Na channels:

```
    # Channel density for Na channel
    hh_cell.add_channel_density(
        hh_cell_doc,
        cd_id="na_channels",
        cond_density="120.0 mS_per_cm2",
        erev="50.0 mV",
        ion="na",
        ion_channel="na_channel",
        ion_chan_def_file=create_na_channel(),
    )
```

and similarly for the K and leak channels. Now, since the ion-channels were created in other files, we need to make this document aware of their declarations. To do this, reference the other files in the `ion_chan_def_file` argument of the `add_channel_density` method. Under the hood, this will `include` the ion channel definition file we have created in this cell document using an `IncludeType` component. Each document we want to include gets appended to the list of `includes` for the document.

Next, we add the other child and children elements: the *Specific Capacitance*, the *Spike Threshold*, the *InitMembPotential*. This completes the membrane properties. We then add the intracellular properties next: *Resistivity*.

```
    # Other membrane properties
    hh_cell.add_membrane_property("SpikeThresh", value="-20mV")
    hh_cell.set_specific_capacitance("1.0 uF_per_cm2")
    hh_cell.set_init_memb_potential("-65mV")

    hh_cell.set_resistivity("0.03 kohm_cm")
```

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:

```
    diam = math.sqrt(1000 / math.pi)
    hh_cell.add_segment(
        prox=[0, 0, 0, diam],
        dist=[0, 0, 0, diam],
        name="soma",
        parent=None,
        fraction_along=1.0,
        group="soma_0",
    )
```

A *segment* has `proximal` and `distal` child elements which describe the extent of the segment. These are described using a *Point3DWithDiam* object, which the `add_segment` function creates for us.

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"␣
```

(continues on next page)

(continued from previous page)

```
 ↪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"/>
    <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 = component_factory(
        "NeuroMLDocument", id="network", notes="HH cell network"
    )
    net_doc_fn = "HH_example_net.nml"
    net_doc.add("IncludeType", href=create_cell())
    net = net_doc.add("Network", id="single_hh_cell_network", validate=False)

    # Create a population: convenient to create many cells of the same type
    pop = net.add(
        "Population",
```

(continues on next page)

(continued from previous page)

```
        id="pop0",
        notes="A population for our cell",
        component="hh_cell",
        size=1,
    )

    # Input
    pulsegen = net_doc.add(
        "PulseGenerator",
        id="pg",
        notes="Simple pulse generator",
        delay="100ms",
        duration="100ms",
        amplitude="0.08nA",
    )

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

    net_doc.validate(recursive=True)

    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>
*      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␣
↪= 3.6000000000000005e-07 S (360000.00000000006 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 =␣
↪3.0000000000000004e-09 S (3000.0000000000005 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 =␣
↪1.0000000000000001e-11 F (10.000000000000002 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:

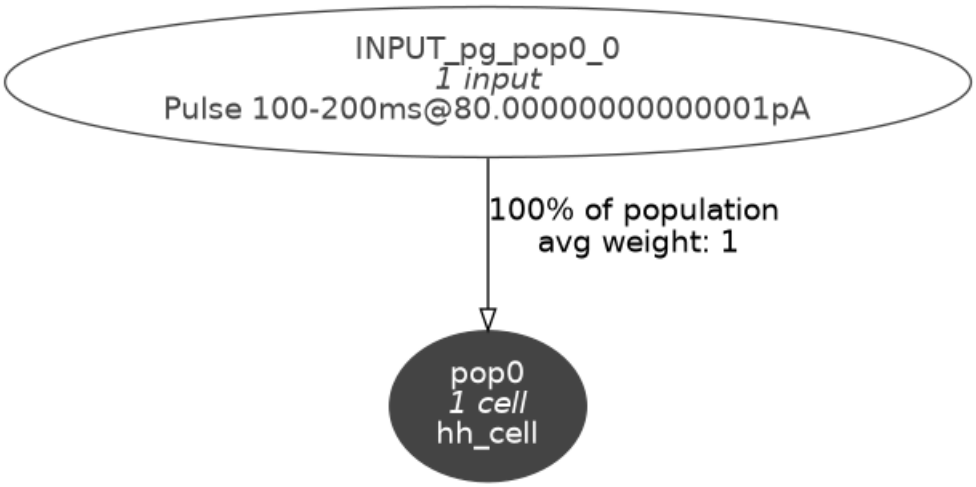


Fig. 4.7: Level 10 network graph generated by pynml

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:

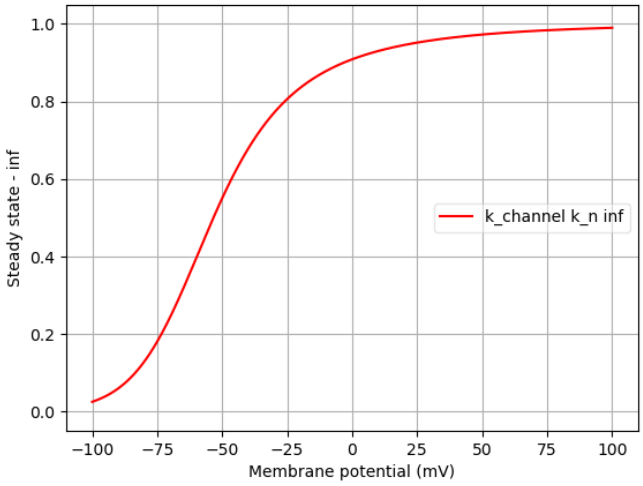


Fig. 4.8: Steady state behaviour of the K ion channel.

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

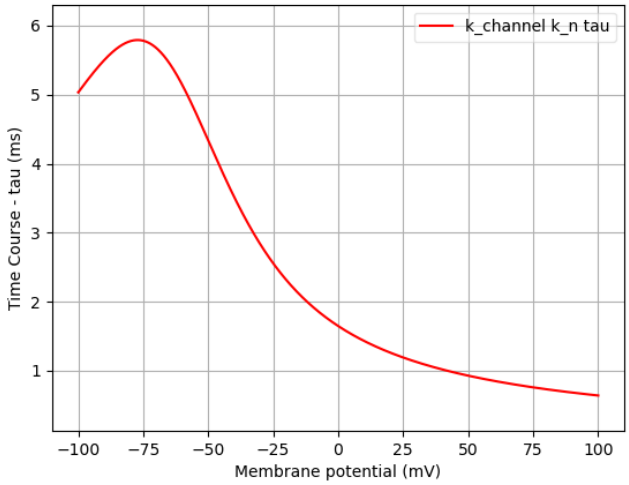


Fig. 4.9: Time course of the K ion channel.

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

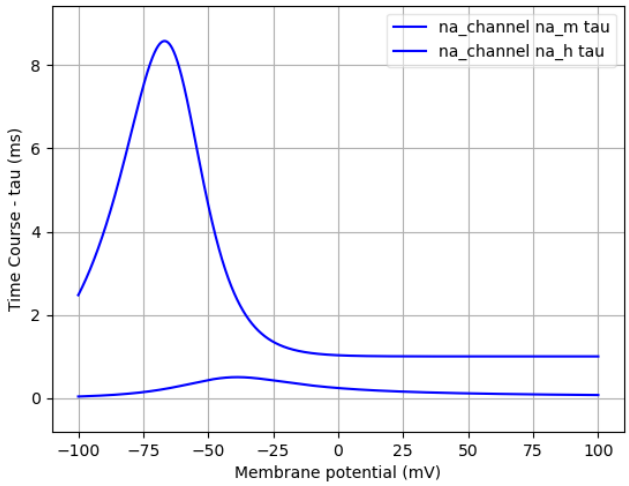


Fig. 4.10: Steady state behaviour of the Na ion channel.

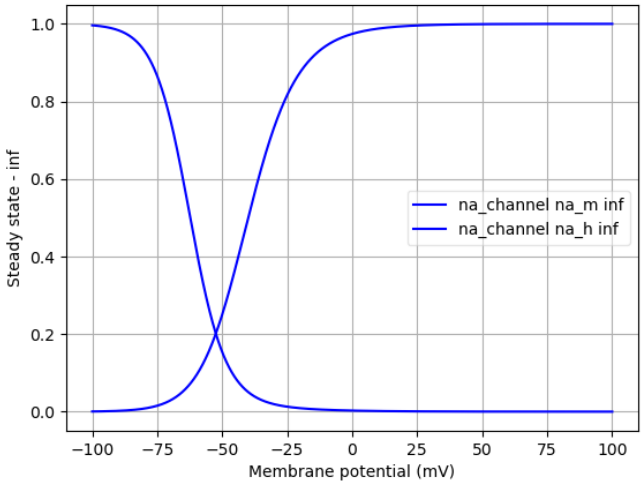


Fig. 4.11: Time course of the Na ion channel.

4.5.2 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]/biophys/membraneProperties/na_channels/iDensity/",
    )
    simulation.add_column_to_output_file(
```

(continues on next page)

(continued from previous page)

```
        "output0",
        column_id="pop0[0]/k/iDensity",
        quantity="pop0[0]/biophys/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)
```

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")
```

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]/biophys/membraneProperties/na_channels/iDensity/",
    )
    simulation.add_column_to_output_file(
        "output0",
        column_id="pop0[0]/k/iDensity",
        quantity="pop0[0]/biophys/membraneProperties/k_channels/iDensity/",
    )
```

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

4.5.3 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 simulaton

    """
    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)",
    )
```

This generates the following graphs:

This concludes out 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.

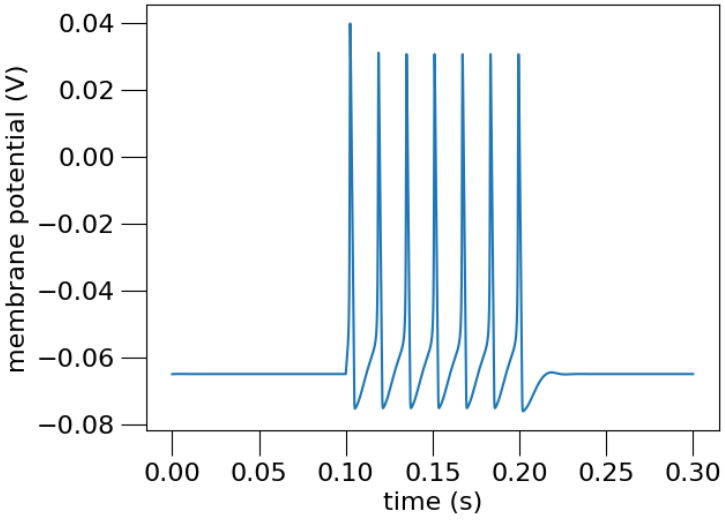


Fig. 4.12: Membrane potential

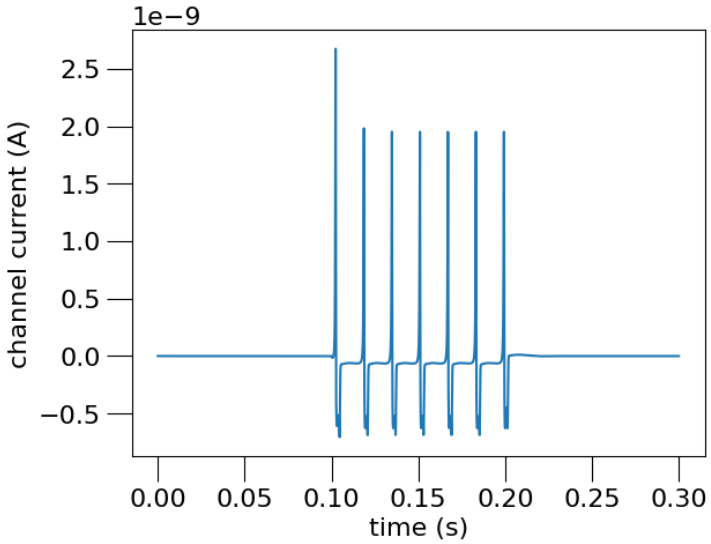


Fig. 4.13: Channel current.

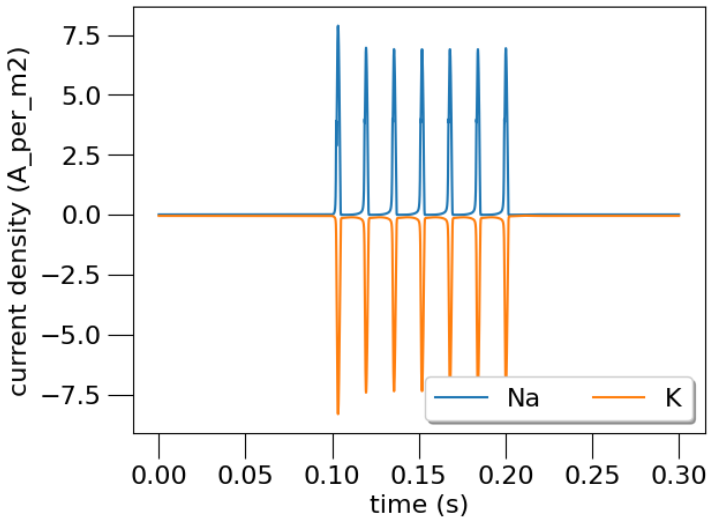


Fig. 4.14: Channel current densities

4.6 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
```

4.6.1 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="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="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="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",␣
```

(continues on next page)

(continued from previous page)

```
↪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()))
    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")
```

(continues on next page)

(continued from previous page)

```
    # 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

    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
```

4.6.2 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 simulaton

    """
    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)
↪")
```

4.6.3 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:␣
```

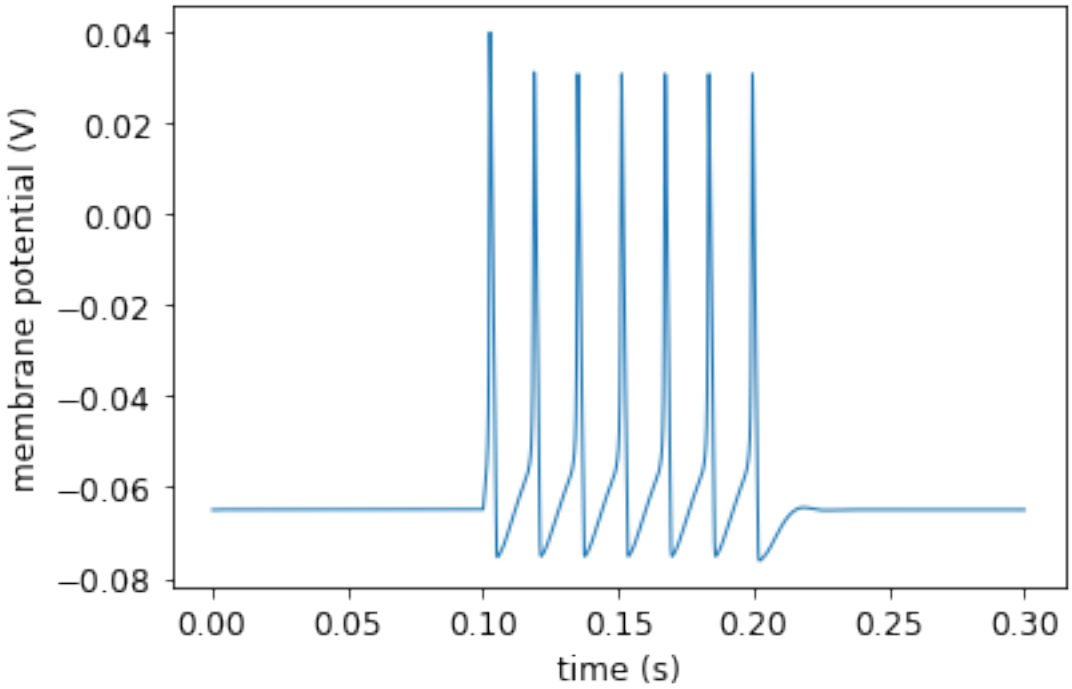
(continues on next page)

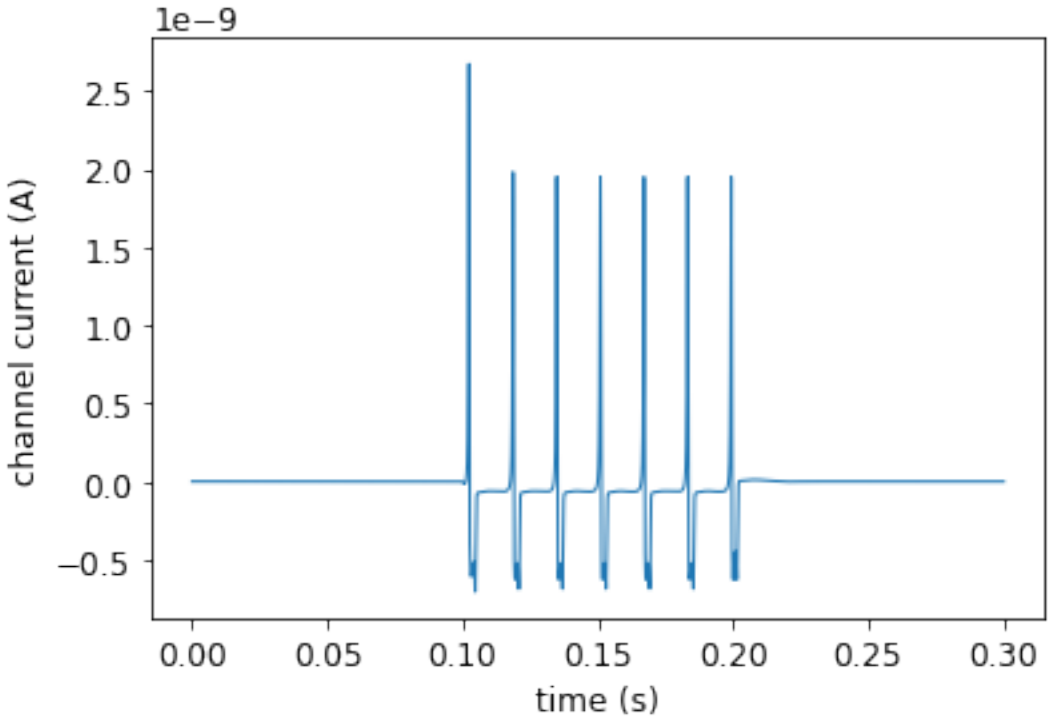
(continued from previous page)

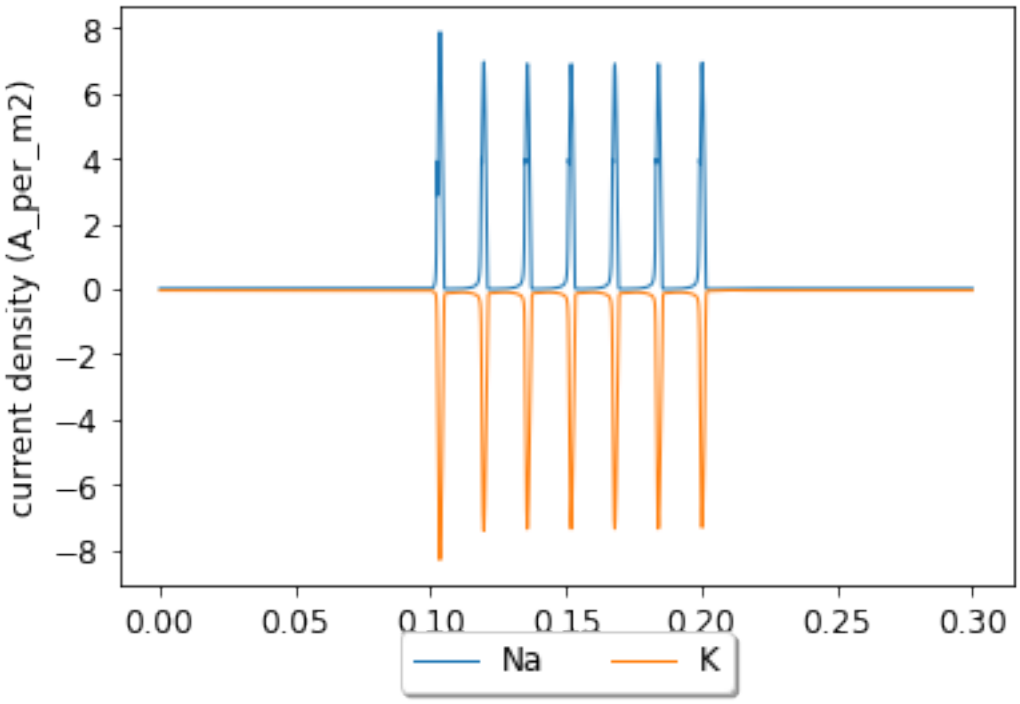
```
 ↪MatplotlibDeprecationWarning:
The set_window_title function was deprecated in Matplotlib 3.4 and will be removed␣
 ↪two minor releases later. Use manager.set_window_title or GUI-specific methods␣
 ↪instead.
  fig.canvas.set_window_title(title)
/usr/lib/python3.9/site-packages/pyneuroml/pynml.py:1727: 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)
```


0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.08
membrane potential (V)
0.00
0.05
0.10
0.15
0.20
0.25
0.30
time (s)


1e−9
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
channel current (A)
0.00
0.05
0.10
0.15
0.20
0.25
0.30
time (s)



4.7 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.

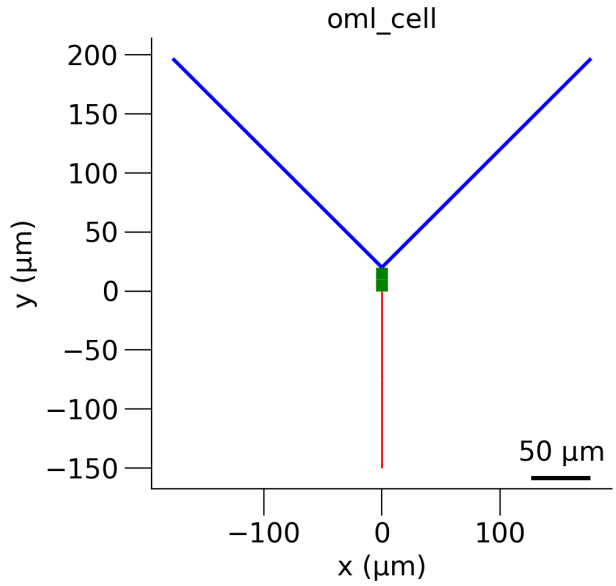


Fig. 4.15: Morphology of OLM cell in xy plane

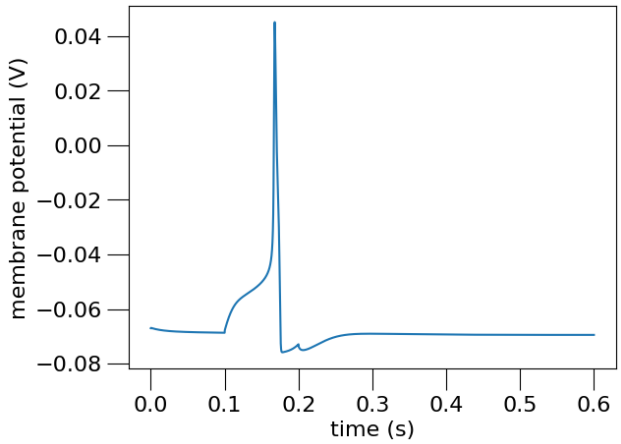


Fig. 4.16: Membrane potential of the simulated OLM cell at the soma.

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 2023 NeuroML contributors
Authors: Padraig Gleeson, Ankur Sinha
"""

import neuroml
from neuroml import NeuroMLDocument
from neuroml.utils import component_factory
from pyneuroml import pynml
from pyneuroml.lems import LEMSSimulation
from pyneuroml.plot.PlotMorphology import plot_2D
import numpy as np


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
```

(continues on next page)

(continued from previous page)

```
    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()

    # 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 simulaton

    """
    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␣
```

(continues on next page)

(continued from previous page)

```
↪(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.add("IncludeType", href=create_olm_cell())
    net = net_doc.add("Network", id="single_olm_cell_network", validate=False)
    # Create a population: convenient to create many cells of the same type
    pop = net.add("Population", id="pop0", notes="A population for our cell",
                  component="olm", size=1, type="populationList",
                  validate=False)
    pop.add("Instance", id=0, location=component_factory("Location", x=0., y=0., z=0.
↪))
    # Input
    net_doc.add("PulseGenerator", id="pg_olm", notes="Simple pulse generator", delay=
↪"100ms", duration="100ms", amplitude="0.08nA")

    net.add("ExplicitInput", target="pop0[0]", input="pg_olm")

    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 = component_factory("NeuroMLDocument", id="oml_cell")
    cell = nml_cell_doc.add("Cell", id="olm", neuro_lex_id="NLXCELL:091206")  # type␣
↪neuroml.Cell
    nml_cell_file = cell.id + ".cell.nml"

    cell.summary()
    cell.info(show_contents=True)
    cell.morphology.info(show_contents=True)

    # Add two soma segments to an unbranched segment group
    cell.add_unbranched_segment_group("soma_0")
    diam = 10.0
    soma_0 = cell.add_segment(
        prox=[0.0, 0.0, 0.0, diam],
        dist=[0.0, 10., 0.0, diam],
        name="Seg0_soma_0",
        group_id="soma_0",
        seg_type="soma"
    )

    soma_1 = cell.add_segment(
```

(continues on next page)

(continued from previous page)

```
        prox=None,
        dist=[0.0, 10. + 10., 0.0, diam],
        name="Seg1_soma_0",
        parent=soma_0,
        group_id="soma_0",
        seg_type="soma"
    )

    # Add axon segments
    diam = 1.5
    cell.add_unbranched_segments(
        [
            [0.0, 0.0, 0.0, diam],
            [0.0, -75, 0.0, diam],
            [0.0, -150, 0.0, diam],
        ],
        parent=soma_0,
        fraction_along=0.0,
        group_id="axon_0",
        seg_type="axon"
    )

    # Add 2 dendrite segments, using the branching utility function

    diam = 3.0
    cell.add_unbranched_segments(
        [
            [0.0, 20.0, 0.0, diam],
            [100, 120, 0.0, diam],
            [177, 197, 0.0, diam],
        ],
        parent=soma_1,
        fraction_along=1.0,
        group_id="dend_0",
        seg_type="dendrite"
    )

    cell.add_unbranched_segments(
        [
            [0.0, 20.0, 0.0, diam],
            [-100, 120, 0.0, diam],
            [-177, 197, 0.0, diam],
        ],
        parent=soma_1,
        fraction_along=1.0,
        group_id="dend_1",
        seg_type="dendrite"
    )

    # color groups for morphology plots
    den_seg_group = cell.get_segment_group("dendrite_group")
    den_seg_group.add("Property", tag="color", value="0.8 0 0")

    ax_seg_group = cell.get_segment_group("axon_group")
    ax_seg_group.add("Property", tag="color", value="0 0.8 0")

    soma_seg_group = cell.get_segment_group("soma_group")
```

(continues on next page)

(continued from previous page)

```
    soma_seg_group.add("Property", tag="color", value="0 0 0.8")

    # Other cell properties
    cell.set_init_memb_potential("-67mV")
    cell.set_resistivity("0.15 kohm_cm")
    cell.set_specific_capacitance("1.3 uF_per_cm2")
    cell.set_spike_thresh("-20 mV")

    # channels
    # leak
    cell.add_channel_density(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
    cell.add_channel_density(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_id="soma_group")
    # Kdrfast_soma
    cell.add_channel_density(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_id="soma_group")
    # Kdrfast_dendrite
    cell.add_channel_density(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_id="dendrite_group")
    # Kdrfast_axon
    cell.add_channel_density(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_id="axon_group")
    # KvAolm_soma
    cell.add_channel_density(nml_cell_doc,
                             cd_id="KvAolm_soma",
                             cond_density="4.95 mS_per_cm2",
```

(continues on next page)

(continued from previous page)

```
                                ion_channel="KvAolm",
                                ion_chan_def_file="olm-example/KvAolm.channel.nml",
                                erev="-77mV",
                                ion="k",
                                group_id="soma_group")
    # KvAolm_dendrite
    cell.add_channel_density(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_id="dendrite_group")
    # Nav_soma
    cell.add_channel_density(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_id="soma_group")
    # Nav_dendrite
    cell.add_channel_density(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_id="dendrite_group")
    # Nav_axon
    cell.add_channel_density(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_id="axon_group")

    cell.optimise_segment_groups()
    cell.validate(recursive=True)
    pynml.write_neuroml2_file(nml_cell_doc, nml_cell_file, True, True)
    plot_2D(nml_cell_file, plane2d="xy", nogui=True,
            save_to_file="olm.cell.xy.png")
    return nml_cell_file


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

4.7.1 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 = component_factory("NeuroMLDocument", id="oml_cell")
    cell = nml_cell_doc.add("Cell", id="olm", neuro_lex_id="NLXCELL:091206")  # type␣
↪neuroml.Cell
    nml_cell_file = cell.id + ".cell.nml"

    cell.summary()
    cell.info(show_contents=True)
    cell.morphology.info(show_contents=True)

    # Add two soma segments to an unbranched segment group
    cell.add_unbranched_segment_group("soma_0")
    diam = 10.0
    soma_0 = cell.add_segment(
        prox=[0.0, 0.0, 0.0, diam],
        dist=[0.0, 10., 0.0, diam],
        name="Seg0_soma_0",
        group_id="soma_0",
        seg_type="soma"
    )

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

    # Add axon segments
    diam = 1.5
    cell.add_unbranched_segments(
        [
            [0.0, 0.0, 0.0, diam],
            [0.0, -75, 0.0, diam],
            [0.0, -150, 0.0, diam],
        ],
```

(continues on next page)

(continued from previous page)

```
        parent=soma_0,
        fraction_along=0.0,
        group_id="axon_0",
        seg_type="axon"
    )

    # Add 2 dendrite segments, using the branching utility function

    diam = 3.0
    cell.add_unbranched_segments(
        [
            [0.0, 20.0, 0.0, diam],
            [100, 120, 0.0, diam],
            [177, 197, 0.0, diam],
        ],
        parent=soma_1,
        fraction_along=1.0,
        group_id="dend_0",
        seg_type="dendrite"
    )

    cell.add_unbranched_segments(
        [
            [0.0, 20.0, 0.0, diam],
            [-100, 120, 0.0, diam],
            [-177, 197, 0.0, diam],
        ],
        parent=soma_1,
        fraction_along=1.0,
        group_id="dend_1",
        seg_type="dendrite"
    )

    # color groups for morphology plots
    den_seg_group = cell.get_segment_group("dendrite_group")
    den_seg_group.add("Property", tag="color", value="0.8 0 0")

    ax_seg_group = cell.get_segment_group("axon_group")
    ax_seg_group.add("Property", tag="color", value="0 0.8 0")

    soma_seg_group = cell.get_segment_group("soma_group")
    soma_seg_group.add("Property", tag="color", value="0 0 0.8")

    # Other cell properties
    cell.set_init_memb_potential("-67mV")
    cell.set_resistivity("0.15 kohm_cm")
    cell.set_specific_capacitance("1.3 uF_per_cm2")
    cell.set_spike_thresh("-20 mV")

    # channels
    # leak
    cell.add_channel_density(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",
```

(continues on next page)

(continued from previous page)

```
                            ion="non_specific")
    # HCNolm_soma
    cell.add_channel_density(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_id="soma_group")
    # Kdrfast_soma
    cell.add_channel_density(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_id="soma_group")
    # Kdrfast_dendrite
    cell.add_channel_density(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_id="dendrite_group")
    # Kdrfast_axon
    cell.add_channel_density(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_id="axon_group")
    # KvAolm_soma
    cell.add_channel_density(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_id="soma_group")
    # KvAolm_dendrite
    cell.add_channel_density(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_id="dendrite_group")
    # Nav_soma
    cell.add_channel_density(nml_cell_doc,
```

(continues on next page)

(continued from previous page)

```
                              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_id="soma_group")
    # Nav_dendrite
    cell.add_channel_density(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_id="dendrite_group")
    # Nav_axon
    cell.add_channel_density(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_id="axon_group")
```

Let us walk through this function:

```
    nml_cell_doc = component_factory("NeuroMLDocument", id="oml_cell")
    cell = nml_cell_doc.add("Cell", id="olm", neuro_lex_id="NLXCELL:091206")  # type␣
↪neuroml.Cell
    nml_cell_file = cell.id + ".cell.nml"

    cell.summary()
    cell.info(show_contents=True)
    cell.morphology.info(show_contents=True)
```

We create a new model document that will hold the cell model. Then, we create and add a new *Cell* using the `add` method to the document. We also provide a `neuro_lex_id` here, which is the NeuroLex ontology identifier. This allows us to better connect models to biological concepts.

As we have seen in the *single Izhikevich neuron example*, the `add` method calls the `component_factory` to create the component object for us. For the `Cell` component type, it does a number of extra things for us to set up, or initialise, the cell.

We have a number of ways of inspecting the cell. The `summary` function provides a very short summary of the cell. This is useful to quickly get a high level overview of it:

```
>>> cell.summary()
**************************************************
* Cell: olm
* Notes: None
* Segments: 0
* SegmentGroups: 4
**************************************************
```

We can also use the general *info function* to inspect the cell:

```
>>> cell.info(show_contents=True)
Cell -- Cell with  **segment** s 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.

Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
 ↪Userdocs/NeuroMLv2.html for more information.

Valid members for Cell are:
* biophysical_properties_attr (class: NmlId, Optional)
* morphology (class: Morphology, Optional)
        * Contents ('ids'/<objects>): 'morphology'

* neuro_lex_id (class: NeuroLexId, Optional)
        * Contents ('ids'/<objects>): NLXCELL:091206

* metaid (class: MetaId, Optional)
* biophysical_properties (class: BiophysicalProperties, Optional)
        * Contents ('ids'/<objects>): 'biophys'

* id (class: NmlId, Required)
        * Contents ('ids'/<objects>): olm

* notes (class: xs:string, Optional)
* properties (class: Property, Optional)
* annotation (class: Annotation, Optional)
* morphology_attr (class: NmlId, Optional)
```

We see the cell already contains `biophysical_properties` or `morphology`. Because these are components of the cell that are expected to be used, these were added automatically for us when the new component was created.

Let us take a look at the morphology of the cell:

```
>>> cell.morphology.info(show_contents=True)
Morphology -- The collection of  **segment** s which specify the 3D structure of the␣
 ↪cell, along with a number of  **segmentGroup** s

Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
 ↪Userdocs/NeuroMLv2.html for more information.

Valid members for Morphology are:
* segments (class: Segment, Required)
* metaid (class: MetaId, Optional)
* segment_groups (class: SegmentGroup, Optional)
        * Contents ('ids'/<objects>): ['all', 'soma_group', 'axon_group', 'dendrite_
 ↪group']

* id (class: NmlId, Required)
        * Contents ('ids'/<objects>): morphology

* notes (class: xs:string, Optional)
* properties (class: Property, Optional)
* annotation (class: Annotation, Optional)
```

We see that there are no segments in the cell because we have not added any. However, there are already a number of "default" segment groups that were automatically added for us: `all`, `soma_group`, `axon_group`, `dendrite_group`.

These groups allow us to keep track of all the segments, and of the segments forming the soma, the axon, and the dendrites of the cell respectively. Take a look at the *conventions page* for more information on these.

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*. We can add segments using the `add_segment` utility function, as we do for the segments forming the soma. Here, our soma has two segments.

```
# Add two soma segments to an unbranched segment group
cell.add_unbranched_segment_group("soma_0")
diam = 10.0
soma_0 = cell.add_segment(
    prox=[0.0, 0.0, 0.0, diam],
    dist=[0.0, 10., 0.0, diam],
    name="Seg0_soma_0",
    group_id="soma_0",
    seg_type="soma"
)

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

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 makes the first segment the *parent* of the second, to build a chain. 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.

Note that by default, the `add_segment` function does not know if the segments are contiguous, i.e., that they form an unbranched branch of the cell. We could have added segments here that do not line up in a chain, when building different parts of a cell for example. In this case, we know that the two soma segments must be contiguous, and that they are on the same unbranched branch (i.e. a continuous section without any branching points on it), so we create an unbranched segment group first using the `add_unbranched_segment_group`.

If we were only creating cell morphologies, this would not not matter much. Even if the two segments were not included in a group of unbranched segments, they would still be connected. However, for simulation, simulators such as NEURON need to know which parts of the cell form unbranched sections so that they can apply the cable equation and break them into smaller segments to simulate the electric current through them. (See [CGH+07] for more information on how different simulators simulate cells with detailed morphologies.)

Next, we can call the same functions multiple times to add soma, dendritic, and axonal segments to our cell but this can get quite lengthy. To easily add unbranched contiguous lists of segments to the cell, we can directly use the `add_unbranched_segments` utility function. Here we use it to create an axonal segment group, and two dendritic groups each with two segments. The first point we provide is the proximal (starting) of the dendrite. The next two points are the distal (ends) of each segment forming the section.

```
# Add axon segments
diam = 1.5
cell.add_unbranched_segments(
    [
        [0.0, 0.0, 0.0, diam],
        [0.0, -75, 0.0, diam],
        [0.0, -150, 0.0, diam],
```

(continues on next page)

(continued from previous page)

```
        ],
        parent=soma_0,
        fraction_along=0.0,
        group_id="axon_0",
        seg_type="axon"
    )

    # Add 2 dendrite segments, using the branching utility function

    diam = 3.0
    cell.add_unbranched_segments(
        [
            [0.0, 20.0, 0.0, diam],
            [100, 120, 0.0, diam],
            [177, 197, 0.0, diam],
        ],
        parent=soma_1,
        fraction_along=1.0,
        group_id="dend_0",
        seg_type="dendrite"
    )

    cell.add_unbranched_segments(
        [
            [0.0, 20.0, 0.0, diam],
            [-100, 120, 0.0, diam],
            [-177, 197, 0.0, diam],
        ],
        parent=soma_1,
        fraction_along=1.0,
        group_id="dend_1",
        seg_type="dendrite"
    )
```

We repeat this process to create more dendritic and axonal sections of contiguous segments.

Finally, we add an extra colour property to the three primary segment groups that can be used when generating morphology graphs:

```
    # color groups for morphology plots
    den_seg_group = cell.get_segment_group("dendrite_group")
    den_seg_group.add("Property", tag="color", value="0.8 0 0")

    ax_seg_group = cell.get_segment_group("axon_group")
    ax_seg_group.add("Property", tag="color", value="0 0.8 0")

    soma_seg_group = cell.get_segment_group("soma_group")
    soma_seg_group.add("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*, e.g.:

- the *membrane properties*
  - *spike threshold*
  - *initial membrane potential*
  - *channel densities*

  - *specifc capacitances*
- the *intracellular properties*
  - *resistivity*

We use more helpful utility functions to set these values

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

For setting channel densities, we have the `add_channel_density` function:

```
    # channels
    # leak
    cell.add_channel_density(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
    cell.add_channel_density(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_id="soma_group")
    # Kdrfast_soma
    cell.add_channel_density(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_id="soma_group")
    # Kdrfast_dendrite
    cell.add_channel_density(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_id="dendrite_group")
    # Kdrfast_axon
    cell.add_channel_density(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",
```

(continues on next page)

(continued from previous page)

```
                            ion="k",
                            group_id="axon_group")
    # KvAolm_soma
    cell.add_channel_density(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_id="soma_group")
    # KvAolm_dendrite
    cell.add_channel_density(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_id="dendrite_group")
    # Nav_soma
    cell.add_channel_density(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_id="soma_group")
    # Nav_dendrite
    cell.add_channel_density(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_id="dendrite_group")
    # Nav_axon
    cell.add_channel_density(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",
```

Note that we are not writing our channel files from scratch here. We are re-using already written NeuroML channel definitions by simply including their NeuroML definition files.

This completes the definition of our cell. We now run the level one validation, write it to a file while also running a complete (level one and level two) validation using pyNeuroML. We also generate the morphology plot shown on the top of this page.

```
    cell.optimise_segment_groups()
    cell.validate(recursive=True)
```

(continues on next page)

(continued from previous page)

```
    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_v2.3.1.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" neuroLexId="NLXCELL:091206">
        <morphology id="morphology">
            <segment id="0" name="Seg0_soma_0">
                <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="0.0"/>
                <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"/>
                <proximal x="0.0" y="-75.0" z="0.0" diameter="1.5"/>
                <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"/>
                <proximal x="100.0" y="120.0" z="0.0" diameter="3.0"/>
                <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"/>
                <proximal x="-100.0" y="120.0" z="0.0" diameter="3.0"/>
                <distal x="-177.0" y="197.0" z="0.0" diameter="3.0"/>
            </segment>
            <segmentGroup id="soma_0" neuroLexId="sao864921383">
                <member segment="0"/>
                <member segment="1"/>
            </segmentGroup>
```

(continues on next page)

(continued from previous page)

```
            <segmentGroup id="axon_0" neuroLexId="sao864921383">
                <member segment="2"/>
                <member segment="3"/>
            </segmentGroup>
            <segmentGroup id="dend_0" neuroLexId="sao864921383">
                <member segment="4"/>
                <member segment="5"/>
            </segmentGroup>
            <segmentGroup id="dend_1" neuroLexId="sao864921383">
                <member segment="6"/>
                <member segment="7"/>
            </segmentGroup>
            <segmentGroup id="soma_group" neuroLexId="GO:0043025">
                <notes>Default soma segment group for the cell</notes>
                <property tag="color" value="0 0 0.8"/>
                <include segmentGroup="soma_0"/>
            </segmentGroup>
            <segmentGroup id="axon_group" neuroLexId="GO:0030424">
                <notes>Default axon segment group for the cell</notes>
                <property tag="color" value="0 0.8 0"/>
                <include segmentGroup="axon_0"/>
            </segmentGroup>
            <segmentGroup id="dendrite_group" neuroLexId="GO:0030425">
                <notes>Default dendrite segment group for the cell</notes>
                <property tag="color" value="0.8 0 0"/>
                <include segmentGroup="dend_0"/>
                <include segmentGroup="dend_1"/>
            </segmentGroup>
            <segmentGroup id="all">
                <notes>Default segment group for all segments in the cell</notes>
                <include segmentGroup="axon_0"/>
                <include segmentGroup="dend_0"/>
                <include segmentGroup="dend_1"/>
                <include segmentGroup="soma_0"/>
            </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_
```

(continues on next page)

(continued from previous page)

```
↪per_cm2" erev="50mV" segmentGroup="axon_group" ion="na"/>
                <spikeThresh value="-20 mV"/>
                <specificCapacitance value="1.3 uF_per_cm2"/>
                <initMembPotential value="-67mV"/>
            </membraneProperties>
            <intracellularProperties>
                <resistivity value="0.15 kohm_cm"/>
            </intracellularProperties>
        </biophysicalProperties>
    </cell>
</neuroml>
```

We can now already inspect our cell using the NeuroML tools. We have already generated the morphology plot in our script, but we can also do it using `pynml`:

```
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, similar to the one we've already generated:

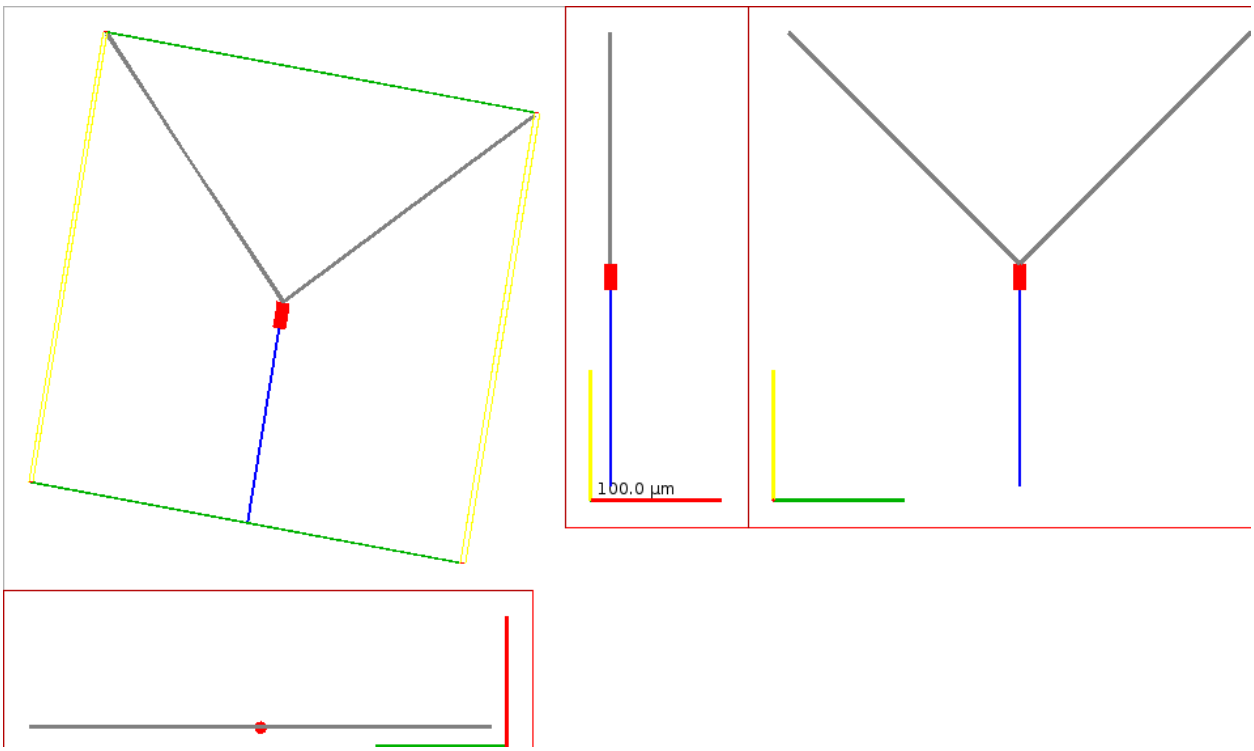


Fig. 4.17: Figure showing the morphology of the OLM cell generated from the NeuroML definition.

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

    net.add("ExplicitInput", target="pop0[0]", input="pg_olm")

    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_v2.3.1.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="0">
                <location x="0.0" y="0.0" z="0.0"/>
            </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
*    Total length of 8 segments: 670.6316010800756 um; total area: 6053.518558099847␣
```

(continues on next page)

(continued from previous page)

```
↪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: HCNolm_soma on soma_group;        conductance of 0.5 mS_per_cm2␣
↪through ion chan HCNolm 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)
*      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)
```

(continues on next page)

(continued from previous page)

```
*      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 = 3.1189038225871 73e-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 = 2.4015559433921 23e-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 = 3.1189038225871 73e-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 = 2.4015559433921 23e-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.
↪013000000000000001 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.
↪013000000000000001 F_per_m2 x 3.1415926535897934e-10 m2 = 4.084070449666732e-12 F␣
↪(4.084070449666732 pF)
*      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)
```

(continues on next page)

(continued from previous page)

```
*      Capacitance of <Segment|3|Seg1_axon_0>,  total capacitance: 0.
↪010000000000000001 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.
↪010000000000000001 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.
↪010000000000000001 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.
↪010000000000000001 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.
↪010000000000000001 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:

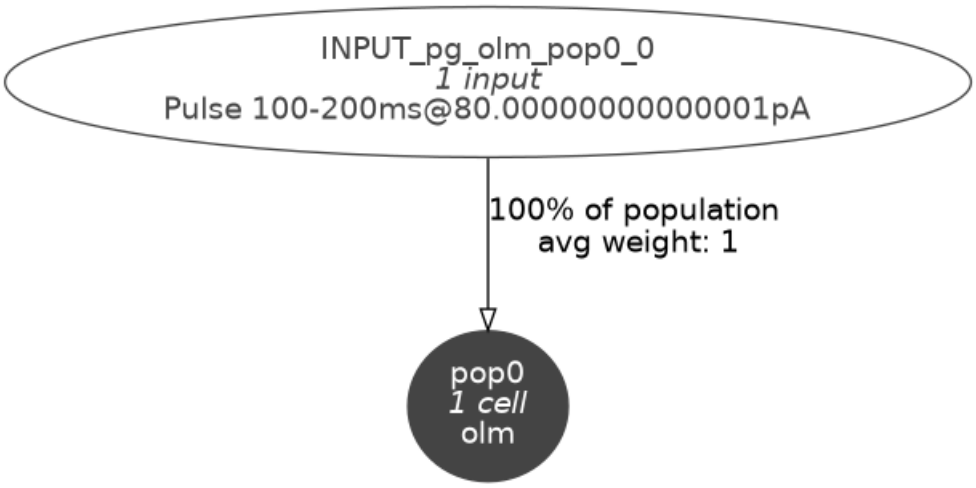


Fig. 4.18: Level 10 network graph generated by pynml

4.7.2 Simulating the model

Please note that this example uses the *NEURON* simulator to simulate the model. Please ensure that the `NEURON_HOME` environment variable is correctly set as noted *here*.

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)
    # 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()

    # 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
```

(continues on next page)

(continued from previous page)

```
    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`:

```
    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.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`). 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).

```
    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)
```

4.7.3 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 simulaton

    """
    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.

4.8 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 2023 NeuroML contributors
Authors: Padraig Gleeson, Ankur Sinha
"""

import neuroml
from neuroml import NeuroMLDocument
from neuroml.utils import component_factory
from pyneuroml import pynml
from pyneuroml.lems import LEMSSimulation
```

(continues on next page)

(continued from previous page)

```
from pyneuroml.plot.PlotMorphology import plot_2D
import numpy as np
```

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.

4.8.1 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 = component_factory("NeuroMLDocument", id="oml_cell")
    cell = nml_cell_doc.add("Cell", id="olm", neuro_lex_id="NLXCELL:091206")  # type␣
↪neuroml.Cell
    nml_cell_file = cell.id + ".cell.nml"

    cell.summary()
    cell.info(show_contents=True)
    cell.morphology.info(show_contents=True)

    # Add two soma segments to an unbranched segment group
    cell.add_unbranched_segment_group("soma_0")
    diam = 10.0
    soma_0 = cell.add_segment(
        prox=[0.0, 0.0, 0.0, diam],
        dist=[0.0, 10., 0.0, diam],
        name="Seg0_soma_0",
        group_id="soma_0",
        seg_type="soma"
    )

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

    # Add axon segments
    diam = 1.5
    cell.add_unbranched_segments(
        [
            [0.0, 0.0, 0.0, diam],
            [0.0, -75, 0.0, diam],
            [0.0, -150, 0.0, diam],
        ],
        parent=soma_0,
```

(continues on next page)

(continued from previous page)

```
        fraction_along=0.0,
        group_id="axon_0",
        seg_type="axon"
    )

    # Add 2 dendrite segments, using the branching utility function

    diam = 3.0
    cell.add_unbranched_segments(
        [
            [0.0, 20.0, 0.0, diam],
            [100, 120, 0.0, diam],
            [177, 197, 0.0, diam],
        ],
        parent=soma_1,
        fraction_along=1.0,
        group_id="dend_0",
        seg_type="dendrite"
    )

    cell.add_unbranched_segments(
        [
            [0.0, 20.0, 0.0, diam],
            [-100, 120, 0.0, diam],
            [-177, 197, 0.0, diam],
        ],
        parent=soma_1,
        fraction_along=1.0,
        group_id="dend_1",
        seg_type="dendrite"
    )

    # color groups for morphology plots
    den_seg_group = cell.get_segment_group("dendrite_group")
    den_seg_group.add("Property", tag="color", value="0.8 0 0")

    ax_seg_group = cell.get_segment_group("axon_group")
    ax_seg_group.add("Property", tag="color", value="0 0.8 0")

    soma_seg_group = cell.get_segment_group("soma_group")
    soma_seg_group.add("Property", tag="color", value="0 0 0.8")

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

    # channels
    # leak
    cell.add_channel_density(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
```

(continues on next page)

(continued from previous page)

```
    cell.add_channel_density(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_id="soma_group")
    # Kdrfast_soma
    cell.add_channel_density(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_id="soma_group")
    # Kdrfast_dendrite
    cell.add_channel_density(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_id="dendrite_group")
    # Kdrfast_axon
    cell.add_channel_density(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_id="axon_group")
    # KvAolm_soma
    cell.add_channel_density(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_id="soma_group")
    # KvAolm_dendrite
    cell.add_channel_density(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_id="dendrite_group")
    # Nav_soma
    cell.add_channel_density(nml_cell_doc,
                             cd_id="Nav_soma",
                             cond_density="10.7 mS_per_cm2",
```

(continues on next page)

(continued from previous page)

```
                              ion_channel="Nav",
                              ion_chan_def_file="olm-example/Nav.channel.nml",
                              erev="50mV",
                              ion="na",
                              group_id="soma_group")
    # Nav_dendrite
    cell.add_channel_density(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_id="dendrite_group")
    # Nav_axon
    cell.add_channel_density(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_id="axon_group")

    cell.optimise_segment_groups()
    cell.validate(recursive=True)
    pynml.write_neuroml2_file(nml_cell_doc, nml_cell_file, True, True)
    plot_2D(nml_cell_file, plane2d="xy", nogui=True,
            save_to_file="olm.cell.xy.png")
    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.add("IncludeType", href=create_olm_cell())
    net = net_doc.add("Network", id="single_olm_cell_network", validate=False)
    # Create a population: convenient to create many cells of the same type
    pop = net.add("Population", id="pop0", notes="A population for our cell",
                  component="olm", size=1, type="populationList",
                  validate=False)
    pop.add("Instance", id=0, location=component_factory("Location", x=0., y=0., z=0.
↪))
    # Input
    net_doc.add("PulseGenerator", id="pg_olm", notes="Simple pulse generator", delay=
↪"100ms", duration="100ms", amplitude="0.08nA")

    net.add("ExplicitInput", target="pop0[0]", input="pg_olm")
```

(continues on next page)

(continued from previous page)

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

4.8.2 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 simulaton

    """
    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)")
```

4.8.3 Create and run the simulaton

```
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",
```

(continues on next page)

(continued from previous page)

```
                                          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()

    # 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 >>> INFO - Executing: (java -Xmx400M  -jar  "/home/asinha/Documents/02_
 ↪Code/00_mine/NeuroML/documentation/.venv/lib/python3.10/site-packages/pyneuroml/
 ↪lib/jNeuroML-0.12.0-jar-with-dependencies.jar" -validate "olm.cell.nml" ) in␣
 ↪directory: .
```

```
******************************************************
* Cell: olm
* Notes: None
* Segments: 0
* SegmentGroups: 4
******************************************************
Cell -- Cell with  **segment** s 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.

Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
 ↪Userdocs/NeuroMLv2.html for more information.

Valid members for Cell are:
* morphology_attr (class: NmlId, Optional)
```

(continues on next page)

(continued from previous page)

```
* biophysical_properties_attr (class: NmlId, Optional)
* morphology (class: Morphology, Optional)
        * Contents ('ids'/<objects>): 'morphology'

* neuro_lex_id (class: NeuroLexId, Optional)
        * Contents ('ids'/<objects>): NLXCELL:091206

* biophysical_properties (class: BiophysicalProperties, Optional)
        * Contents ('ids'/<objects>): 'biophys'

* annotation (class: Annotation, Optional)
* properties (class: Property, Optional)
* notes (class: xs:string, Optional)
* id (class: NmlId, Required)
        * Contents ('ids'/<objects>): olm

* metaid (class: MetaId, Optional)

Morphology -- The collection of  **segment** s which specify the 3D structure of␣
 ↪the cell, along with a number of  **segmentGroup** s

Please see the NeuroML standard schema documentation at https://docs.neuroml.org/
 ↪Userdocs/NeuroMLv2.html for more information.

Valid members for Morphology are:
* segments (class: Segment, Required)
* segment_groups (class: SegmentGroup, Optional)
        * Contents ('ids'/<objects>): ['all', 'soma_group', 'axon_group',
 ↪'dendrite_group']

* annotation (class: Annotation, Optional)
* properties (class: Property, Optional)
* notes (class: xs:string, Optional)
* id (class: NmlId, Required)
        * Contents ('ids'/<objects>): morphology

* metaid (class: MetaId, Optional)
```

```
pyNeuroML >>> INFO - Command completed. Output:
  jNeuroML >>   jNeuroML v0.12.0
  jNeuroML >>  Validating: /home/asinha/Documents/02_Code/00_mine/NeuroML/
 ↪documentation/source/Userdocs/NML2_examples/olm.cell.nml
  jNeuroML >>  Valid against schema and all tests
  jNeuroML >>  No warnings
  jNeuroML >>
  jNeuroML >>  Validated 1 files: All valid and no warnings
  jNeuroML >>
  jNeuroML >>
pyNeuroML >>> INFO - Loading NeuroML2 file: olm.cell.nml
pyNeuroML >>> INFO - Including included files (included already: [])
pyNeuroML >>> INFO - Loading NeuroML2 file: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm-example/leak_chan.
 ↪channel.nml
pyNeuroML >>> INFO - Including included files (included already: [])
pyNeuroML >>> INFO - Loading NeuroML2 file: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm-example/HCNolm.channel.
```

(continues on next page)

(continued from previous page)

```
 ↪nml
pyNeuroML >>> INFO - Including included files (included already: ['/home/asinha/
 ↪Documents/02_Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_examples/
 ↪olm-example/leak_chan.channel.nml'])
pyNeuroML >>> INFO - Loading NeuroML2 file: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm-example/Kdrfast.channel.
 ↪nml
pyNeuroML >>> INFO - Including included files (included already: ['/home/asinha/
 ↪Documents/02_Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_examples/
 ↪olm-example/leak_chan.channel.nml', '/home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm-example/HCNolm.channel.
 ↪nml'])
pyNeuroML >>> INFO - Loading NeuroML2 file: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm-example/KvAolm.channel.
 ↪nml
pyNeuroML >>> INFO - Including included files (included already: ['/home/asinha/
 ↪Documents/02_Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_examples/
 ↪olm-example/leak_chan.channel.nml', '/home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm-example/HCNolm.channel.
 ↪nml', '/home/asinha/Documents/02_Code/00_mine/NeuroML/documentation/source/
 ↪Userdocs/NML2_examples/olm-example/Kdrfast.channel.nml'])
pyNeuroML >>> INFO - Loading NeuroML2 file: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm-example/Nav.channel.nml
pyNeuroML >>> INFO - Including included files (included already: ['/home/asinha/
 ↪Documents/02_Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_examples/
 ↪olm-example/leak_chan.channel.nml', '/home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm-example/HCNolm.channel.
 ↪nml', '/home/asinha/Documents/02_Code/00_mine/NeuroML/documentation/source/
 ↪Userdocs/NML2_examples/olm-example/Kdrfast.channel.nml', '/home/asinha/Documents/
 ↪02_Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_examples/olm-example/
 ↪KvAolm.channel.nml'])
pyNeuroML >>> INFO - Executing: (java -Xmx400M  -jar  "/home/asinha/Documents/02_
 ↪Code/00_mine/NeuroML/documentation/.venv/lib/python3.10/site-packages/pyneuroml/
 ↪lib/jNeuroML-0.12.0-jar-with-dependencies.jar" -validate "olm_example_net.nml" )␣
 ↪in directory: .
```

```
Saved image on plane xy to /home/asinha/Documents/02_Code/00_mine/NeuroML/
 ↪documentation/source/Userdocs/NML2_examples/olm.cell.xy.png of plot: 2D plot of␣
 ↪olm from olm.cell.nml
```

```
pyNeuroML >>> INFO - Command completed. Output:
  jNeuroML >>   jNeuroML v0.12.0
  jNeuroML >>  Validating: /home/asinha/Documents/02_Code/00_mine/NeuroML/
 ↪documentation/source/Userdocs/NML2_examples/olm_example_net.nml
  jNeuroML >>  Valid against schema and all tests
  jNeuroML >>  No warnings
  jNeuroML >>
  jNeuroML >>  Validated 1 files: All valid and no warnings
  jNeuroML >>
  jNeuroML >>
pyNeuroML >>> INFO - Loading NeuroML2 file: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm_example_net.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm.cell.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm-example/leak_chan.
```

(continues on next page)

(continued from previous page)

```
 ↪channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm-example/HCNolm.channel.
 ↪nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm-example/Kdrfast.channel.
 ↪nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm-example/KvAolm.channel.
 ↪nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm-example/Nav.channel.nml
pyNeuroML >>> INFO - Loading LEMS file: LEMS_olm_example_sim.xml and running with␣
 ↪jNeuroML_NEURON
pyNeuroML >>> INFO - Executing: (java -Xmx2G  -Djava.awt.headless=true -jar  "/
 ↪home/asinha/Documents/02_Code/00_mine/NeuroML/documentation/.venv/lib/python3.10/
 ↪site-packages/pyneuroml/lib/jNeuroML-0.12.0-jar-with-dependencies.jar"  "LEMS_
 ↪olm_example_sim.xml"  -neuron -run -compile -nogui -I '') in directory: .
pyNeuroML >>> INFO - Command completed. Output:
  jNeuroML >>   jNeuroML v0.12.0
  jNeuroML >>  (INFO) Reading from: /home/asinha/Documents/02_Code/00_mine/NeuroML/
 ↪documentation/source/Userdocs/NML2_examples/LEMS_olm_example_sim.xml
  jNeuroML >>  (INFO) Creating NeuronWriter to output files to /home/asinha/
 ↪Documents/02_Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_examples
  jNeuroML >>  (INFO) Adding simulation Component(id=olm_example_sim␣
 ↪type=Simulation) of network/component: single_olm_cell_network (Type: network)
  jNeuroML >>  (INFO) Adding population: pop0
  jNeuroML >>  (INFO) -- Writing to hoc: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm.hoc
  jNeuroML >>  (INFO) -- Writing to mod: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/leak_chan.mod
  jNeuroML >>  (INFO) File /home/asinha/Documents/02_Code/00_mine/NeuroML/
 ↪documentation/source/Userdocs/NML2_examples/leak_chan.mod exists and is identical
  jNeuroML >>  (INFO) -- Writing to mod: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/HCNolm.mod
  jNeuroML >>  (INFO) File /home/asinha/Documents/02_Code/00_mine/NeuroML/
 ↪documentation/source/Userdocs/NML2_examples/HCNolm.mod exists and is identical
  jNeuroML >>  (INFO) -- Writing to mod: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/Kdrfast.mod
  jNeuroML >>  (INFO) File /home/asinha/Documents/02_Code/00_mine/NeuroML/
 ↪documentation/source/Userdocs/NML2_examples/Kdrfast.mod exists and is identical
  jNeuroML >>  (INFO) -- Writing to mod: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/KvAolm.mod
  jNeuroML >>  (INFO) File /home/asinha/Documents/02_Code/00_mine/NeuroML/
 ↪documentation/source/Userdocs/NML2_examples/KvAolm.mod exists and is identical
  jNeuroML >>  (INFO) -- Writing to mod: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/Nav.mod
  jNeuroML >>  (INFO) File /home/asinha/Documents/02_Code/00_mine/NeuroML/
 ↪documentation/source/Userdocs/NML2_examples/Nav.mod exists and is identical
  jNeuroML >>  (INFO) Adding projections/connections...
  jNeuroML >>  (INFO) -- Writing to mod: /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/pg_olm.mod
  jNeuroML >>  (INFO) File /home/asinha/Documents/02_Code/00_mine/NeuroML/
 ↪documentation/source/Userdocs/NML2_examples/pg_olm.mod exists and is identical
  jNeuroML >>  (INFO) Trying to compile mods in: /home/asinha/Documents/02_Code/00_
 ↪mine/NeuroML/documentation/source/Userdocs/NML2_examples
  jNeuroML >>  (INFO) Going to compile the mod files in: /home/asinha/Documents/02_
```

(continues on next page)

(continued from previous page)

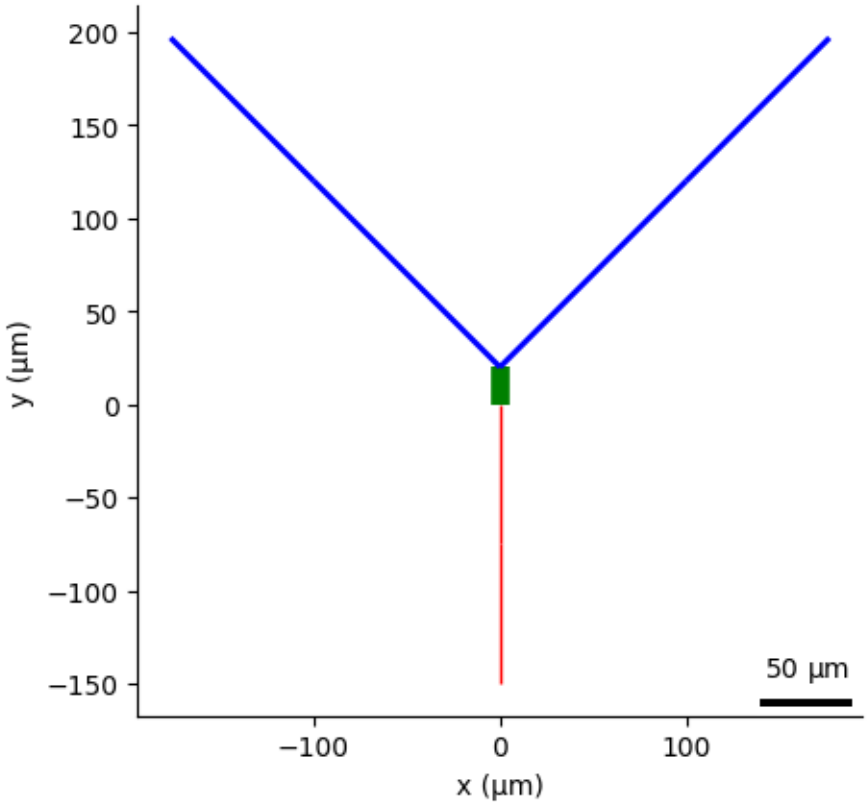
```
↪Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_examples, forcing␣
↪recompile: false
 jNeuroML >>  (INFO) Parent dir: /home/asinha/Documents/02_Code/00_mine/NeuroML/
↪documentation/source/Userdocs/NML2_examples
 jNeuroML >>  (INFO) Assuming *nix environment...
 jNeuroML >>  (INFO) Name of file to be created: /home/asinha/Documents/02_Code/
↪00_mine/NeuroML/documentation/source/Userdocs/NML2_examples/x86_64/libnrnmech.la
 jNeuroML >>  (INFO) Name of file to be created: /home/asinha/Documents/02_Code/
↪00_mine/NeuroML/documentation/source/Userdocs/NML2_examples/x86_64/libnrnmech.so
 jNeuroML >>  (INFO) Name of file to be created: /home/asinha/Documents/02_Code/
↪00_mine/NeuroML/documentation/source/Userdocs/NML2_examples/x86_64/.libs/
↪libnrnmech.so
 jNeuroML >>  (INFO) commandToExecute: /usr/bin/nrnivmodl
 jNeuroML >>  (INFO) Found previously compiled file: /home/asinha/Documents/02_
↪Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_examples/x86_64/
↪libnrnmech.so
 jNeuroML >>  (INFO) Going to check if mods in /home/asinha/Documents/02_Code/00_
↪mine/NeuroML/documentation/source/Userdocs/NML2_examples are newer than /home/
↪asinha/Documents/02_Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_
↪examples/x86_64/libnrnmech.so
 jNeuroML >>  (INFO) Going to check if mods in /home/asinha/Documents/02_Code/00_
↪mine/NeuroML/documentation/source/Userdocs/NML2_examples are newer than /home/
↪asinha/Documents/02_Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_
↪examples/x86_64/.libs/libnrnmech.so
 jNeuroML >>  (INFO) Not being asked to recompile, and no mod files exist in /
↪home/asinha/Documents/02_Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_
↪examples which are newer than /home/asinha/Documents/02_Code/00_mine/NeuroML/
↪documentation/source/Userdocs/NML2_examples/x86_64/.libs/libnrnmech.so
 jNeuroML >>  (INFO) Success in compiling mods: true
 jNeuroML >>  (INFO) Have successfully executed command: python /home/asinha/
↪Documents/02_Code/00_mine/NeuroML/documentation/source/Userdocs/NML2_examples/
↪LEMS_olm_example_sim_nrn.py
 jNeuroML >>  (INFO) NRN Output >>>
 jNeuroML >>  (INFO) NRN Output >>>     Starting simulation in NEURON of 600ms␣
↪generated from NeuroML2 model...
 jNeuroML >>  (INFO) NRN Output >>>
 jNeuroML >>  (INFO) NRN Output >>> Population pop0 contains 1 instance(s) of␣
↪component: olm of type: cell
 jNeuroML >>  (INFO) NRN Output >>>         1
 jNeuroML >>  (INFO) NRN Output >>> Setting up the network to simulate took 0.
↪006119 seconds
 jNeuroML >>  (INFO) NRN Output >>> Running a simulation of 600.0ms (dt = 0.01ms;␣
↪seed=123)
 jNeuroML >>  (INFO) NRN Output >>> Finished NEURON simulation in 0.424608␣
↪seconds (0.007077 mins)...
 jNeuroML >>  (INFO) NRN Output >>> Saving results at t=599.9999999995268...
 jNeuroML >>  (INFO) NRN Output >>> Saved data to: time.dat
 jNeuroML >>  (INFO) NRN Output >>> Saved data to: olm_example_sim.dat
 jNeuroML >>  (INFO) NRN Output >>> Finished saving results in 0.455872 seconds
 jNeuroML >>  (INFO) NRN Output >>> Done
 jNeuroML >>  (INFO) Exit value for running NEURON: 0
 jNeuroML >>
pyNeuroML >>> INFO - Generating plot: Membrane potential (soma seg 0)
/home/asinha/Documents/02_Code/00_mine/NeuroML/documentation/.venv/lib64/python3.
↪10/site-packages/pyneuroml/plot/Plot.py:186: UserWarning: marker is redundantly␣
↪defined by the 'marker' keyword argument and the fmt string "o" (-> marker='o').␣
↪The keyword argument will take precedence.
```

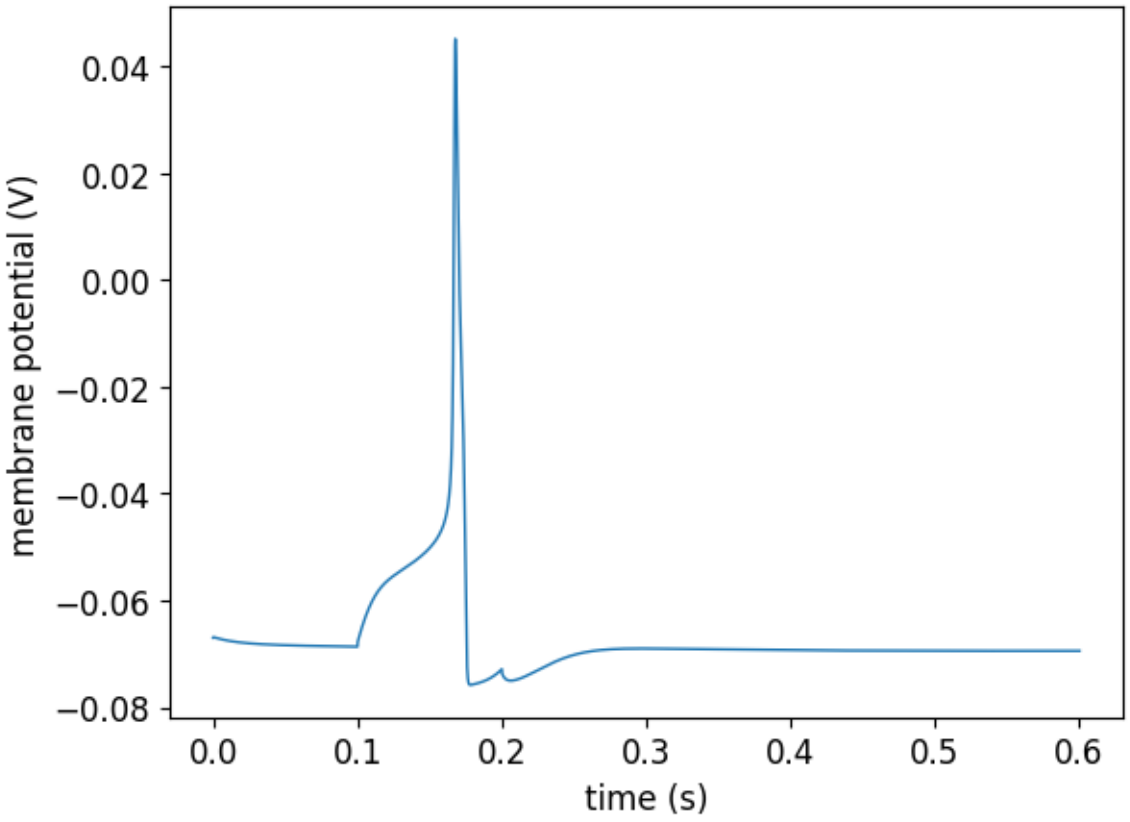
(continues on next page)

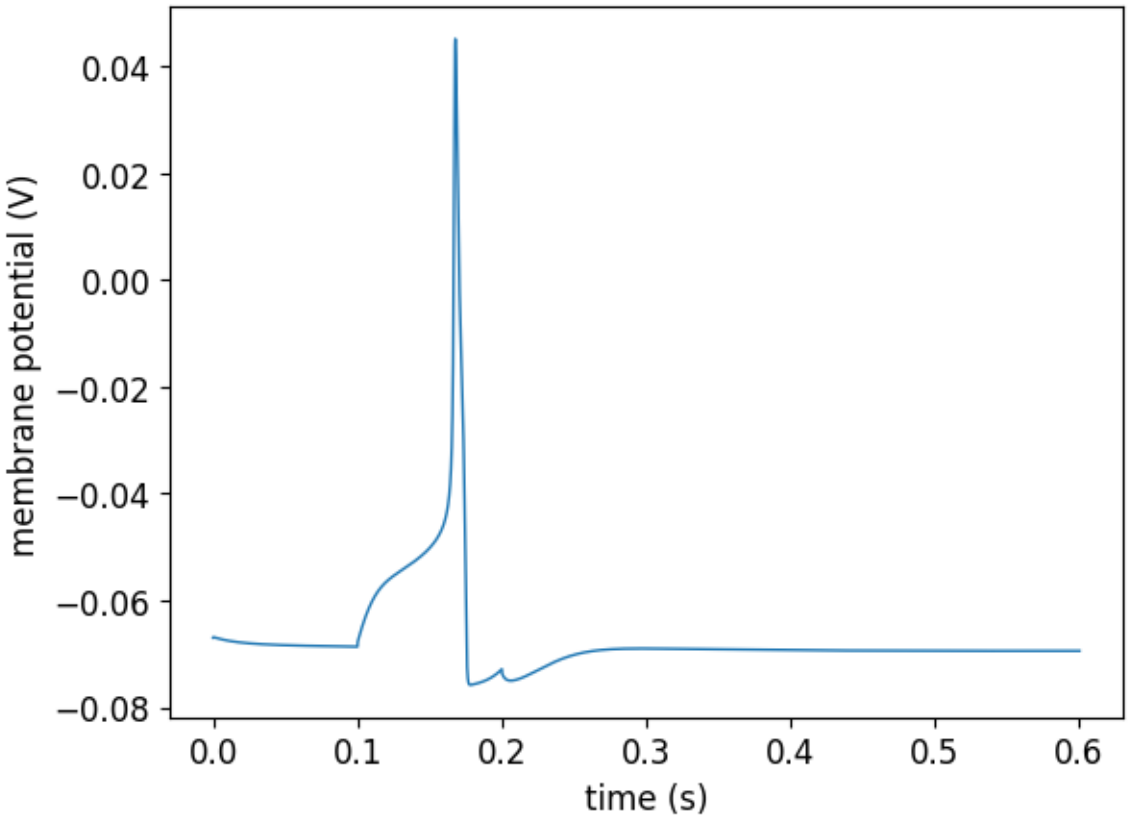
(continued from previous page)

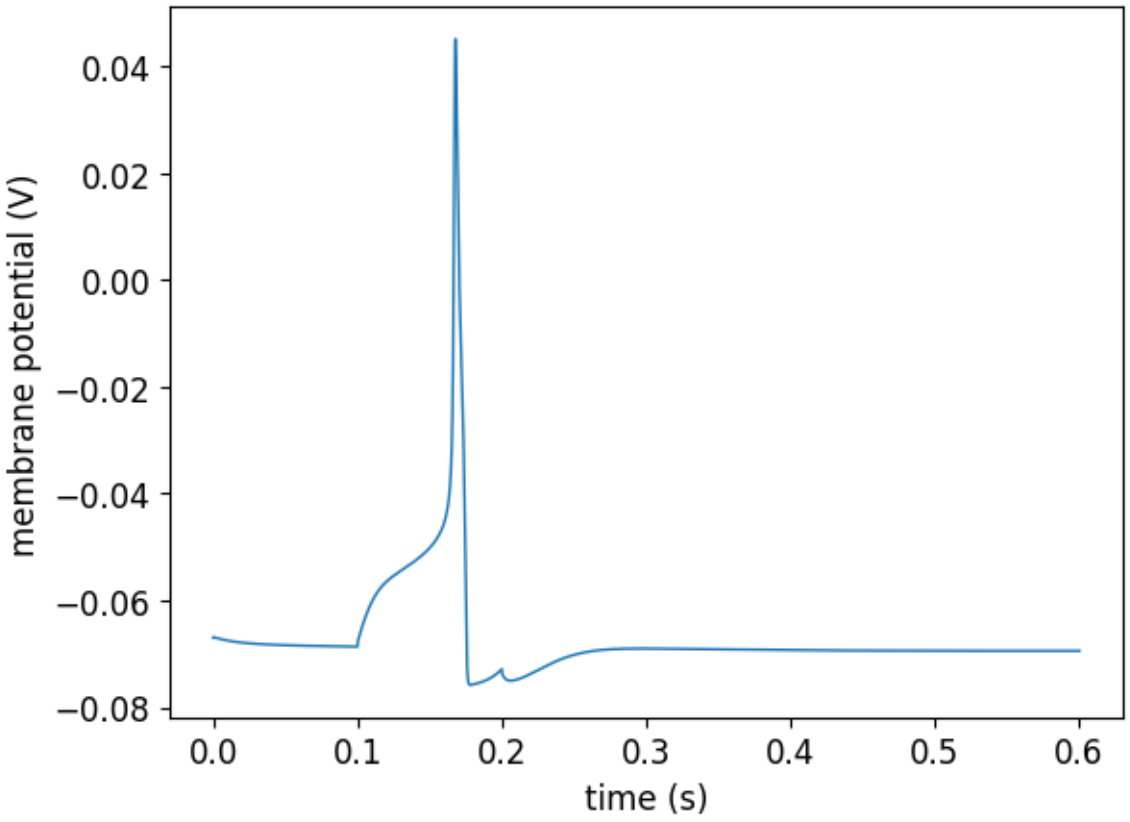
```
  plt.plot(
pyNeuroML >>> INFO - Saving image to /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm_example_sim_seg0_soma0-v.
 ↪png of plot: Membrane potential (soma seg 0)
```

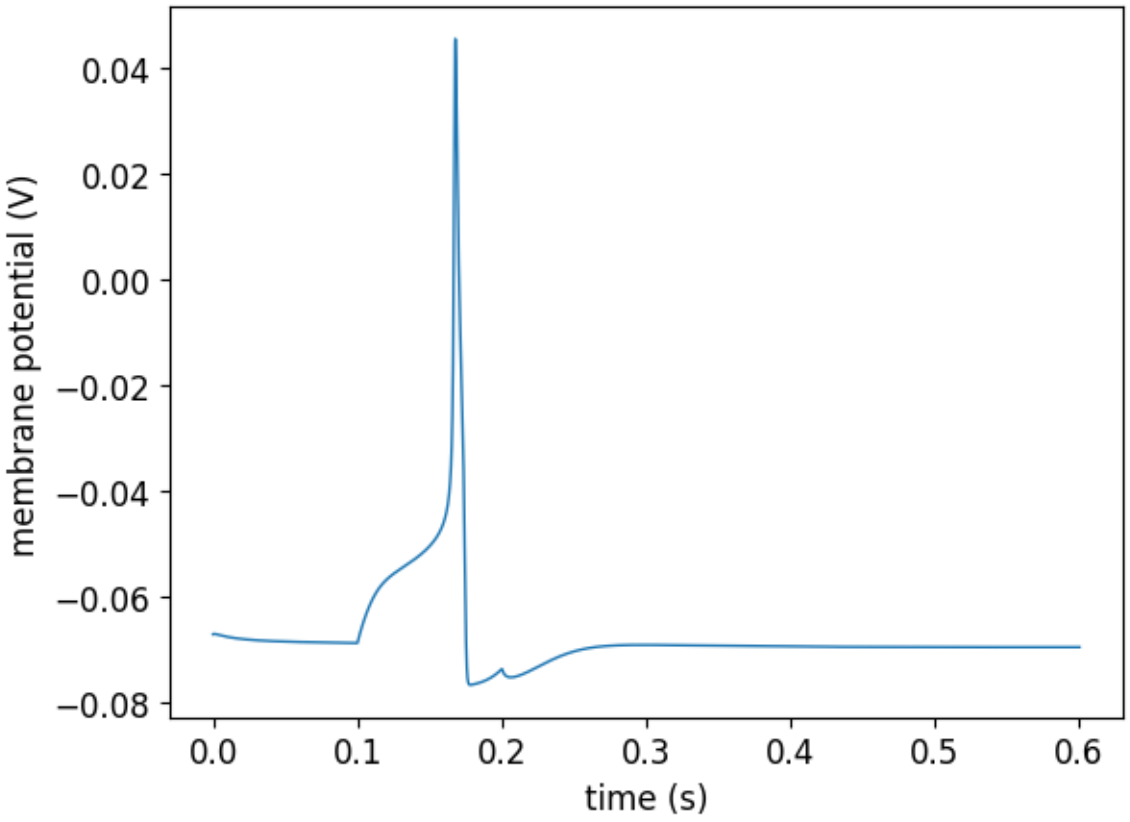
```
pyNeuroML >>> INFO - Saved image to olm_example_sim_seg0_soma0-v.png of plot:␣
 ↪Membrane potential (soma seg 0)
pyNeuroML >>> INFO - Generating plot: Membrane potential (soma seg 1)
pyNeuroML >>> INFO - Saving image to /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm_example_sim_seg1_soma0-v.
 ↪png of plot: Membrane potential (soma seg 1)
pyNeuroML >>> INFO - Saved image to olm_example_sim_seg1_soma0-v.png of plot:␣
 ↪Membrane potential (soma seg 1)
pyNeuroML >>> INFO - Generating plot: Membrane potential (axon seg 0)
pyNeuroML >>> INFO - Saving image to /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm_example_sim_seg0_axon0-v.
 ↪png of plot: Membrane potential (axon seg 0)
pyNeuroML >>> INFO - Saved image to olm_example_sim_seg0_axon0-v.png of plot:␣
 ↪Membrane potential (axon seg 0)
pyNeuroML >>> INFO - Generating plot: Membrane potential (axon seg 1)
pyNeuroML >>> INFO - Saving image to /home/asinha/Documents/02_Code/00_mine/
 ↪NeuroML/documentation/source/Userdocs/NML2_examples/olm_example_sim_seg1_axon0-v.
 ↪png of plot: Membrane potential (axon seg 1)
pyNeuroML >>> INFO - Saved image to olm_example_sim_seg1_axon0-v.png of plot:␣
 ↪Membrane potential (axon seg 1)
```




0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.08
membrane potential (V)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
time (s)


0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.08
membrane potential (V)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
time (s)


0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.08
membrane potential (V)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
time (s)


0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.08
membrane potential (V)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
time (s)

4.9 Create novel NeuroML models from components on NeuroML-DB

This notebook demonstrates how to access the NeuroML-DB database, extract elements in NeuroML format and use them to create new models

```
from pyneuroml import pynml
import urllib.request, json
import requests
import os
```

4.9.1 1) Search for, download and analyse channels accessed via the NeuroML-DB API

1.1) Search in the database for a particular type of channel

```
types = {'cell':'NMLCL','channel':'NMLCH'}

# Helper method for search
def search_neuromldb(search_term, type=None):

    with urllib.request.urlopen('https://neuroml-db.org/api/search?q=%s' % search_
↪term.replace(' ','%20')) as url:
        data = json.load(url)

    for l in data:
        if type!=None:
            for type_ in types:
                if type==type_ and not types[type_] in l['Model_ID']:
                    data.remove(l)
        if l in data:
            print('%s: %s, %s %s %s'%(l['Model_ID'],l['Name'],l['First_Author'],l[
↪'Second_Author'],l['Publication_Year']))

    return data

data = search_neuromldb("Fast Sodium", 'channel')
```

```
NMLCH000023: NaF Inactivating Fast Sodium, Maex De Schutter 1998
NMLCH001490: NaTs Fast Inactivating Sodium, Gouwens Berg 2018
NMLCH001398: NaTa Fast Inactivating Sodium, Hay Hill 2011
NMLCH000171: NaF Fast Transient Inactivating Sodium, Traub Contreras 2005
NMLCH000170: NaF Fast Transient Inactivating Sodium, Traub Contreras 2005
NMLCH000169: NaF Fast Transient Inactivating Sodium, Traub Contreras 2005
NMLCH000168: NaF Fast Transient Inactivating Sodium, Traub Contreras 2005
NMLCH000167: NaF Fast Transient Inactivating Sodium, Traub Contreras 2005
NMLCH000166: NaF Fast Transient Inactivating Sodium, Traub Contreras 2005
NMLCH000165: NaF Fast Transient Inactivating Sodium, Traub Contreras 2005
NMLCH000164: NaP Persistent Noninactivating Sodium, Traub Buhl 2003
NMLCH000163: NaF Fast Transient Inactivating Sodium, Traub Contreras 2005
NMLCH000162: NaF Fast Transient Inactivating Sodium, Traub Contreras 2005
NMLCH000161: NaF Fast Transient Inactivating Sodium, Traub Contreras 2005
NMLCH000160: NaF Fast Transient Inactivating Sodium, Traub Buhl 2003
NMLCH000159: NaF Fast Transient Inactivating Sodium, Traub Contreras 2005
NMLCH000158: NaF Fast Transient Inactivating Sodium, Traub Contreras 2005
```

(continues on next page)

(continued from previous page)

```
NMLCH000157: NaF Fast Transient Inactivating Sodium, Traub Buhl 2003
NMLCH000131: NaF Fast Sodium, Pospischil Toledo-Rodriguez 2008
NMLCH000111: NaTs Fast Inactivating Sodium, Colbert Pan 2002
NMLCH000110: NaTa Fast Inactivating Sodium, Colbert Pan 2002
NMLCH000008: NaF Inactivating Fast Sodium, Maex De Schutter 1998
```

1.2) Select one of these results and download it, and browse the contents

```
# Helper method to retrieve a NeuroML file based on modelID
def get_model_from_neuromldb(model_id, type):

    fname = '%s.%s.nml'%(model_id, type)

    url = 'https://neuroml-db.org/render_xml_file?modelID=%s'%model_id
    r = requests.get(url)
    with open(fname , 'wb') as f:
        f.write(r.content)

    return pynml.read_neuroml2_file(fname), fname


# Choose one of the channels
chan_model_id = 'NMLCH001398'

na_chan_doc, na_chan_fname_orig = get_model_from_neuromldb(chan_model_id, 'channel')

na_chan = na_chan_doc.ion_channel[0] # select the first/only ion channel in the nml␣
 ↪doc

na_chan_fname = '%s.channel.nml'%na_chan.id
os.rename(na_chan_fname_orig, na_chan_fname) # Rename for clarity

print('Channel %s (in file %s) has notes: %s'%(na_chan.id, na_chan_fname, na_chan.
 ↪notes))
```

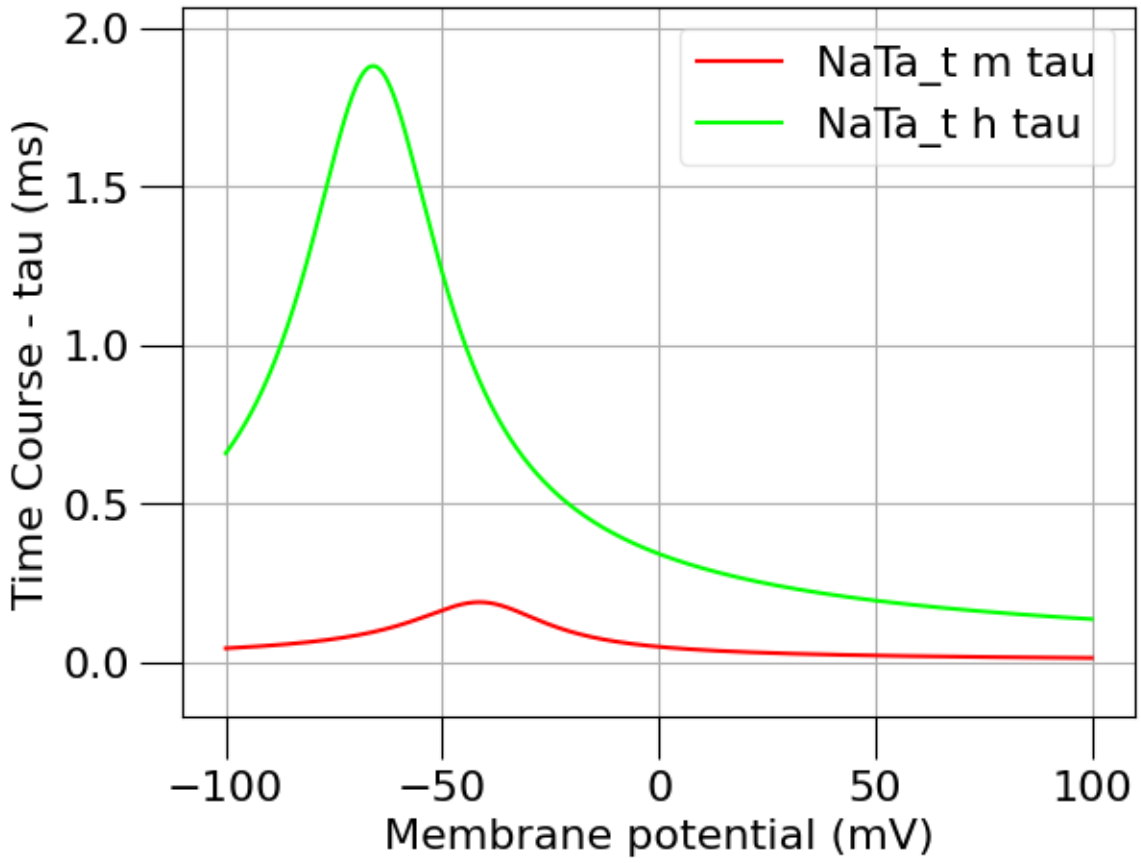
```
pyNeuroML >>> INFO - Loading NeuroML2 file: NMLCH001398.channel.nml
```

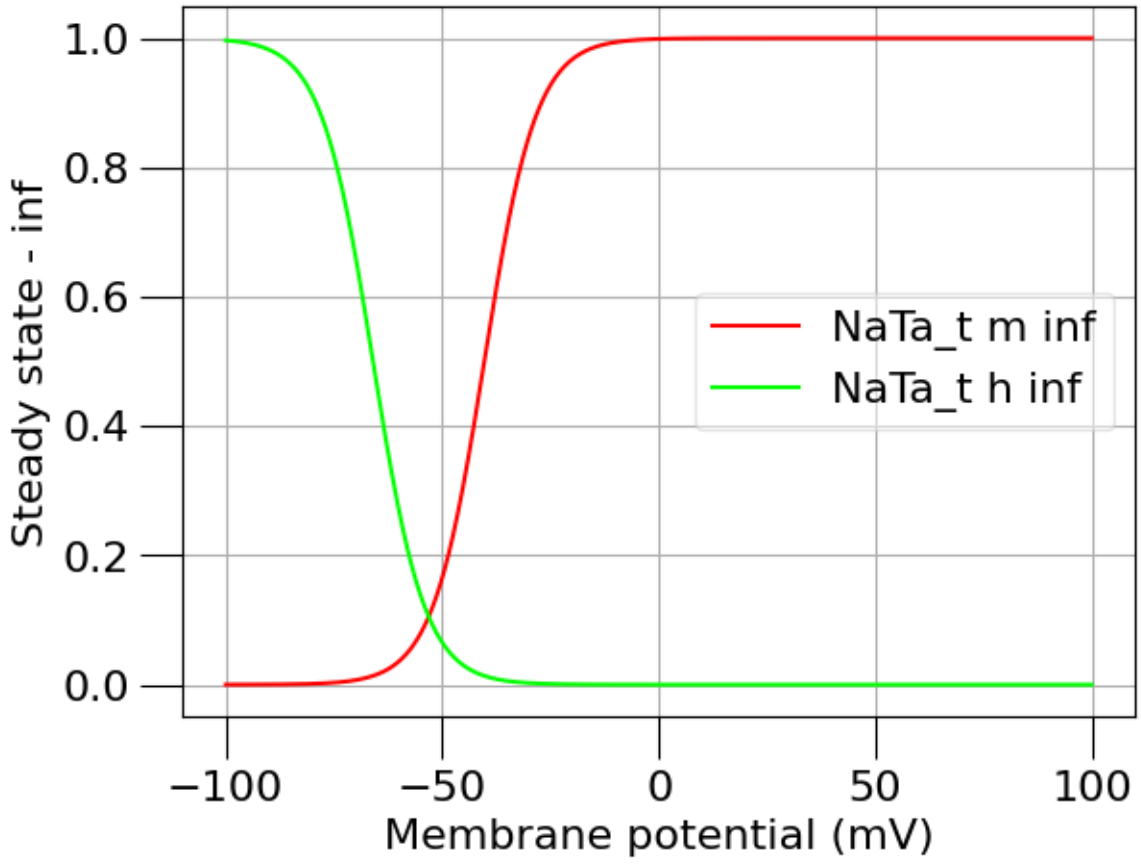
```
Channel NaTa_t (in file NaTa_t.channel.nml) has notes: Fast inactivating Na+␣
 ↪current

Comment from original mod file:
:Reference :Colbert and Pan 2002
```

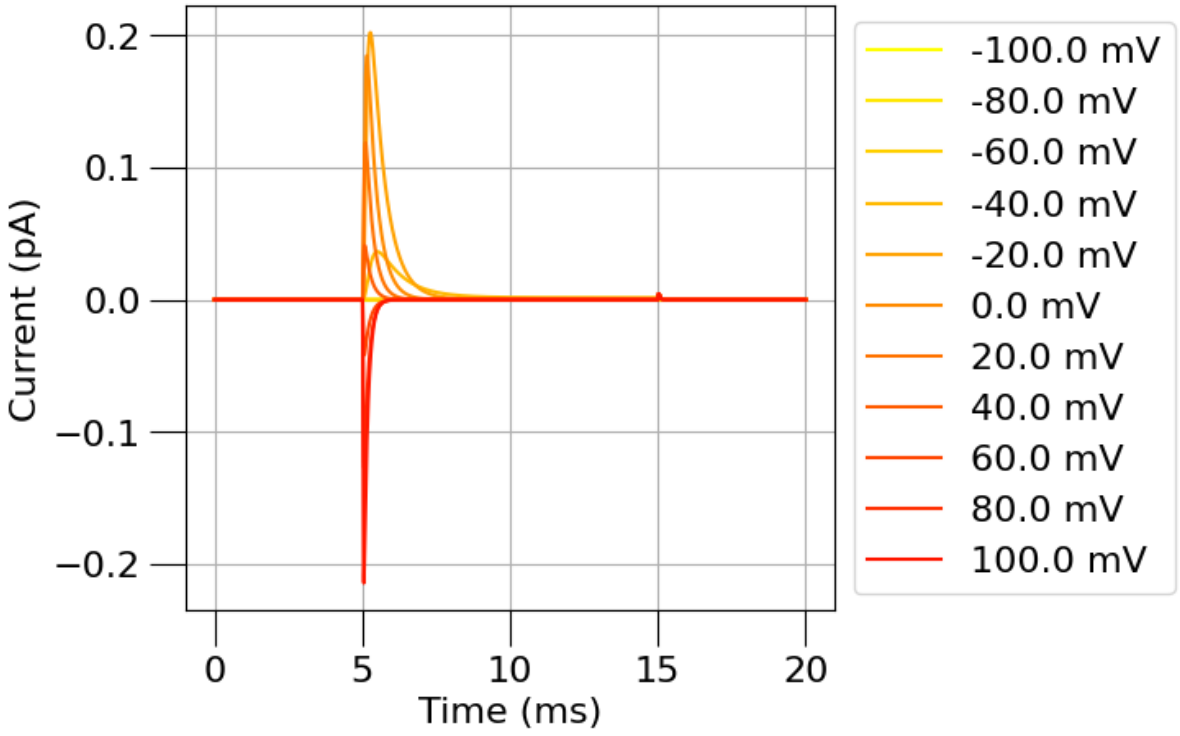
1.3) Plot the time course and steady state of this channel

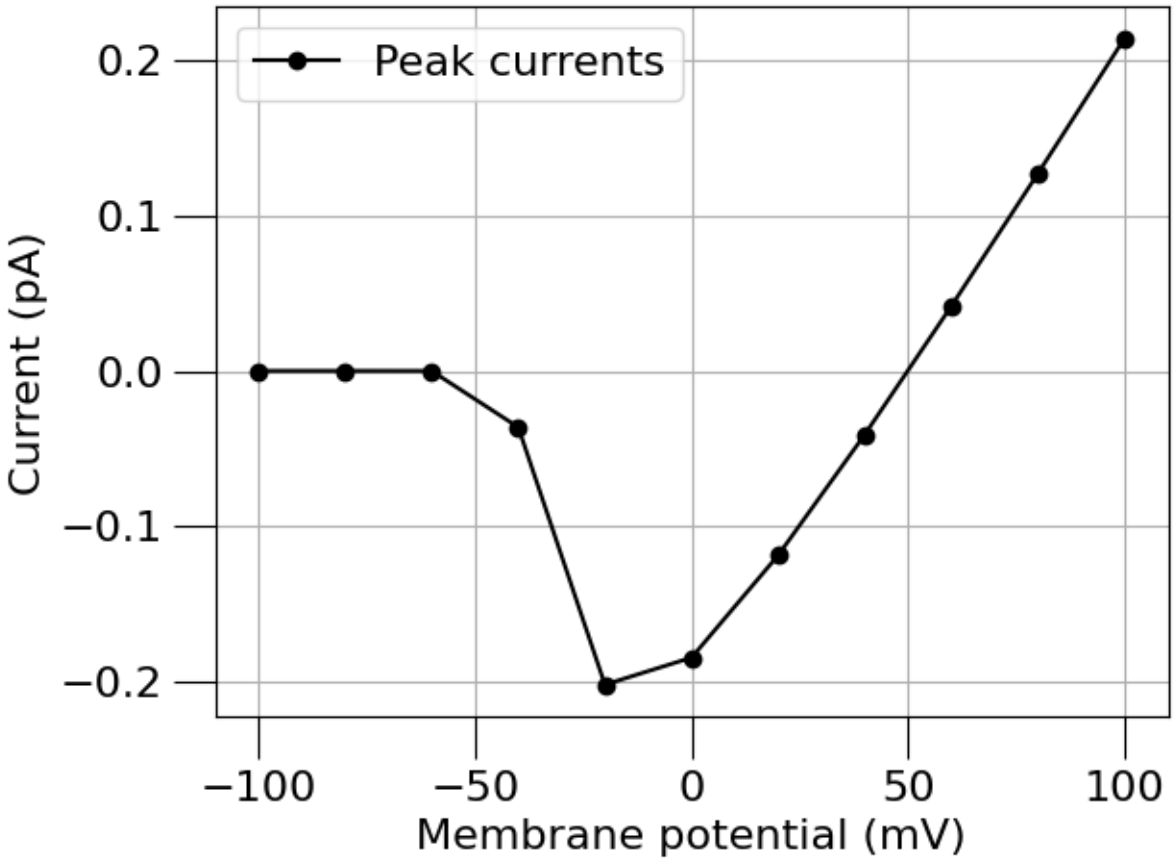
```
from pyneuroml.analysis.NML2ChannelAnalysis import run

na_erev = 50 # mV
run(channel_files=[na_chan_fname], ivCurve=True, erev=na_erev, clampDelay=5,␣
↪clampDuration=10, duration=20)
```

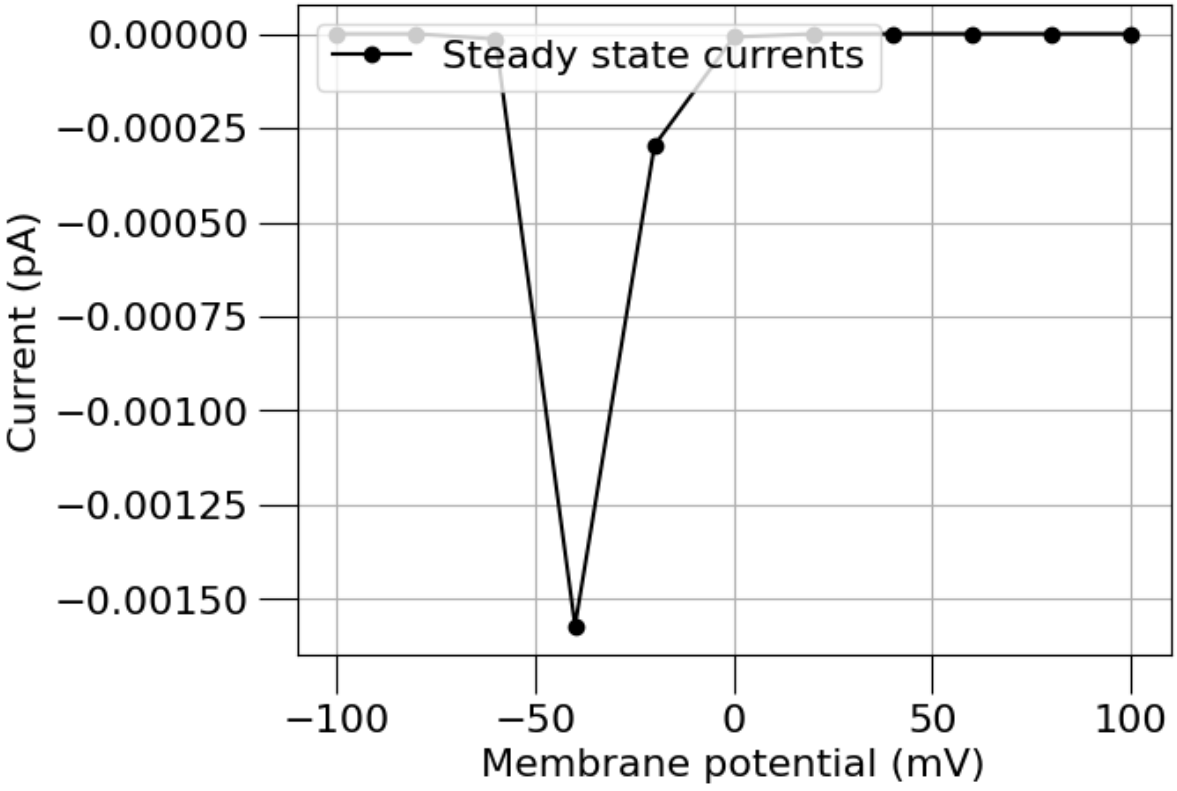
```
pyNeuroML >>> INFO - Loading NeuroML2 file: NaTa_t.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: NaTa_t.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: NaTa_t.channel.nml
pyNeuroML >>> INFO - Loading LEMS file: /Users/padraig/git/Documentation/source/
↪Userdocs/NML2_examples/LEMS_Test_NaTa_t.xml and running with jNeuroML
pyNeuroML >>> INFO - Executing: (java -Xmx400M  -Djava.awt.headless=true -jar  "/
↪opt/homebrew/anaconda3/envs/py39n/lib/python3.9/site-packages/pyneuroml/lib/
↪jNeuroML-0.13.0-jar-with-dependencies.jar"  /Users/padraig/git/Documentation/
↪source/Userdocs/NML2_examples/LEMS_Test_NaTa_t.xml  -nogui -I '') in directory: .
pyNeuroML >>> INFO - Command completed successfully!
```




1.0
0.8
0.6
0.4
0.2
0.0
Steady state - inf
NaTa_t m inf
NaTa_t h inf
−100
−50
0
50
100
Membrane potential (mV)


0.2
0.1
0.0
−0.1
−0.2
Current (pA)
-100.0 mV
-80.0 mV
-60.0 mV
-40.0 mV
-20.0 mV
0.0 mV
20.0 mV
40.0 mV
60.0 mV
80.0 mV
100.0 mV
0
5
10
15
20
Time (ms)


Peak currents
0.2
0.1
0.0
−0.1
−0.2
Current (pA)
−100
−50
0
50
100
Membrane potential (mV)


Steady state currents
0.00000
−0.00025
−0.00050
−0.00075
−0.00100
−0.00125
−0.00150
Current (pA)
−100
−50
0
50
100
Membrane potential (mV)

1.4) Find a Potassium channel

```
data = search_neuromldb("Fast Potassium", 'channel')
```

```
NMLCH000113: K Fast Noninactivating Potassium, Rettig Wunder 1992
NMLCH001627: KDr Fast Delayed Rectifier Potassium , Bezaire Raikov 2016
NMLCH001611: KDr Fast Delayed Rectifier Potassium, Bezaire Raikov 2016
NMLCH001609: KDr Fast Delayed Rectifier Potassium , Bezaire Raikov 2016
NMLCH001549: K Fast Potassium, Boyle Cohen 2008
NMLCH001529: KT Fast Inactivating Potassium, Gouwens Berg 2018
NMLCH001465: K Fast Noninactivating Potassium, Gouwens Berg 2018
NMLCH001400: KTst Fast Inactivating Potassium, Hay Hill 2011
NMLCH001394: K Fast Noninactivating Potassium, Hay Hill 2011
NMLCH000156: IM M Type Potassium, Traub Buhl 2003
NMLCH000155: KDr Delayed Rectifier Potassium, Traub Buhl 2003
NMLCH000154: KDr Delayed Rectifier Potassium Channel for Fast Spiking (FS)␣
↪Interneurons, Traub Contreras 2005
NMLCH000153: KCa BK Type Fast Calcium Dependent Potassium, Traub Buhl 2003
NMLCH000152: KCa BK Type Fast Calcium Dependent Potassium, Traub Buhl 2003
NMLCH000151: KCa AHP Type Calcium Dependent Potassium, Traub Buhl 2003
NMLCH000149: KCa Slow AHP Type Calcium Dependent Potassium, Traub Buhl 2003
NMLCH000148: IA A Type Potassium, Traub Buhl 2003
NMLCH000146: K2 Type Slowly Activating and Inactivating Potassium, Traub Buhl 2003
NMLCH000123: K Fast Potassium, Korngreen Sakmann 2000
NMLCH000108: KTst Fast Inactivating Potassium, Korngreen Sakmann 2000
```

```
# Download one of these

k_chan_doc, k_chan_fname_orig = get_model_from_neuromldb('NMLCH000113', 'channel')

k_chan = k_chan_doc.ion_channel[0] # select the first ion channel in the nml doc

k_chan_fname = '%s.channel.nml'%k_chan.id
os.rename(k_chan_fname_orig, k_chan_fname) # Rename for clarity

print('Channel %s (in file %s) has notes: %s'%(k_chan.id, k_chan_fname, k_chan.notes))

k_erev = -77
run(channel_files=[k_chan_fname], ivCurve=True, erev=k_erev)
```

```
pyNeuroML >>> INFO - Loading NeuroML2 file: NMLCH000113.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: SKv3_1.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: SKv3_1.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: SKv3_1.channel.nml
pyNeuroML >>> INFO - Loading LEMS file: /Users/padraig/git/Documentation/source/
↪Userdocs/NML2_examples/LEMS_Test_SKv3_1.xml and running with jNeuroML
pyNeuroML >>> INFO - Executing: (java -Xmx400M  -Djava.awt.headless=true -jar  "/
↪opt/homebrew/anaconda3/envs/py39n/lib/python3.9/site-packages/pyneuroml/lib/
↪jNeuroML-0.13.0-jar-with-dependencies.jar"  /Users/padraig/git/Documentation/
↪source/Userdocs/NML2_examples/LEMS_Test_SKv3_1.xml  -nogui -I '') in directory: .
```

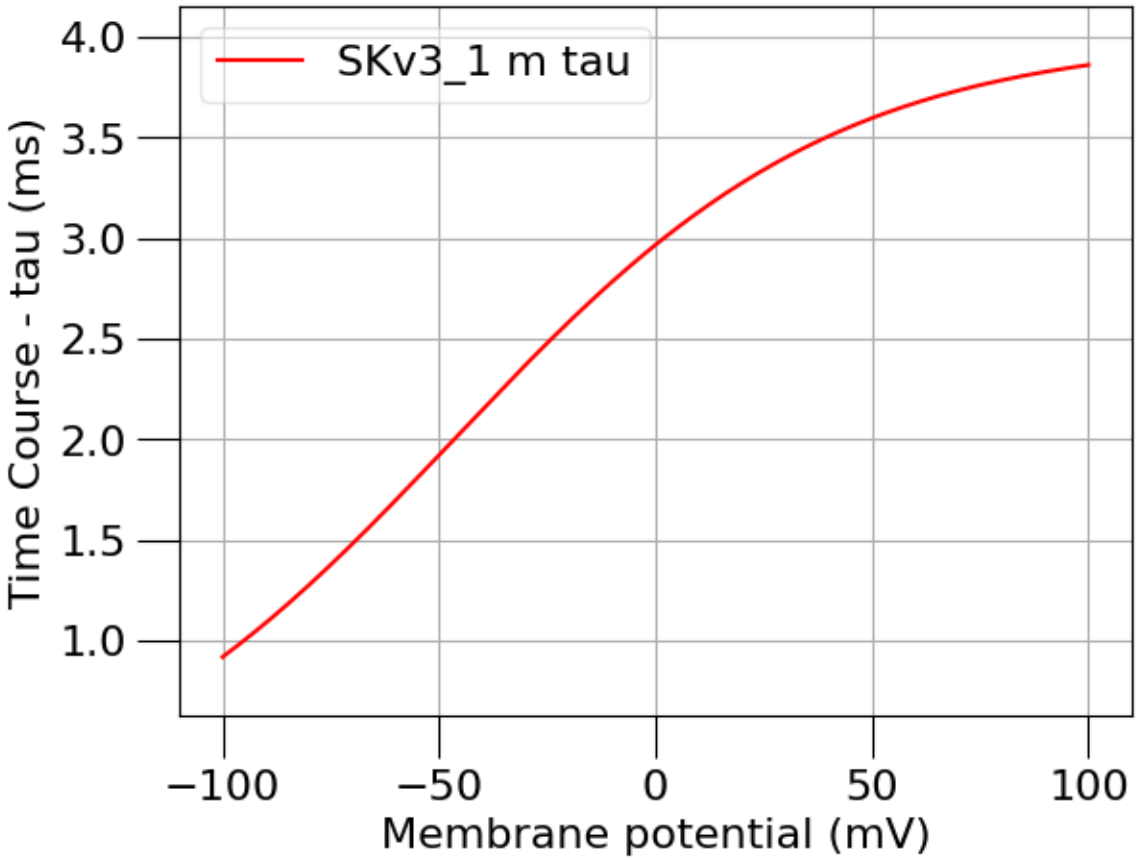
```
Channel SKv3_1 (in file SKv3_1.channel.nml) has notes: Fast, non inactivating K+␣
↪current

Comment from original mod file:
:Reference : :               Characterization of a Shaw-related potassium channel␣
```

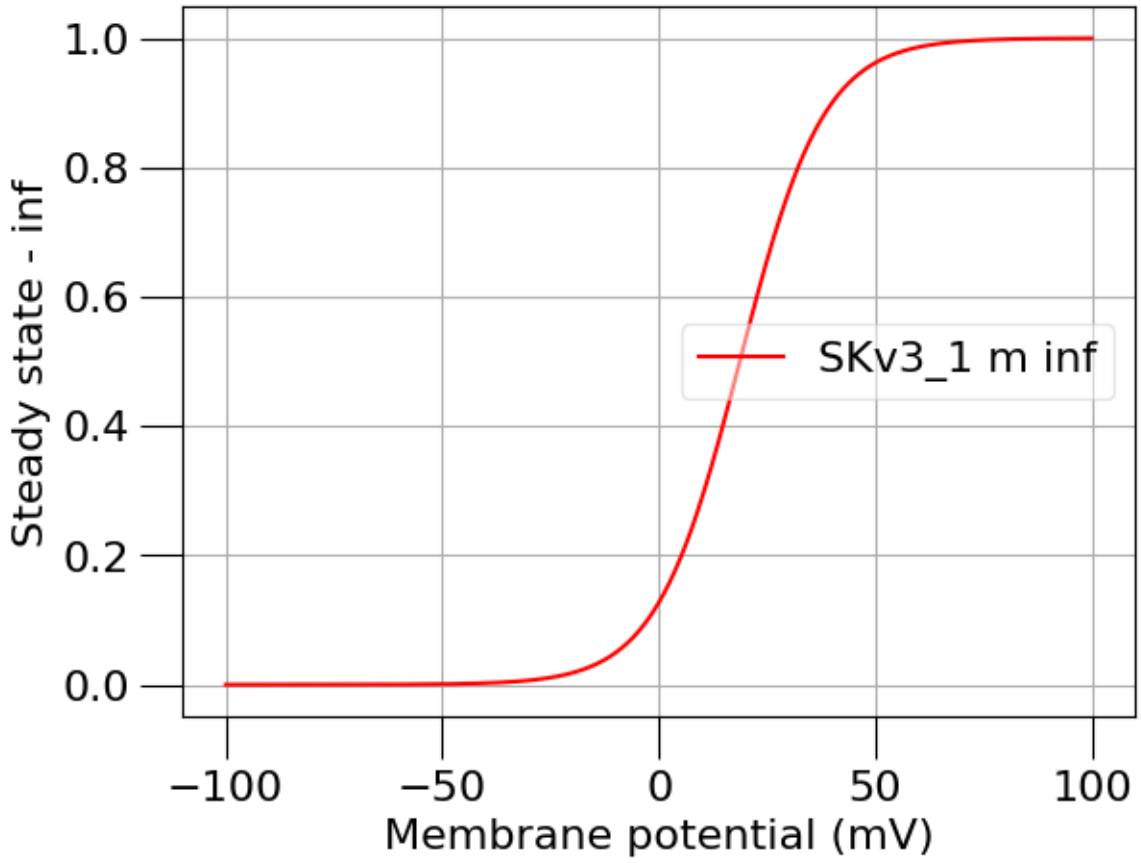
(continues on next page)

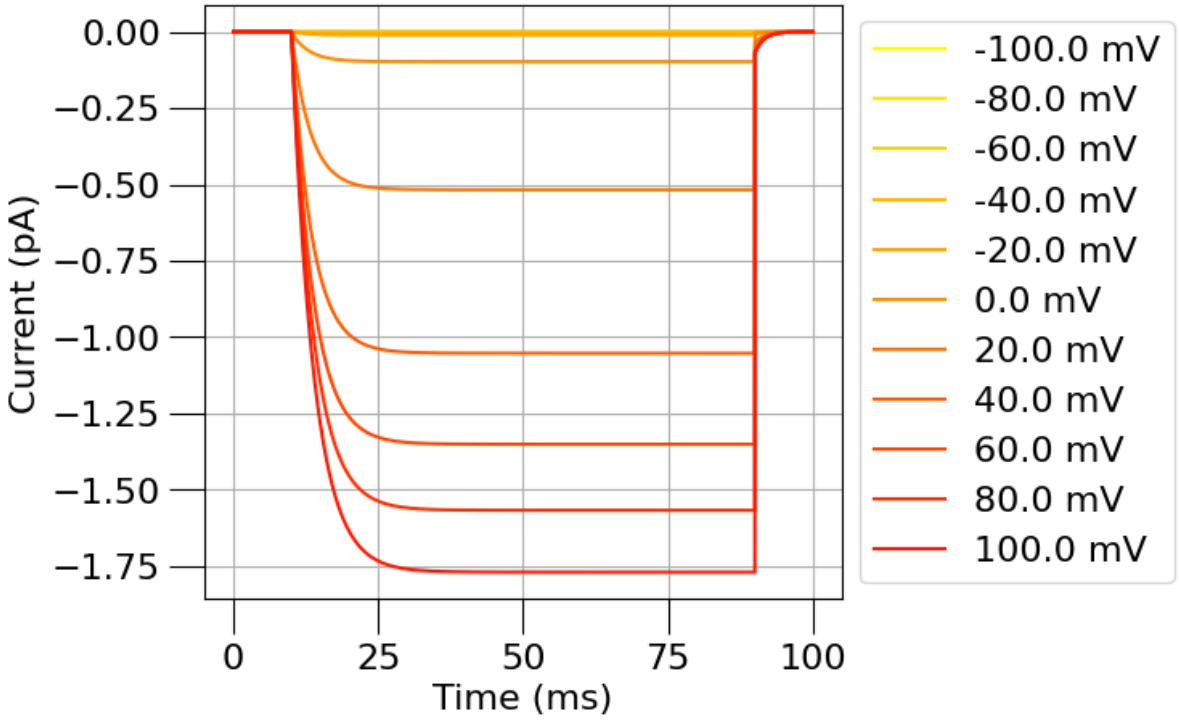
(continued from previous page)

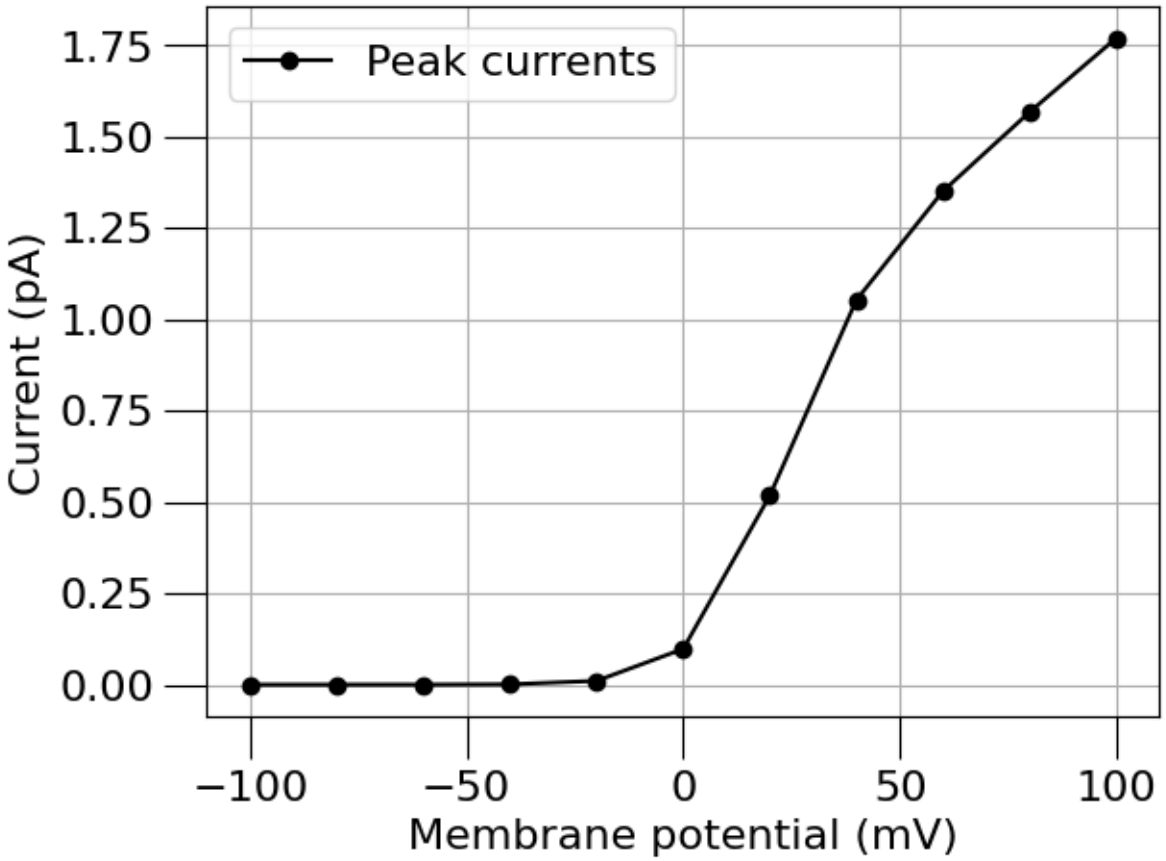
```
↪family in rat brain, The EMBO Journal, vol.11, no.7,2473-2486 (1992)
```

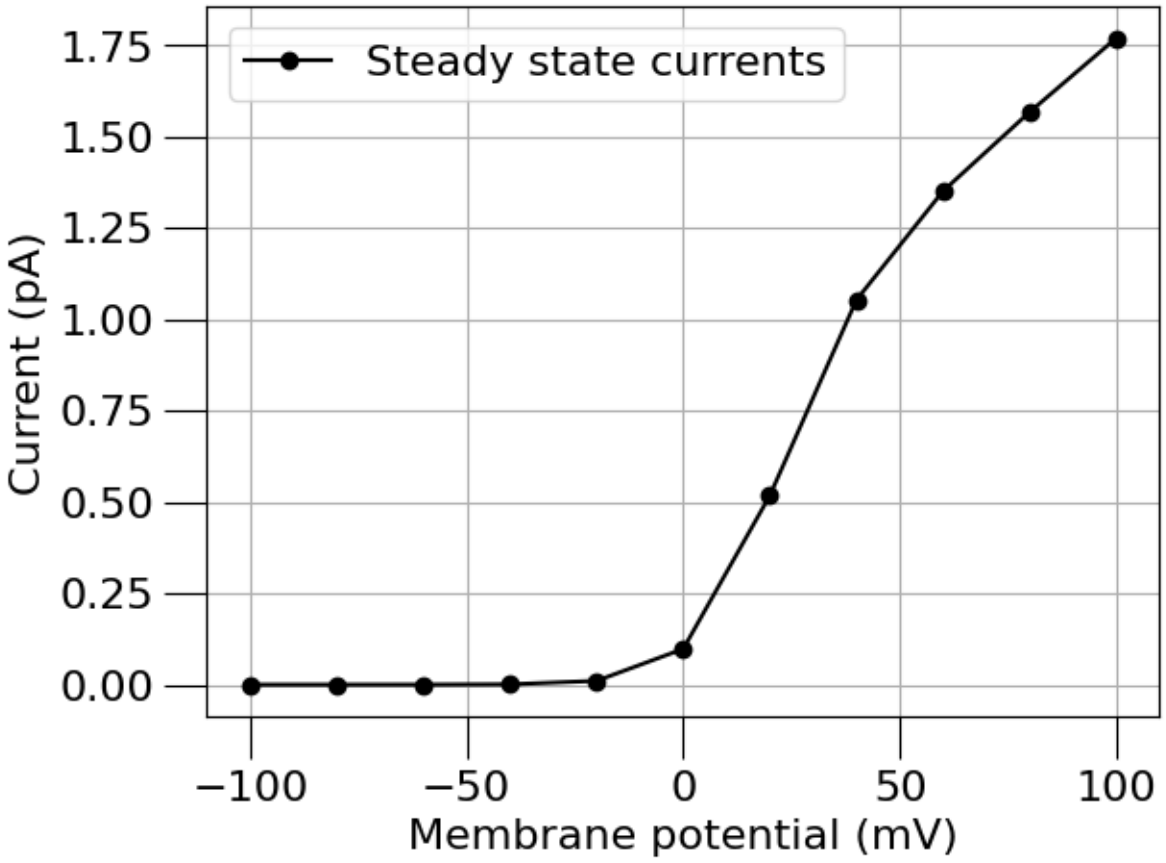
```
pyNeuroML >>> INFO - Command completed successfully!
```




1.0
0.8
0.6
0.4
0.2
0.0
Steady state - inf
SKv3_1 m inf
−100
−50
0
50
100
Membrane potential (mV)


0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
−1.75
Current (pA)
0
25
50
75
100
Time (ms)
-100.0 mV
-80.0 mV
-60.0 mV
-40.0 mV
-20.0 mV
0.0 mV
20.0 mV
40.0 mV
60.0 mV
80.0 mV
100.0 mV


Peak currents
Current (pA)
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−100
−50
0
50
100
Membrane potential (mV)



1.5) Find a passive (leak) current

```
data = search_neuromldb("Leak", 'channel')

pas_chan_doc, pas_chan_fname = get_model_from_neuromldb(data[0]['Model_ID'], 'channel
↪')
pas_chan = pas_chan_doc.ion_channel[0] # select the first ion channel in the nml doc

print('Channel %s has notes: %s'%(pas_chan.id, pas_chan.notes))
```

```
NMLCH000015: Passive Leak, Maex De Schutter 1998
NMLCH001623: Passive Leak, Bezaire Raikov 2016
NMLCH001601: Passive Leak, Smith Smith 2013
NMLCH001590: Passive Leak, Prinz Bucher 2004
NMLCH001545: Passive Leak, Boyle Cohen 2008
NMLCH001471: Passive Leak, Gouwens Berg 2018
NMLCH001427: Passive Leak, Hodgkin Huxley 1952
NMLCH001403: Passive Leak, Hay Hill 2011
NMLCH000172: Passive Leak, Traub Contreras 2005
NMLCH000130: Passive Leak, Pospischil Toledo-Rodriguez 2008
NMLCH000114: Passive Leak, Hodgkin Huxley 1952
NMLCH000095: Passive Leak, Vervaeke Lorincz 2010
NMLCH000027: Passive Leak, De Schutter Bower 1994
NMLCH000024: Passive Leak, Maex De Schutter 1998
NMLCH000016: Passive Leak, Maex De Schutter 1998
```

(continues on next page)

(continued from previous page)

```
NMLCH000007: Passive Leak, Migliore Ferrante 2005
NMLCH000139: Passive Leak, Hodgkin Huxley 1952
```

```
pyNeuroML >>> INFO - Loading NeuroML2 file: NMLCH000015.channel.nml
```

```
Channel GranPassiveCond has notes: Simple leak conductance for Granule cell
```

4.9.2 2) Create a new cell model using these channels

2.1) Create the Cell object, add channels and save to file

```
from neuroml import *
from neuroml.utils import component_factory
import neuroml.writers as writers


nml_doc = NeuroMLDocument(id="TestCell")

cell = component_factory("Cell", id="novel_cell")
nml_doc.add(cell)

cell.add_segment(prox=[0,0,0,17.841242],
                 dist=[0,0,0,17.841242],
                 seg_type='soma')

cell.set_resistivity('0.03 kohm_cm')
cell.set_init_memb_potential('-65mV')
cell.set_specific_capacitance('1.0 uF_per_cm2')
cell.set_spike_thresh('0mV')


cell.add_channel_density(nml_doc,
                         cd_id='%s_chans'%na_chan.id,
                         ion_channel=na_chan.id,
                         cond_density='150 mS_per_cm2',
                         ion_chan_def_file=na_chan_fname,
                         erev="%s mV"%na_erev)

cell.add_channel_density(nml_doc,
                         cd_id='%s_chans'%k_chan.id,
                         ion_channel=k_chan.id,
                         cond_density='36 mS_per_cm2',
                         ion_chan_def_file=k_chan_fname,
                         erev="%s mV"%k_erev)

cell.add_channel_density(nml_doc,
                         cd_id='%s_chans'%pas_chan.id,
                         ion_channel=pas_chan.id,
                         cond_density='0.3 mS_per_cm2',
                         ion_chan_def_file=pas_chan_fname,
                         erev="-65 mV")
```

(continues on next page)

(continued from previous page)

```
cell_file = "%s.cell.nml"%nml_doc.id
writers.NeuroMLWriter.write(nml_doc, cell_file)


print("Written cell file to: " + cell_file)

from neuroml.utils import validate_neuroml2

validate_neuroml2(cell_file)

#!cat TestCell.cell.nml


pynml.summary(nml_doc, verbose=True)
```

```
Warning: Segment group all already exists.
Warning: Segment group soma_group already exists.
Written cell file to: TestCell.cell.nml
It's valid!
*******************************************************
* NeuroMLDocument: TestCell
*
*
*  Cell: novel_cell
*    <Segment|0|Seg0>
*      Parent segment: None (root segment)
*      (0.0, 0.0, 0.0), diam 17.841242um -> (0.0, 0.0, 0.0), diam 17.841242um; seg␣
 ↪length: 0.0 um
*      Surface area: 1000.0000939925986 um2, volume: 2973.540612824116 um3
*    Total length of 1 segment: 0.0 um; total area: 1000.0000939925986 um2
*
*    SegmentGroup: soma_group,        1 member(s),        0 included group(s);
 ↪        contains 1 segment, id: 0
*    SegmentGroup: all,        1 member(s),        0 included group(s);
 ↪        contains 1 segment, id: 0
*
*    Channel density: NaTa_t_chans on all;        conductance of 150 mS_per_cm2␣
 ↪through ion chan NaTa_t with ion non_specific, erev: 50 mV
*      Channel is on <Segment|0|Seg0>,        total conductance: 1500.0 S_per_m2 x␣
 ↪1.0000000939925986e-09 m2 = 1.500000140988898e-06 S (1500000.140988898 pS)
*    Channel density: SKv3_1_chans on all;        conductance of 36 mS_per_cm2␣
 ↪through ion chan SKv3_1 with ion non_specific, erev: -77 mV
*      Channel is on <Segment|0|Seg0>,        total conductance: 360.0 S_per_m2 x␣
 ↪1.0000000939925986e-09 m2 = 3.600000338373355e-07 S (360000.0338373355 pS)
*    Channel density: GranPassiveCond_chans on all;        conductance of 0.3 mS_
 ↪per_cm2 through ion chan GranPassiveCond with ion non_specific, erev: -65 mV
*      Channel is on <Segment|0|Seg0>,        total conductance: 3.0 S_per_m2 x 1.
 ↪0000000939925986e-09 m2 = 3.000000281977796e-09 S (3000.000281977796 pS)
*
*    Specific capacitance on all: 1.0 uF_per_cm2
*      Capacitance of <Segment|0|Seg0>,        total capacitance: 0.01 F_per_m2 x␣
 ↪1.0000000939925986e-09 m2 = 1.0000000939925987e-11 F (10.000000939925988 pF)
*
*******************************************************
```

```
/opt/homebrew/anaconda3/envs/py39n/lib/python3.9/site-packages/neuroml/nml/
 ↪generatedssupersuper.py:176: UserWarning: morphology has already been assigned. ␣
 ↪Use `force=True` to overwrite. Hint: you can make changes to the already added␣
 ↪object as required without needing to re-add it because only references to the␣
 ↪objects are added, not their values.
  warnings.warn(
/opt/homebrew/anaconda3/envs/py39n/lib/python3.9/site-packages/neuroml/nml/
 ↪generatedssupersuper.py:176: UserWarning: biophysical_properties has already␣
 ↪been assigned.  Use `force=True` to overwrite. Hint: you can make changes to the␣
 ↪already added object as required without needing to re-add it because only␣
 ↪references to the objects are added, not their values.
  warnings.warn(
/opt/homebrew/anaconda3/envs/py39n/lib/python3.9/site-packages/neuroml/nml/
 ↪generatedssupersuper.py:176: UserWarning: intracellular_properties has already␣
 ↪been assigned.  Use `force=True` to overwrite. Hint: you can make changes to the␣
 ↪already added object as required without needing to re-add it because only␣
 ↪references to the objects are added, not their values.
  warnings.warn(
/opt/homebrew/anaconda3/envs/py39n/lib/python3.9/site-packages/neuroml/nml/
 ↪generatedssupersuper.py:176: UserWarning: membrane_properties has already been␣
 ↪assigned.  Use `force=True` to overwrite. Hint: you can make changes to the␣
 ↪already added object as required without needing to re-add it because only␣
 ↪references to the objects are added, not their values.
  warnings.warn(
```

2.2) Generate a plot of the actvity of the cell under current clamp input

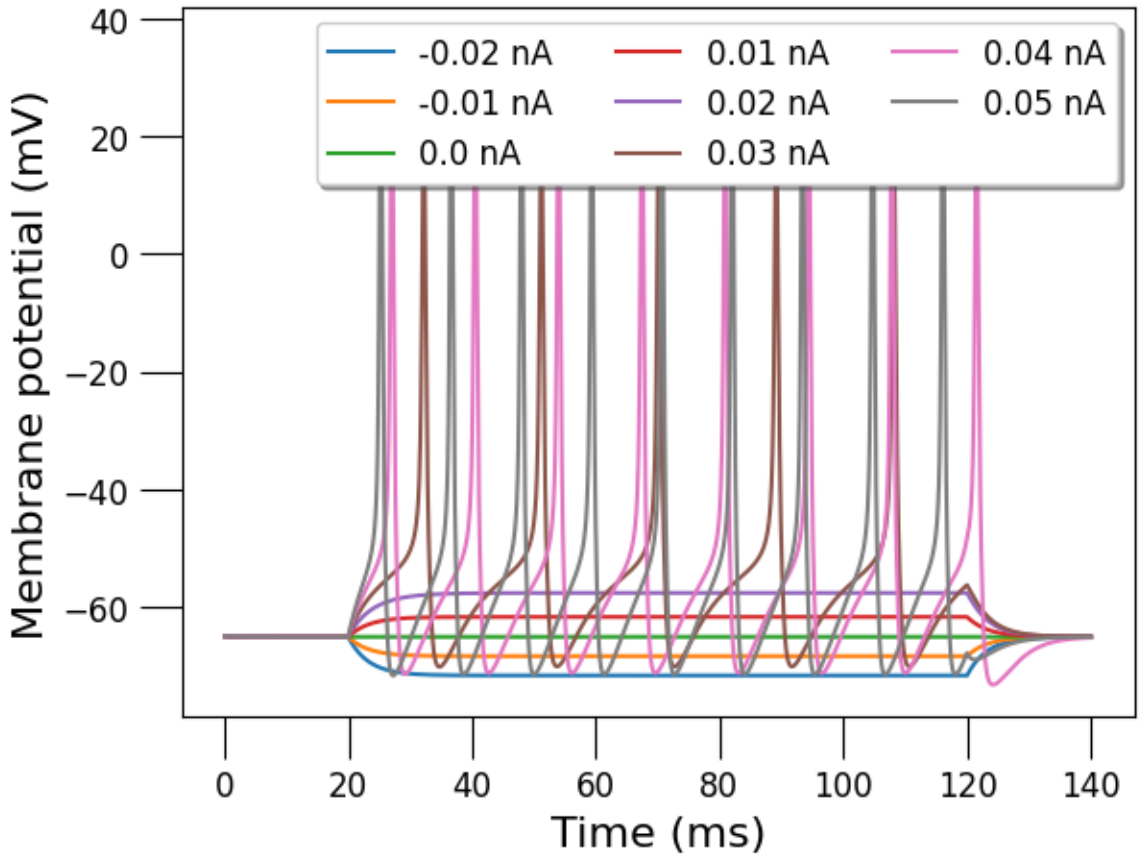
```
from pyneuroml.analysis import generate_current_vs_frequency_curve

generate_current_vs_frequency_curve(cell_file,
    cell.id,
    start_amp_nA=-0.02,
    end_amp_nA=0.06,
    step_nA=0.01,
    pre_zero_pulse=20,
    post_zero_pulse=20,
    analysis_duration=100,
    temperature='34degC',
    plot_voltage_traces=True)
```

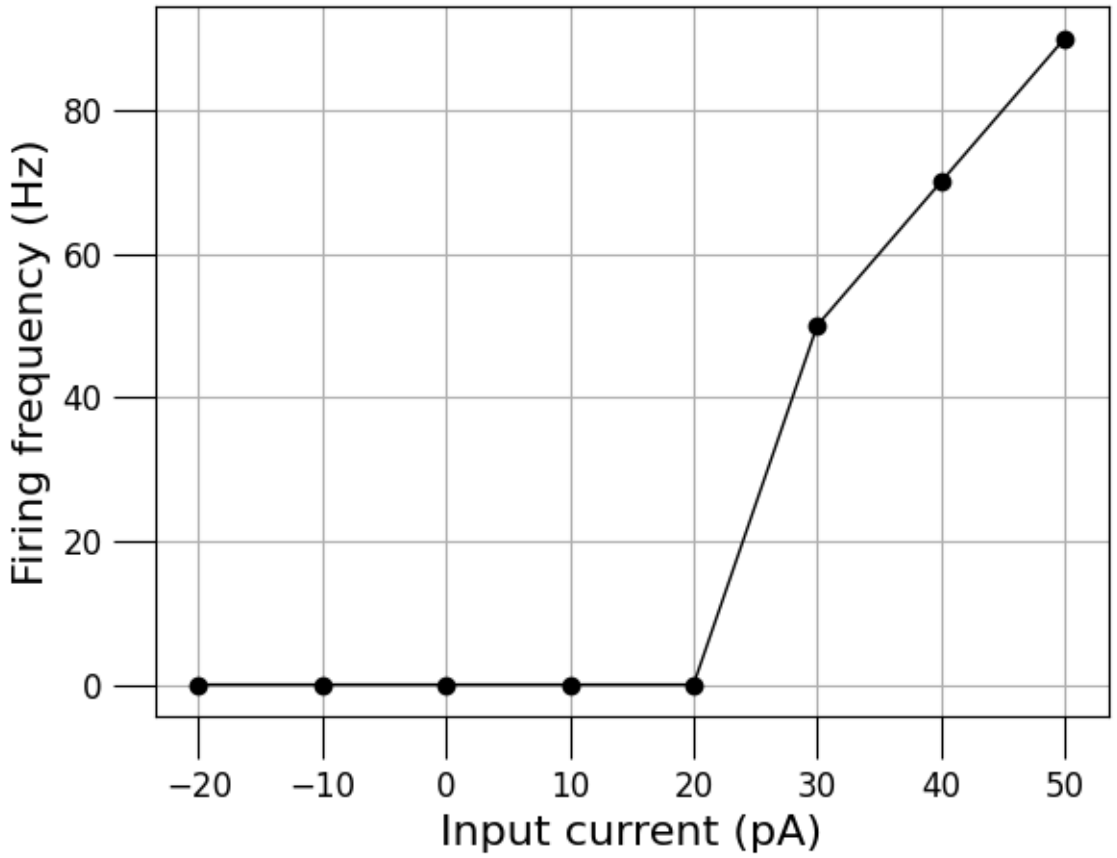
```
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/TestCell.cell.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/NaTa_t.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/SKv3_1.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/NMLCH000015.channel.nml
pyNeuroML >>> INFO - Executing: (java -Xmx400M  -jar  "/opt/homebrew/anaconda3/
 ↪envs/py39n/lib/python3.9/site-packages/pyneuroml/lib/jNeuroML-0.13.0-jar-with-
 ↪dependencies.jar" -validate iv_novel_cell.net.nml ) in directory: .
pyNeuroML >>> INFO - Command completed successfully!
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/iv_novel_cell.net.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
```

(continues on next page)

(continued from previous page)

```
 ↪source/Userdocs/NML2_examples/TestCell.cell.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/NaTa_t.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/SKv3_1.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/NMLCH000015.channel.nml
pyNeuroML >>> INFO - Loading LEMS file: LEMS_iv_novel_cell.xml and running with␣
 ↪jNeuroML
pyNeuroML >>> INFO - Executing: (java -Xmx400M  -Djava.awt.headless=true -jar  "/
 ↪opt/homebrew/anaconda3/envs/py39n/lib/python3.9/site-packages/pyneuroml/lib/
 ↪jNeuroML-0.13.0-jar-with-dependencies.jar"  LEMS_iv_novel_cell.xml  -nogui -I '
 ↪') in directory: .
pyNeuroML >>> INFO - Command completed successfully!
/opt/homebrew/anaconda3/envs/py39n/lib/python3.9/site-packages/numpy/core/
 ↪fromnumeric.py:3432: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/opt/homebrew/anaconda3/envs/py39n/lib/python3.9/site-packages/numpy/core/_methods.
 ↪py:190: RuntimeWarning: invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
pyNeuroML >>> INFO - Generating plot: Membrane potential traces for: TestCell.cell.
 ↪nml
pyNeuroML >>> INFO - Generating plot: Firing frequency versus injected current␣
 ↪for: TestCell.cell.nml
```






```
{-0.02: 0.0,
 -0.01: 0.0,
 0.0: 0.0,
 0.01: 0.0,
 0.02: 0.0,
 0.03: 50.0,
 0.04: 70.0,
 0.05: 90.0}
```

4.9.3 3) Search for, download and analyse a complete NeuroML cell

3.1) Search for and download the cell

```
data = search_neuromldb("Burst Accommodating Martinotti", 'cell')
```

```
NMLCL000109: Layer 2/3 Burst Accommodating Martinotti Cell (3), Markram Muller 2015
NMLCL000395: Layer 6 Burst Non-accommodating Martinotti Cell (4), Markram Muller␣
 ↪2015
NMLCL000393: Layer 6 Burst Non-accommodating Martinotti Cell (2), Markram Muller␣
 ↪2015
NMLCL000345: Layer 4 Burst Non-accommodating Martinotti Cell (4), Markram Muller␣
 ↪2015
NMLCL000343: Layer 4 Burst Non-accommodating Martinotti Cell (2), Markram Muller␣
 ↪2015
```

(continues on next page)

(continued from previous page)

```
NMLCL000315: Layer 2/3 Burst Non-accommodating Martinotti Cell (4), Markram Muller␣
↪2015
NMLCL000313: Layer 2/3 Burst Non-accommodating Martinotti Cell (2), Markram Muller␣
↪2015
NMLCL000191: Layer 6 Burst Accommodating Martinotti Cell (5), Markram Muller 2015
NMLCL000190: Layer 6 Burst Accommodating Martinotti Cell (4), Markram Muller 2015
NMLCL000188: Layer 6 Burst Accommodating Martinotti Cell (2), Markram Muller 2015
NMLCL000161: Layer 5 Burst Accommodating Martinotti Cell (5), Markram Muller 2015
NMLCL000160: Layer 5 Burst Accommodating Martinotti Cell (4), Markram Muller 2015
NMLCL000158: Layer 5 Burst Accommodating Martinotti Cell (2), Markram Muller 2015
NMLCL000157: Layer 5 Burst Accommodating Martinotti Cell (1), Markram Muller 2015
NMLCL000136: Layer 4 Burst Accommodating Martinotti Cell (5), Markram Muller 2015
NMLCL000134: Layer 4 Burst Accommodating Martinotti Cell (3), Markram Muller 2015
NMLCL000133: Layer 4 Burst Accommodating Martinotti Cell (2), Markram Muller 2015
NMLCL000111: Layer 2/3 Burst Accommodating Martinotti Cell (5), Markram Muller 2015
NMLCL000108: Layer 2/3 Burst Accommodating Martinotti Cell (2), Markram Muller 2015
NMLCL000187: Layer 6 Burst Accommodating Martinotti Cell (1), Markram Muller 2015
```

```
model_id = 'NMLCL000109'

def get_model_details_from_neuromldb(model_id):

    url = 'https://neuroml-db.org/api/model?id=%s'%model_id

    with urllib.request.urlopen(url) as res:
        model_details = json.load(res)

    for k in model_details['model']: print('%s:\t%s'%(k,model_details['model'][k]))

    return model_details


model_details = get_model_details_from_neuromldb(model_id)
```

```
Model_ID:        NMLCL000109
Status:        CURRENT
Errors:        None
Status_Timestamp:        2018-12-24T16:29:21+00:00
Type:        Cell
Equations:        2061
Runtime_Per_Step:        0.000510815108024267
Max_Stable_DT:        0.0625
Max_Stable_DT_Error:        0.969373115260111
Max_Stable_DT_Benchmark_RunTime:        4.903825037032964
Optimal_DT:        0.00781557069252977
Optimal_DT_Error:        0.122802372280653
Optimal_DT_Benchmark_RunTime:        39.21518682026722
Optimal_DT_a:        0.000968992248062015
Optimal_DT_b:        15.8634962901345
Optimal_DT_c:        -0.0136485748710791
CVODE_baseline_step_frequency:        13111.7719061286
CVODE_steps_per_spike:        766.973690134512
CVODE_Benchmark_RunTime:        10.615508666396945
Name:        Layer 2/3 Burst Accommodating Martinotti Cell (3)
Directory_Path:        /var/www/NeuroMLmodels/NMLCL000109
File_Name:        bAC217_L23_MC_40be3bf0e8_0_0.cell.nml
```

(continues on next page)

(continued from previous page)

```
File_Updated:        None
File_MD5_Checksum:        bcf74ccd8c6840f6c8f7551084c989b7
File:        /var/www/NeuroMLmodels/NMLCL000109/bAC217_L23_MC_40be3bf0e8_0_0.cell.
↪nml
Publication_ID:        6000246
Upload_Time:        2016-12-14T14:29:41+00:00
Notes:        None
ID_Helper:        175
Sections:        40
Compartments:        204
Stability_Range_Low:        None
Stability_Range_High:        None
Is_Passive:        0
Is_Intrinsically_Spiking:        0
Resting_Voltage:        -71.2726975181673
Rheobase_Low:        0.0469207763671875
Rheobase_High:        0.048065185546875
Threshold_Current_Low:        0.234375
Threshold_Current_High:        0.29296875
Bias_Voltage:        -80.0
Bias_Current:        -0.0221022438906573
CVODE_Active:        None
Threshold:        None
Is_GLIF:        0
V_Variable:        None
Steady_State_Delay:        None
AP12AmplitudeDrop:        1.2698800006924
AP1SSAmplitudeChange:        2.3903765783198
AP1Amplitude:        72.1822400011024
AP1WidthHalfHeight:        0.53
AP1WidthPeakToTrough:        5.03000000000009
AP1RateOfChangePeakToTrough:        -17.8182926253306
AP1AHPDepth:        17.4437719043123
AP2Amplitude:        70.91236000041
AP2WidthHalfHeight:        0.55
AP2WidthPeakToTrough:        6.22000000000003
AP2RateOfChangePeakToTrough:        -14.2206642462703
AP2AHPDepth:        17.5401716113915
AP12AmplitudeChangePercent:        -1.75926931704115
AP12HalfWidthChangePercent:        3.77358490566034
AP12RateOfChangePeakToTroughPercentChange:        -20.1906459541805
AP12AHPDepthPercentChange:        0.552631091532312
InputResistance:        386.187465741356
AP1DelayMean:        20.6200000000001
AP1DelaySD:        0.0
AP2DelayMean:        72.1300000000001
AP2DelaySD:        0.0
Burst1ISIMean:        60.255
Burst1ISISD:        0.0
InitialAccommodationMean:        -75.0
SSAccommodationMean:        -50.0
AccommodationRateToSS:        -0.143802128271498
AccommodationAtSSMean:        -83.5991575049674
AccommodationRateMeanAtSS:        166.006825685025
ISICV:        2.95112315366106
ISIMedian:        216.41
ISIBurstMeanChange:        33.954571927781
```

(continues on next page)

(continued from previous page)

```
SpikeRateStrongStim:        10.0
AP1DelayMeanStrongStim:        7.30999999999995
AP1DelaySDStrongStim:        0.0
AP2DelayMeanStrongStim:        33.8900000000001
AP2DelaySDStrongStim:        0.0
Burst1ISIMeanStrongStim:        28.135
Burst1ISISDStrongStim:        0.0
RampFirstSpike:        2046.34
FrequencyFilterType:        Low-Pass
FrequencyPassAbove:        29.0
FrequencyPassBelow:        58.7316087498722
Ephyz_Cluster_ID:        /3/1/4/
Channel_Type:        None
Time_Step:        None
Max_Stable_DT_Benchmark_RunTime_HH:        911.1330720253954
Optimal_DT_Benchmark_RunTime_HH:        466.46500114887556
CVODE_Benchmark_RunTime_HH:        498.01998845545205
Stability_Range_Low_Corr:        -1.875
Stability_Range_High_Corr:        15.0
```

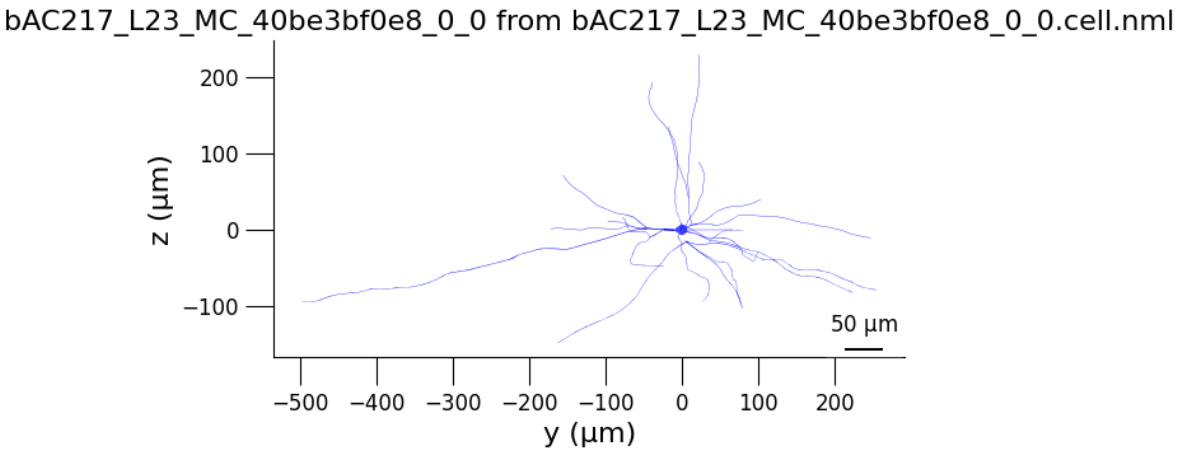
```
def get_full_model_from_neuromldb(model_id):

    fname = '%s.nml.zip'%(model_id)

    url = 'https://neuroml-db.org/GetModelZip?modelID=%s&version=NeuroML'%model_id
    r = requests.get(url)
    with open(fname , 'wb') as f:
        f.write(r.content)


    import zipfile
    with zipfile.ZipFile(fname, 'r') as z:
        z.extractall('.')


    print('Saved as %s'%fname)

get_full_model_from_neuromldb(model_id)

detailed_cell_file = model_details['model']['File_Name']

validate_neuroml2(detailed_cell_file)
```

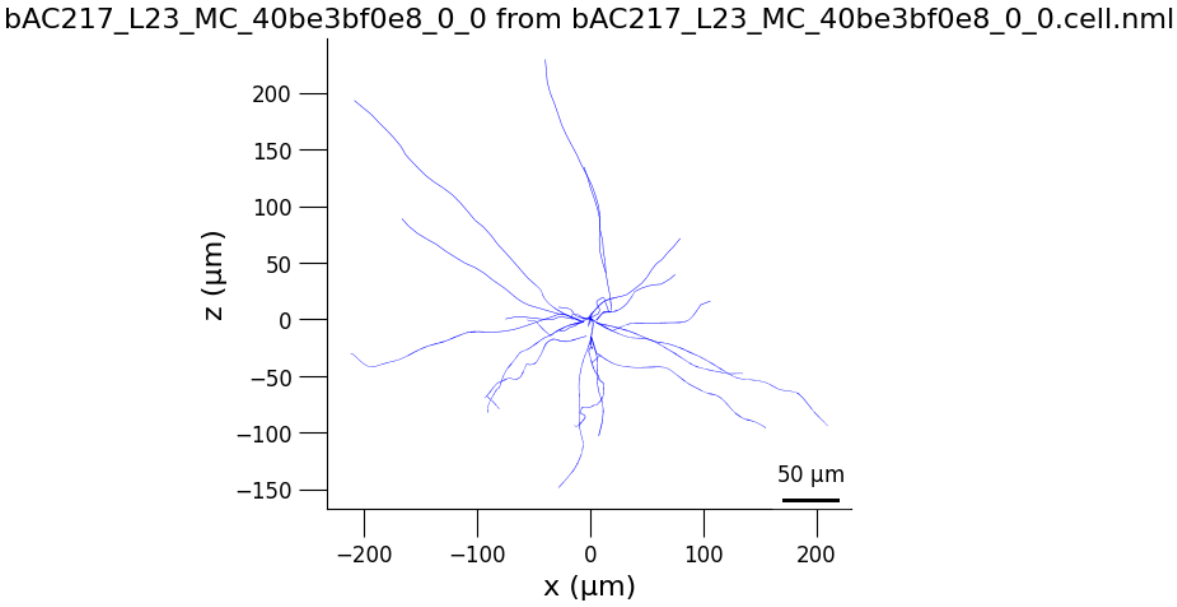
```
Saved as NMLCL000109.nml.zip
It's valid!
```

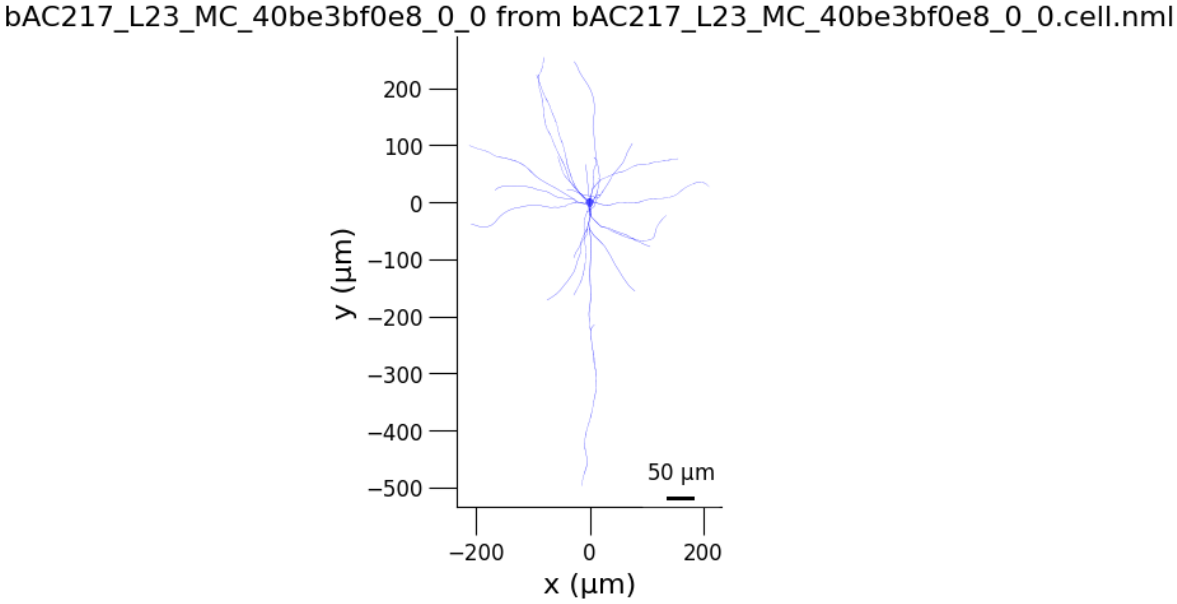
3.2) Generate 3D views of the cell

```
from pyneuroml.plot.PlotMorphology import plot_2D

planes = ['yz', 'xz', 'xy']
for plane in planes:

    plot_2D(detailed_cell_file,
            plane2d  = plane,
            min_width = 0,
            verbose= False,
            nogui = True,
            square=False)

detailed_cell_doc = pynml.read_neuroml2_file(detailed_cell_file)
detailed_cell = detailed_cell_doc.cells[0]
```

```
pyNeuroML >>> INFO - Loading NeuroML2 file: bAC217_L23_MC_40be3bf0e8_0_0.cell.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: bAC217_L23_MC_40be3bf0e8_0_0.cell.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: bAC217_L23_MC_40be3bf0e8_0_0.cell.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: bAC217_L23_MC_40be3bf0e8_0_0.cell.nml
```







3.3) Analyse cell spiking behaviour

```
martinotti_cell = detailed_cell_doc.cells[0]

generate_current_vs_frequency_curve(detailed_cell_file,
    martinotti_cell.id,
    start_amp_nA=-0.02,
    end_amp_nA=0.1,
    step_nA=0.02,
    pre_zero_pulse=100,
    post_zero_pulse=100,
    analysis_duration=1000,
```

(continues on next page)

(continued from previous page)

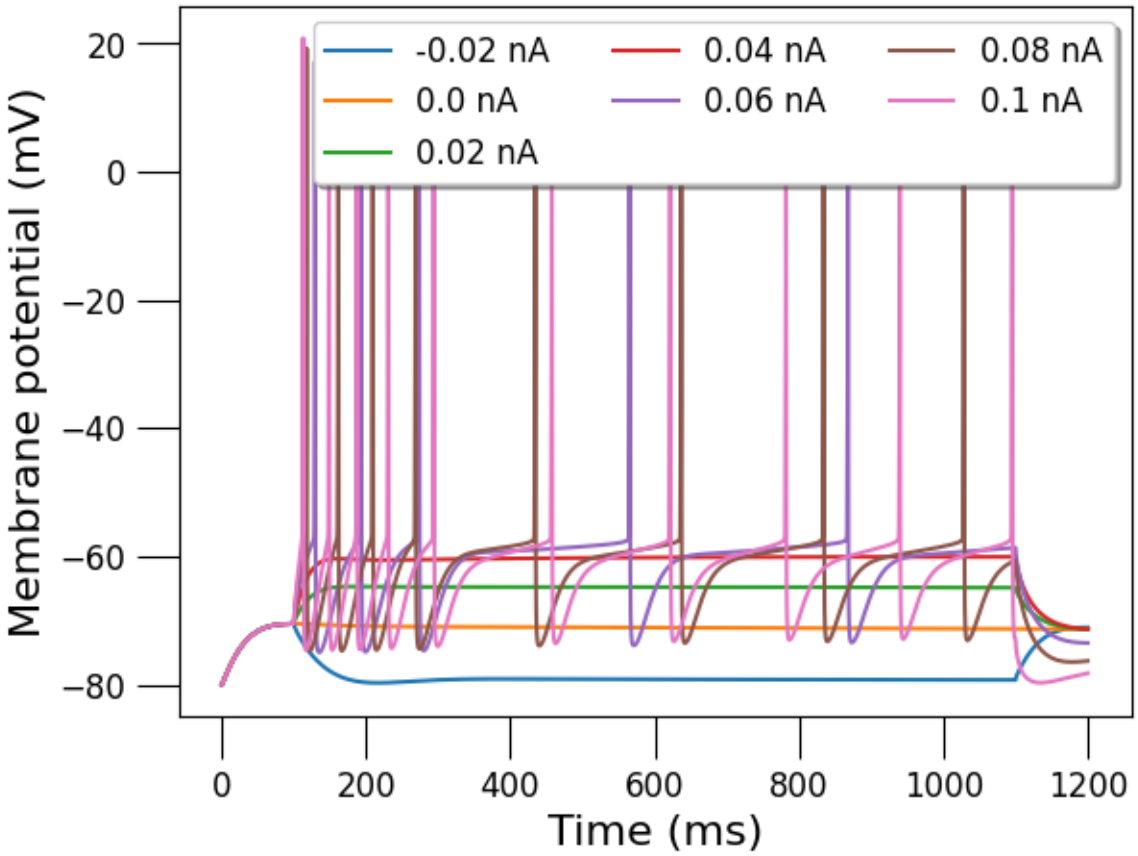
```
    plot_voltage_traces=True,
    simulator='jNeuroML_NEURON')
```

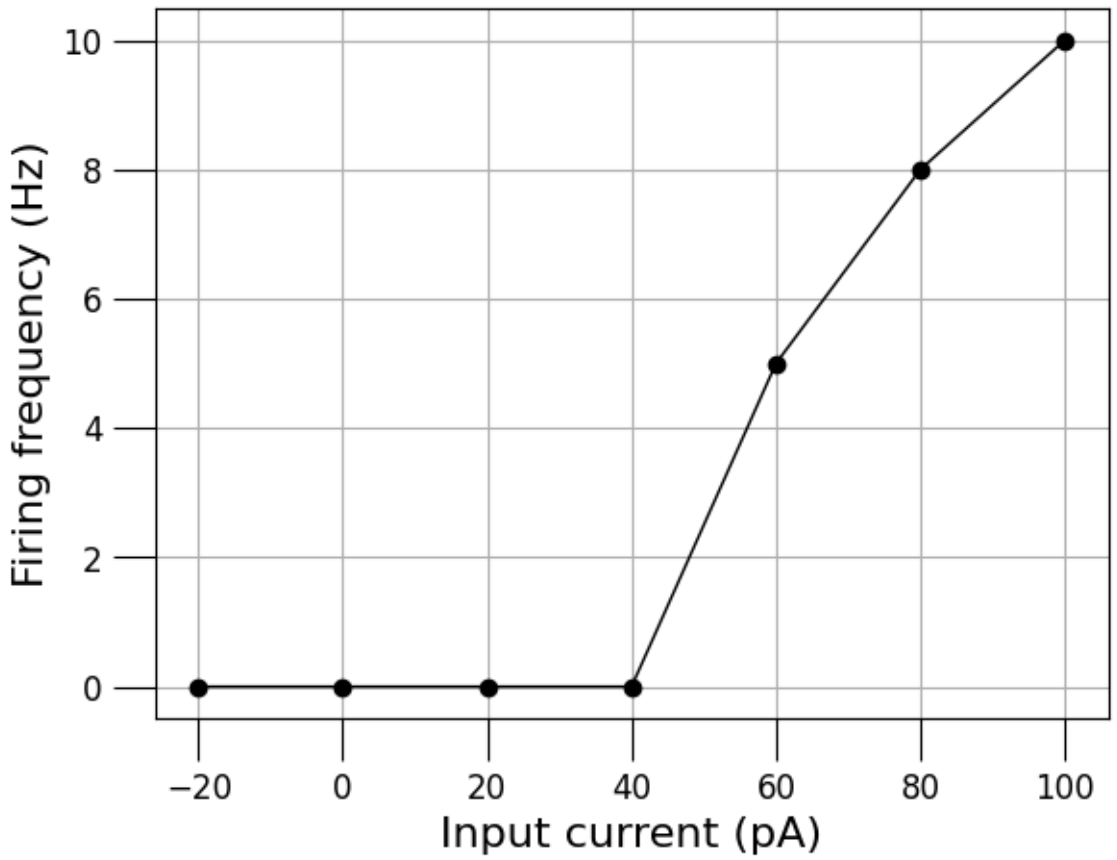
```
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/bAC217_L23_MC_40be3bf0e8_0_0.cell.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/K_Tst.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/SKv3_1.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/Nap_Et2.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/NaTs2_t.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/Ih.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/pas.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/Im.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/NaTa_t.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/Ca_LVAst.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/SK_E2.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/K_Pst.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/CaDynamics_E2_NML2.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/Ca.channel.nml
pyNeuroML >>> INFO - Executing: (java -Xmx400M  -jar  "/opt/homebrew/anaconda3/
 ↪envs/py39n/lib/python3.9/site-packages/pyneuroml/lib/jNeuroML-0.13.0-jar-with-
 ↪dependencies.jar" -validate iv_bAC217_L23_MC_40be3bf0e8_0_0.net.nml ) in␣
 ↪directory: .
pyNeuroML >>> INFO - Command completed successfully!
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/iv_bAC217_L23_MC_40be3bf0e8_0_0.net.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/bAC217_L23_MC_40be3bf0e8_0_0.cell.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/K_Tst.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/SKv3_1.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/Nap_Et2.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/NaTs2_t.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/Ih.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/pas.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/Im.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/NaTa_t.channel.nml
```

(continues on next page)

(continued from previous page)

```
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/Ca_LVAst.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/SK_E2.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/K_Pst.channel.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/CaDynamics_E2_NML2.nml
pyNeuroML >>> INFO - Loading NeuroML2 file: /Users/padraig/git/Documentation/
 ↪source/Userdocs/NML2_examples/Ca.channel.nml
pyNeuroML >>> INFO - Loading LEMS file: LEMS_iv_bAC217_L23_MC_40be3bf0e8_0_0.xml␣
 ↪and running with jNeuroML_NEURON
pyNeuroML >>> INFO - Executing: (java -Xmx400M  -Djava.awt.headless=true -jar  "/
 ↪opt/homebrew/anaconda3/envs/py39n/lib/python3.9/site-packages/pyneuroml/lib/
 ↪jNeuroML-0.13.0-jar-with-dependencies.jar"  LEMS_iv_bAC217_L23_MC_40be3bf0e8_0_0.
 ↪xml  -neuron -run -compile -nogui -I '') in directory: .
pyNeuroML >>> INFO - Command completed successfully!
/opt/homebrew/anaconda3/envs/py39n/lib/python3.9/site-packages/numpy/core/
 ↪fromnumeric.py:3432: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/opt/homebrew/anaconda3/envs/py39n/lib/python3.9/site-packages/numpy/core/_methods.
 ↪py:190: RuntimeWarning: invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
pyNeuroML >>> INFO - Generating plot: Membrane potential traces for: bAC217_L23_MC_
 ↪40be3bf0e8_0_0.cell.nml
pyNeuroML >>> INFO - Generating plot: Firing frequency versus injected current␣
 ↪for: bAC217_L23_MC_40be3bf0e8_0_0.cell.nml
```


-0.02 nA
0.0 nA
0.02 nA
0.04 nA
0.06 nA
0.08 nA
0.1 nA
Membrane potential (mV)
20
0
−20
−40
−60
−80
0
200
400
600
800
1000
1200
Time (ms)




```
{-0.02: 0.0, 0.0: 0.0, 0.02: 0.0, 0.04: 0.0, 0.06: 5.0, 0.08: 8.0, 0.1: 10.0}
```

4.9.4 4) Create a small network using the cells

4.1) Generate the NeuroML network

```
from neuroml.utils import component_factory
from pyneuroml import pynml
from pyneuroml.lems import LEMSSimulation
import neuroml.writers as writers
import random

nml_doc = component_factory("NeuroMLDocument", id="NML_DB_Net")


### Create the network
net = nml_doc.add("Network", id="NML_DB_Net", validate=False)
net.type="networkWithTemperature"
net.temperature="34.0degC"

### Add a synapse
syn0 = nml_doc.add(
    "ExpOneSynapse", id="syn0", gbase="65nS", erev="0mV", tau_decay="3ms"
)
```

(continues on next page)

(continued from previous page)

```
## Create the first population
size_exc = 4
nml_doc.add("IncludeType", href=cell_file)

pop_exc = component_factory("Population", id="Exc", component=cell.id, size=size_exc,␣
 ↪type="population")
# Set optional color property. Note: used later when generating plots
##pop0.add("Property", tag="color", value="0 0 .8")
net.add(pop_exc)

## Create the second population
size_inh = 4
nml_doc.add("IncludeType", href=detailed_cell_file)

pop_inh = component_factory("Population", id="Inh", component=detailed_cell.id,␣
 ↪size=size_inh, type="population")
# Set optional color property. Note: used later when generating plots
##pop1.add("Property", tag="color", value="0 0 .8")
net.add(pop_inh)

## Create connections and inputs
random.seed(123)
prob_connection = 0.8

proj_count = 0

projection = Projection(
                    id="Proj_exc_inh",
                    presynaptic_population=pop_exc.id,
                    postsynaptic_population=pop_inh.id,
                    synapse=syn0.id,
                )

net.projections.append(projection)

for i in range(0, size_exc):
    for j in range(0, size_inh):
        if random.random() <= prob_connection: # probablistic connection...
            connection = ConnectionWD(
                id=proj_count,
                pre_cell_id="%s[%i]" % (pop_exc.id, i),
                post_cell_id="%s[%i]" % (pop_inh.id, j),
                weight=random.random(),
                delay='0ms'
            )
            projection.add(connection)
            proj_count += 1

for i in range(0, size_exc):
    # pulse generator as explicit stimulus
    pg = nml_doc.add(
        "PulseGenerator",
        id="pg_exc_%i" % i,
        delay="20ms",
        duration="260ms",
```

(continues on next page)

(continued from previous page)

```
        amplitude="%f nA" % (0.03 + 0.01 * random.random()),
    )

    exp_input = net.add(
        "ExplicitInput", target="%s[%i]" % (pop_exc.id, i), input=pg.id
    )

for i in range(0, size_inh):
    # pulse generator as explicit stimulus
    pg = nml_doc.add(
        "PulseGenerator",
        id="pg_inh_%i"%i,
        delay="20ms",
        duration="260ms",
        amplitude="%f nA" % (0.02 + 0.02 * random.random()),
    )

    exp_input = net.add(
        "ExplicitInput", target="%s[%i]" % (pop_inh.id, i), input=pg.id
    )


print(nml_doc.summary())

nml_net_file = 'NML_DB_network.net.nml'
writers.NeuroMLWriter.write(nml_doc, nml_net_file)

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

```
pyNeuroML >>> INFO - Executing: (java -Xmx400M  -jar  "/opt/homebrew/anaconda3/
 ↪envs/py39n/lib/python3.9/site-packages/pyneuroml/lib/jNeuroML-0.13.0-jar-with-
 ↪dependencies.jar" -validate NML_DB_network.net.nml ) in directory: .
```

```
*******************************************************
* NeuroMLDocument: NML_DB_Net
*
*  ExpOneSynapse: ['syn0']
*  IncludeType: ['TestCell.cell.nml', 'bAC217_L23_MC_40be3bf0e8_0_0.cell.nml']
*  PulseGenerator: ['pg_exc_0', 'pg_exc_1', 'pg_exc_2', 'pg_exc_3', 'pg_inh_0',
 ↪'pg_inh_1', 'pg_inh_2', 'pg_inh_3']
*
*  Network: NML_DB_Net (temperature: 34.0degC)
*
*   8 cells in 2 populations
*     Population: Exc with 4 components of type novel_cell
*     Population: Inh with 4 components of type bAC217_L23_MC_40be3bf0e8_0_0
*
*   13 connections in 1 projections
*     Projection: Proj_exc_inh from Exc to Inh, synapse: syn0
*       13 connections (wd): [(Connection 0: 0 -> 0, weight: 0.087187, delay: 0.
 ↪00000 ms), ...]
*
*   0 inputs in 0 input lists
*
*   8 explicit inputs (outside of input lists)
```

(continues on next page)

(continued from previous page)

```
*     Explicit Input of type pg_exc_0 to Exc(cell 0), destination: unspecified
*     Explicit Input of type pg_exc_1 to Exc(cell 1), destination: unspecified
*     Explicit Input of type pg_exc_2 to Exc(cell 2), destination: unspecified
*     Explicit Input of type pg_exc_3 to Exc(cell 3), destination: unspecified
*     Explicit Input of type pg_inh_0 to Inh(cell 0), destination: unspecified
*     Explicit Input of type pg_inh_1 to Inh(cell 1), destination: unspecified
*     Explicit Input of type pg_inh_2 to Inh(cell 2), destination: unspecified
*     Explicit Input of type pg_inh_3 to Inh(cell 3), destination: unspecified
*
*********************************************************
Written network file to: NML_DB_network.net.nml
```

```
pyNeuroML >>> INFO - Command completed successfully!
pyNeuroML >>> INFO - Output:
  jNeuroML >>   jNeuroML v0.13.0
  jNeuroML >>  Validating: /Users/padraig/git/Documentation/source/Userdocs/NML2_
 ↪examples/NML_DB_network.net.nml
  jNeuroML >>  Valid against schema and all tests
  jNeuroML >>  No warnings
  jNeuroML >>
  jNeuroML >>  Validated 1 files: All valid and no warnings
  jNeuroML >>
  jNeuroML >>
```

```
True
```

4.2) Generate views of the network

```
from pyneuroml.pynml import generate_nmlgraph

generate_nmlgraph(nml_net_file, level=-1, engine="dot")

from IPython.display import Image
Image(filename='%s.gv.png'%net.id,width=500)
```

```
pyNeuroML >>> INFO - Converting NML_DB_network.net.nml to graphical form, level -1,
 ↪ engine dot
```

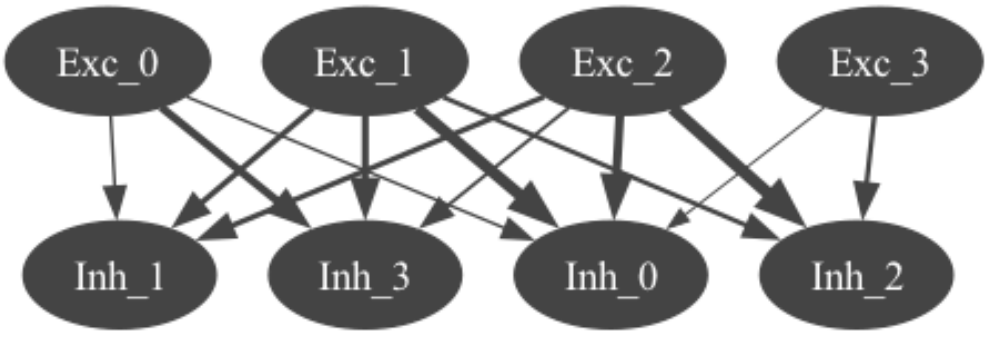
```
neuromllite >>> Initiating GraphViz handler, level -1, engine: dot, seed: 1234
Parsing: NML_DB_network.net.nml
Loaded: NML_DB_network.net.nml as NeuroMLDocument
neuromllite >>> Document: NML_DB_Net
neuromllite >>> Network: NML_DB_Net
neuromllite >>> Population: Exc, component: novel_cell (Cell), size: 4 cells,␣
 ↪properties: {}
neuromllite >>> Population: Inh, component: bAC217_L23_MC_40be3bf0e8_0_0 (Cell),␣
 ↪size: 4 cells, properties: {}
neuromllite >>> GRAPH PROJ: Proj_exc_inh (Exc (4) -> Inh (4), projection): w 1.0;␣
 ↪wtot: 4.849021209484878; sign: 1; cond: 65.0 nS (65nS); all: {'projection': 58.
 ↪85091239849243, 'inhibitory': -1e+100, 'excitatory': -1e+100,
 ↪'electricalProjection': -1e+100, 'continuousProjection': -1e+100} -> {'projection
 ↪': 1.380338363104285, 'inhibitory': 1e+100, 'excitatory': 1e+100,
 ↪'electricalProjection': 1e+100, 'continuousProjection': 1e+100}
```

(continues on next page)

(continued from previous page)

```
neuromllite >>>  - conn Exc_0 -> Inh_0: 5.66713403897108 (5.66713403897108)
neuromllite >>>  - conn Exc_0 -> Inh_1: 7.000515270998535 (7.000515270998535)
neuromllite >>>  - conn Exc_0 -> Inh_3: 34.85313260220526 (34.85313260220526)
neuromllite >>>  - conn Exc_1 -> Inh_0: 55.385630230408985 (55.385630230408985)
neuromllite >>>  - conn Exc_1 -> Inh_1: 21.91908271210294 (21.91908271210294)
neuromllite >>>  - conn Exc_1 -> Inh_2: 15.935617913644716 (15.935617913644716)
neuromllite >>>  - conn Exc_1 -> Inh_3: 28.357926571989168 (28.357926571989168)
neuromllite >>>  - conn Exc_2 -> Inh_0: 38.84396519171941 (38.84396519171941)
neuromllite >>>  - conn Exc_2 -> Inh_1: 20.504833292831094 (20.504833292831094)
neuromllite >>>  - conn Exc_2 -> Inh_2: 58.85091239849243 (58.85091239849243)
neuromllite >>>  - conn Exc_2 -> Inh_3: 9.241379188047565 (9.241379188047565)
neuromllite >>>  - conn Exc_3 -> Inh_0: 1.380338363104285 (1.380338363104285)
neuromllite >>>  - conn Exc_3 -> Inh_2: 17.245910842001596 (17.245910842001596)
neuromllite >>> Generating graph for: NML_DB_Net
neuromllite >>> ************************************
neuromllite >>> * Settings for GraphVizHandler:
neuromllite >>> *
neuromllite >>> *    engine:                  dot
neuromllite >>> *    level:                   -1
neuromllite >>> *    is_cell_level:           True
neuromllite >>> *    CUTOFF_INH_SYN_MV:       -50
neuromllite >>> *    include_ext_inputs:      True
neuromllite >>> *    include_input_pops:      True
neuromllite >>> *    scale_by_post_pop_size:  True
neuromllite >>> *    scale_by_post_pop_cond:  True
neuromllite >>> *    min_weight_to_show:      0
neuromllite >>> *    show_chem_conns:         True
neuromllite >>> *    show_elect_conns:        True
neuromllite >>> *    show_cont_conns:         True
neuromllite >>> *    output_format:           png
neuromllite >>> *
neuromllite >>> * Used values:
neuromllite >>> *    syn_conds_used:          syn0:         65.0 nS (65nS)
neuromllite >>> *
neuromllite >>> ************************************
```

```
/opt/homebrew/anaconda3/envs/py39n/lib/python3.9/subprocess.py:1052:␣
 ↪ResourceWarning: subprocess 25497 is still running
  _warn("subprocess %s is still running" % self.pid,
ResourceWarning: Enable tracemalloc to get the object allocation traceback
pyNeuroML >>> INFO - Done with GraphViz...
```



4.3) Generate LEMS file to simulate the network

See documentation about LEMS Simulation files.

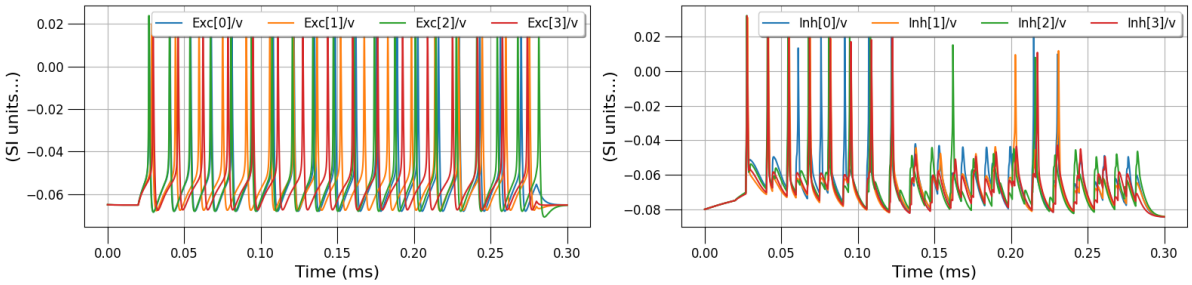
```
simulation_id = "NML_DB_network_sim"
simulation = LEMSSimulation(sim_id=simulation_id,
                            duration=300, dt=0.025, simulation_seed=123)
simulation.assign_simulation_target(net.id)
simulation.include_neuroml2_file(nml_net_file, include_included=False)


pops = [pop_exc, pop_inh]

for pop in pops:
    simulation.create_event_output_file(
        f"{pop.id}_spikes", f"{pop.id}.spikes", format='ID_TIME'
    )
    simulation.create_output_file(pop.id, "%s.v.dat" % pop.id)

    for pre in range(0, pop.size):
        next_cell = '{}[{}]'.format(pop.id,pre)
        simulation.add_selection_to_event_output_file(
            f"{pop.id}_spikes", pop.component, next_cell, 'spike')

        simulation.add_column_to_output_file(pop.id, f"v{pre}", f"{next_cell}/v")


lems_simulation_file = simulation.save_to_file()

print(f"Saved LEMS Simulation file to: {lems_simulation_file}")
```

```
Saved LEMS Simulation file to: LEMS_NML_DB_network_sim.xml
```

4.4) Run simulation of the network using NEURON

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

```
pyNeuroML >>> INFO - Loading LEMS file: LEMS_NML_DB_network_sim.xml and running␣
 ↪with jNeuroML_NEURON
pyNeuroML >>> INFO - Executing: (java -Xmx2G  -Djava.awt.headless=true -jar  "/opt/
 ↪homebrew/anaconda3/envs/py39n/lib/python3.9/site-packages/pyneuroml/lib/jNeuroML-
 ↪0.13.0-jar-with-dependencies.jar"  LEMS_NML_DB_network_sim.xml  -neuron -run -
 ↪compile -nogui -I '') in directory: .
pyNeuroML >>> INFO - Command completed successfully!
pyNeuroML >>> WARNING - Reloading: Data loaded from ./Exc.v.dat (jNeuroML_NEURON)
pyNeuroML >>> WARNING - Reloading: Data loaded from ./Inh.v.dat (jNeuroML_NEURON)
```


Exc[0]/v
Exc[1]/v
Exc[2]/v
Exc[3]/v
(SI units...)
Time (ms)
Inh[0]/v
Inh[1]/v
Inh[2]/v
Inh[3]/v
(SI units...)
Time (ms)

CHAPTER

FIVE

FINDING AND SHARING NEUROML MODELS

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

5.1 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.

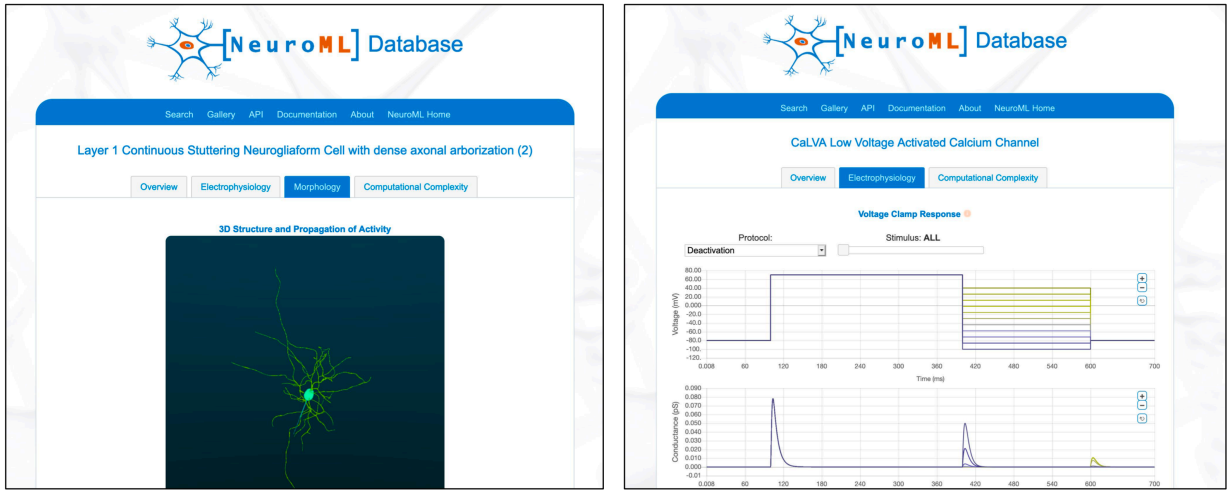


Fig. 5.1: The NeuroML Database contains NeuroML files for many cells (left above), channels (right) and synapses taken from Open Source Brain, Blue Brain Project, Allen Institute and more.

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.

5.2 Open Source Brain

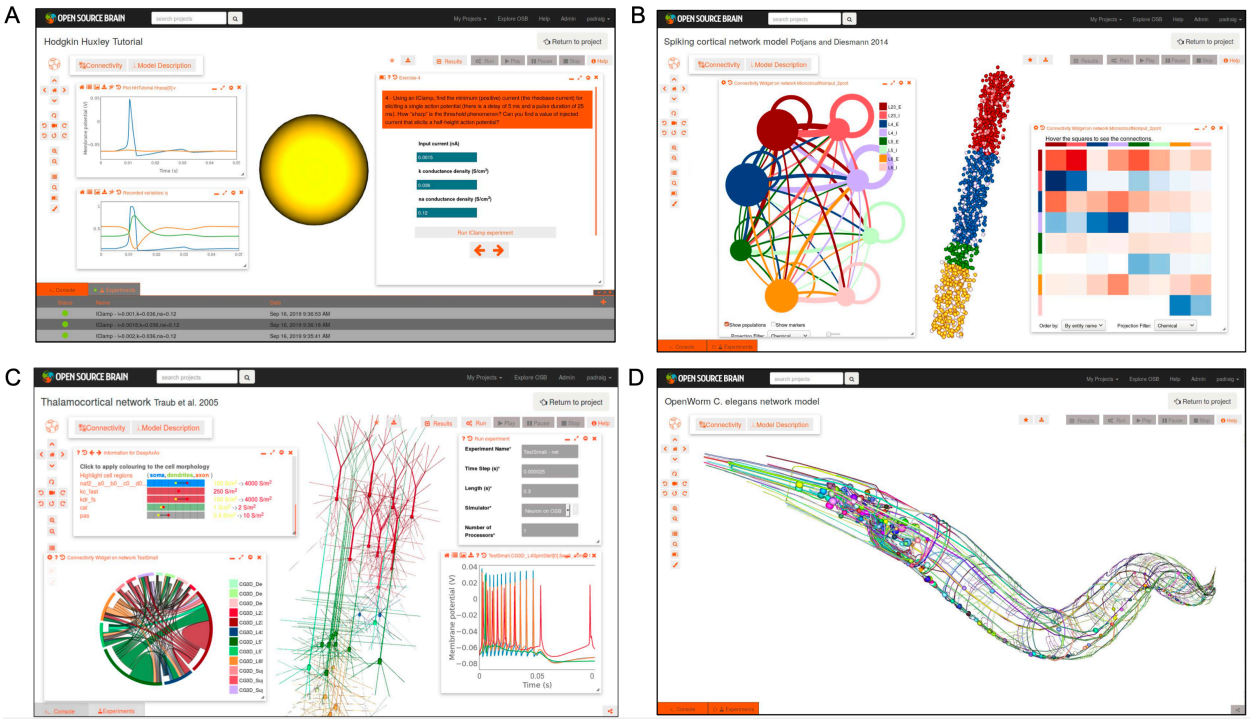


Fig. 5.2: Examples of NeuroML 2 models visualised on Open Source Brain. A) Hodgkin Huxley model interactive tutorial. B) Integrate and fire network model of cortical column (Potjans and Diesmann 2014), showing network connectivity. C) Cortical model with multicompartmental cells (Traub et al. 2005), 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, analysing, and simulating standardized models from different brain regions and species. An index of various NeuroML models on Open Source Brain and their validation status can be seen here.

To add your NeuroML model to Open Source Brain, please see the information on *this page*.

5.3 Other related projects

Note

Needs introductory text.

5.4 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.

5.5 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.

5.6 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.

5.7 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.

CHAPTER

SIX

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 next 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.

6.1 Converting models to NeuroML and sharing them on Open Source Brain

Walk throughs available

Look at the *Walkthroughs chapter* for worked examples.

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.

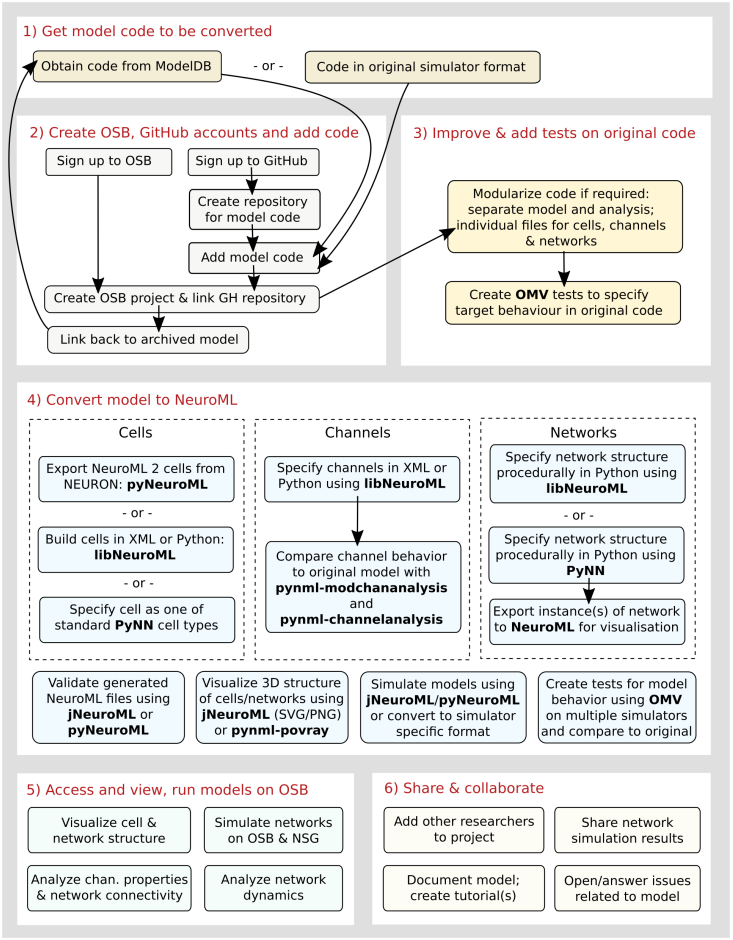


Fig. 6.1: Procedures and tools to convert models from native formats to NeuroML and PyNN (Taken from Gleeson et al. 2019 [GCM+19])

6.1.1 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 is invaluable. The original code allows the identification of all parameters related to the model, and it is required to verify the dynamical behaviour of the NeuroML equivalent.

Scripts for an increasing number of published models are available on ModelDB. ModelDB models are also published as GitHub repositories. Forks of these are also managed in the Open Source Brain GitHub organization and indexed on Open Source Brain version 2.

So, the first step is to obtain the original model code and verify that this can be run to reproduce the published results.

6.1.2 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.

6.1.3 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).

6.1.4 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 passibe version inthe 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-compartmental 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 agains 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.

6.1.5 4e) (Re)optimising cell models

You can use Neurotune inside pyNeuroML to re-optimise your cell models. An example is here, and a full sequence of optimising a NeuroML model against data in NWB can be found *here*.

6.1.6 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*.

6.1.7 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.

6.1.8 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).

6.2 Handling Morphology Files

A number of formats are used in neuroscience to encode neuronal morphologies obtained from experiments involving neuronal reconstructions. This page provides general information on these formats, and documents how they may be converted to NeuroML 2 for use in computational models.

6.2.1 Terminology

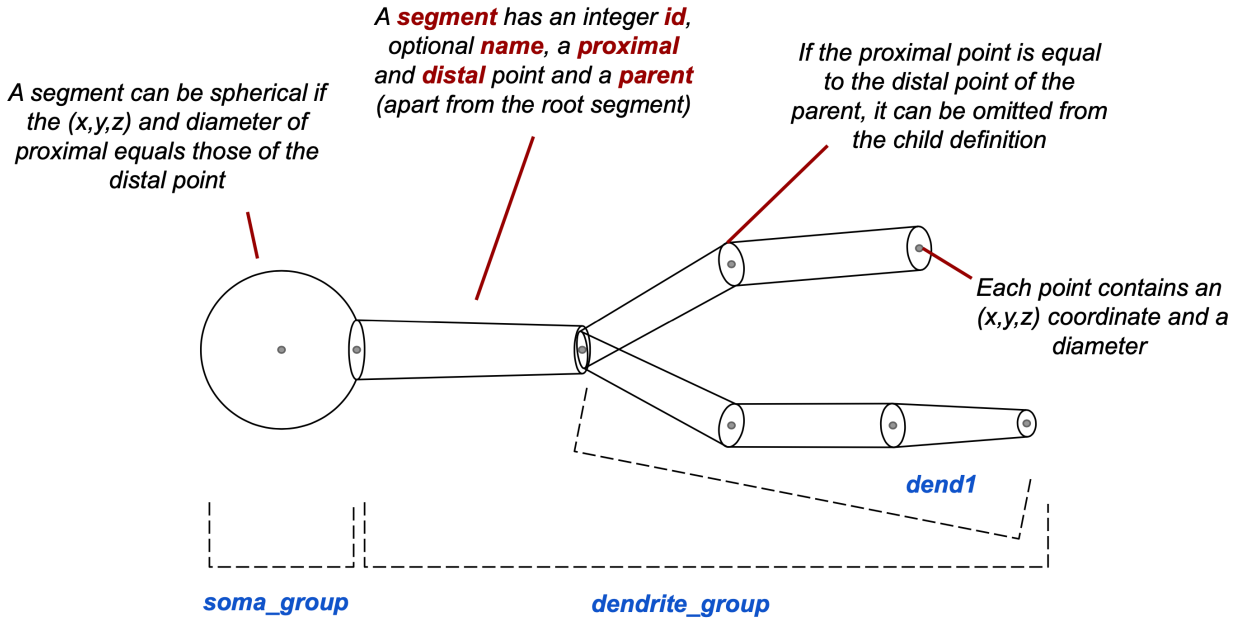


Fig. 6.2: Specification of morphologies in **NeuroML 2**. More details can be found for each element in the specification, e.g. *<cell>*, *<morphology>*, **, *<segmentGroup>*, *<proximal>*, *<distal>*.

All formats have their own terminology that is used to refer to different parts of the cell.

In NEURON:

- a `section` is an unbranched contiguous cell region
- the morphology of a cell is defined by 3D points, `pt3D`
- for simulation, one can specify how many segments a section should be divided into, given by `nseg`

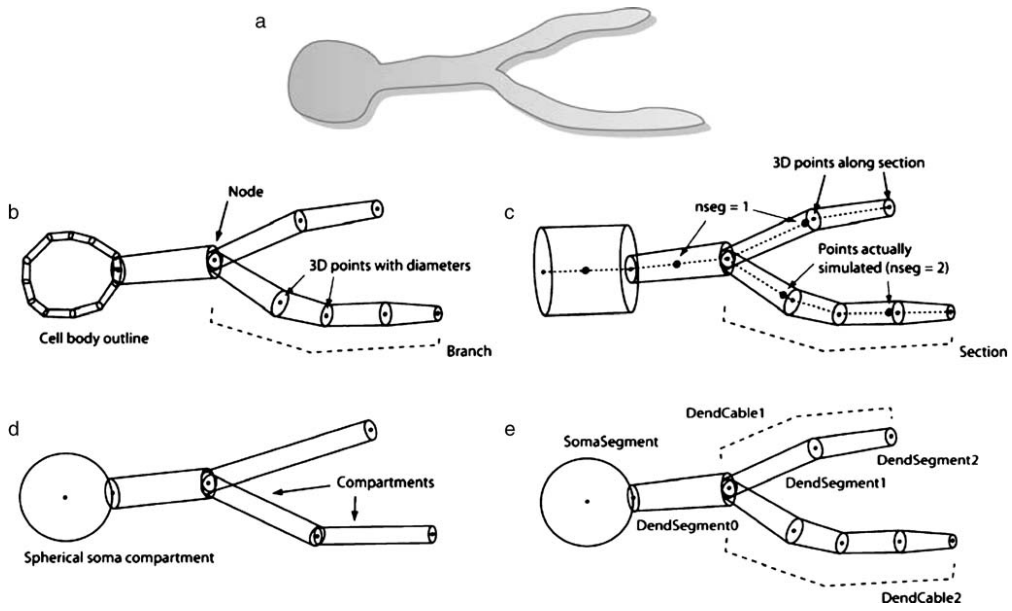


Fig. 6.3: Figure 1 from [CGH+07], a schematic comparing handling of morphological information for a simple cell by different applications. a) Schematic of biological cell structure to be represented. b) **Neurolucida reconstruction** where the soma is represented by an outline and three-dimensional points are specified along each branch. c) **NEURON simulator** format where cell structure is specified in *sections*. Only the center of the section is simulated unless the nseg parameter is greater than one. d) **GENESIS simulator representation** using compartments that are cylinders except for the soma, which can be spherical. The optimal length of each compartment is determined by the electrotonic length. e) **MorphML representation** (i.e. NeuroML v1) where any of the information shown in panels b through d can be encoded.

In NeuroML:

- segments are 3D points describing the cell morphology
- continuous, unbranched segments groups, would form a section
- the `numberInternalDivisions` property can be used to set the number of divisions a segment or segment group should be divided into for simulation

6.2.2 Cell validity

In general, it is usually necessary to examine NeuroML cells converted from various formats, especially experimental reconstructions, before they can be used in simulations. This is because reconstructions may not always contain all the information necessary to simulate the cell.

Two potential problems that must be checked are:

- Point of connection of dendritic branches to the soma: e.g., in Neurolucida, there is no explicit soma but usually only an outline.
- Zero length sections: NEURON can work with zero segment lengths (consecutive `pt3d` points being equal), but a standard mapping of this may not be supported in other simulators such as GENESIS.

An incomplete list of checks to make to ensure a valid cell is (taken from *neuroConstruct*):

- Only one segment should be without a parent (root)
- All segments must have sections
- All segments must have endpoints
- All segments must have unique IDs
- All segments must have unique names
- All sections must have unique names
- Segments after the first in a section must only be connected to 1 parent
- Only one segment may be spherical and must belong to the `soma_group` SegmentGroup
- The cell must have at least one segment
- The cell must have at least one soma section, i.e., which is in the `soma_group`
- The cell must have a cell name

The NeuroML validation tools will check for some of these and report errors where possible.

6.2.3 Formats

NeuroML2

In NeuroML, morphologies are encapsulated in the *morphology* modelling element. A morphology includes *segments* and *segments groups*, and these can be used to refer to parts of the cell's morphology, for example, when placing ionic conductances. A number of conventions for use in morphologies are listed *here*.

Morphologies can be stored in external files

Requires jNeuroML v0.13.2, pyNeuroML v1.3.2

The functionality to store morphology information in external files was implemented in jNeuroML v0.13.2, and pyNeuroML v1.3.2. Please ensure you are using these or newer versions to use this feature.

Usually, morphologies are embedded in NeuroML cell definition files, *for example*:

```
    <cell id="pyr_soma_m_in_b_in">
    <!-- ... -->

        <morphology id="morph0">

            <segment >
            <!-- more segments and segment groups -->

        </morphology>

        <biophysicalProperties id="biophys1">
            <!-- biophysical properties contents -->
        </biophysicalProperties>

    </cell>
```

However, morphologies (and *biophysical properties*) can also be stored as "standalone" entities outside the cell definition and referred to. Further, they can also be stored in external files that may be "included" in the cell definition file (using the IncludeType model element). This allows the re-use of morphology and biophysical properties in multiple cell models:

```
    <cell id="pyr_soma_m_out_b_out" morphology="morph0" biophysicalProperties=
↪"biophys1">
        <!-- cell contents without morphology and biophysical properties -->
    </cell>

    <!-- Potentially in other files... -->

    <morphology id="morph0">

        <segment >
        <!-- more segments and segment groups -->

    </morphology>

    <biophysicalProperties id="biophys1">
        <!-- biophysical properties contents -->
    </biophysicalProperties>
```

NEURON

There is no fixed format in NEURON for specifying morphologies. However, cells created in NEURON may be exported to NeuroML2 format using the `export_to_neuroml2` method included in *pyNeuroML* (example).

GENESIS

The format for a GENESIS cell description is given here.

CVApp/SWC files

Please see this *page*.

Neurolucida

The Neurolucida file format is used by MicroBrightField products to store information on neuronal reconstructions. Both binary and ASCII format files can be generated by these products. The format allows recording of various anatomical features, not only neuronal processes such as dendrites and cell bodies, but can record other micro-anatomical features of potential interest to anatomists. Not all of these features will be relevant when constructing a single cell computational model.

6.2.4 Tools

CVApp

The standalone CVApp tool provides an interface to visualize SWC files and export them into NeuroML2. For more information, please see this *page*.

neuroConstruct

neuroConstruct includes functionality to interactively import GENESIS, NEURON, CVapp (SWC), Neurolucida, and older MorphML formats to NeuroML2. Please see the neuroConstruct documentation for more information.

pyNeuroML

pyNeuroML includes functionality to convert NEURON files into NeuroML using the `export_to_neuroml2` method included in *pyNeuroML* (example).

6.3 HDF5 support

The XML serializations of large NeuroML models can be prohibitive to store. For such cases, NeuroML also includes support for saving models in the binary HDF5 format via the NeuroMLHdf5Writer in libNeuroML. The same format can be exported also from the Java API (example).

The format of the export is documented below:

- *Network* is exported as a `network` HDF5 group with `id`, `notes`, and the `temperature` (optional) stored as attributes.

- *Population* is exported as a group with id `population_<id of the population>` with `id`, `component`, `size`, `type`, and `property` tags stored as attributes.
  - If the population is a *population list* that includes *instances* of cells, the locations of cells (x, y, z), these are stored in a 3 column table ("chunked array") with a row per instance.
- *Projection* is exported as a group with id `project_<id of the projection>` with `id`, `type`, `presynapticPopulation`, `postSynapticPopulation`, `synapse` as attributes.
  - *Connection* and *ConnectionWD* elements in projections are stored as rows in a table with the first two columns as the `pre_cell_id` and `post_cell_id` respectively, and the successive columns for the necessary attributes.
- *ElectricalProjection* is exported similar to Projection with the *ElectricalConnection*, *ElectricalConnectionInstance*, and *ElectricalConnectionInstanceW* entries stored in tables.
- *ContinuousProjection* is exported similar to Projection with the *ContinuousConnection*, *ContinuousConnectionInstance*, and *ContinuousConnectionInstanceW* entries stored in tables.
- *InputList* is exported similar to Projection with the *Input*, and *InputW* entries stored in tables.

For more details, the source code of these export functions can be seen here in the libNeuroML repository and here in org.neuroml.model.

HDF5 NeuroML files can be read and processed by `jnml` and `pynml` in the same way as XML files (see here for LEMS Simulation file examples which reference HDF5 NeuroML models).

6.4 Maintaining provenance in NeuroML models

It is important to include metadata related to the "provenance" of model elements in NeuroML models. This ensures that the sources of different components of different models can be easily obtained/verified. For example, all NeuroML models should ideally include information about the original implementations, original research papers, and the original creators for all model elements.

NeuroML supports the inclusion of such information in the *annotation* model element. Annotations in NeuroML can include metadata using the Resource Description Framework (RDF). The advantage of using RDF is that it makes the metadata machine readable which enables automated analysis and parsing.

6.4.1 Annotation specification

NeuroML follows the COMBINE specification for annotations. This specification provides a set of well defined relationships that can be used to connect model elements to various metadata. These are included in the NeuroML schema as *core NeuroML component types*.

The suggested format for inclusion of the RDF metadata in annotations is Minimum information required in the annotation of models (MIRIAM).

6.4.2 Creating annotations

pyNeuroML includes the create_annotation function for the addition of annotations to model elements. Please see the API documentation for explanation on its usage. Note that this functionality is not included by default when installing pyNeuroML. One must install the annotation extra:

```
pip install pyneuroml[annotations]
```

An example of an annotation for the NeuroML conversion of Ray et al 2020 [RAS20] is shown below:

```
    annotation = create_annotation(
        subject=cell.id, title="Giant GABAergic Neuron model",
        description="Subhasis Ray, Zane N Aldworth, Mark A Stopfer (2020) Feedback␣
↪inhibition and its control in an insect olfactory circuit eLife 9:e53281.",
        annotation_style="miriam",
        serialization_format="pretty-xml",
        xml_header=False,
        citations={"https://doi.org/10.7554/eLife.53281": "Subhasis Ray, Zane N␣
↪Aldworth, Mark A Stopfer (2020) Feedback inhibition and its control in an insect␣
↪olfactory circuit eLife 9:e53281."},
        sources={"https://modeldb.science/262670": "ModelDB",
                 "https://github.com/OpenSourceBrain/262670": "GitHub",
                 "https://v1.opensourcebrain.org/projects/locust-mushroom-body":
↪"Open Source Brain"},
        authors={"Subhasis Ray":
                 {"https://orcid.org/0000-0003-2566-7146": "orcid"},
                 },
        contributors={
            "Ankur Sinha": {"https://orcid.org/0000-0001-7568-7167": "orcid"},
        },
        creation_date="2024-04-25"
    )
    cell.annotation = neuroml.Annotation([annotation])
```

This generates the following RDF annotation in the MIRIAM format to be included in the NeuroML cell file:

```
    <annotation>
        <rdf:RDF
          xmlns:dc="http://purl.org/dc/elements/1.1/"
          xmlns:dcterms="http://purl.org/dc/terms/"
          xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          xmlns:bqmodel="http://biomodels.net/model-qualifiers/"
        >
          <rdf:Description rdf:about="GGN">
            <dc:title>Giant GABAergic Neuron model</dc:title>
            <dc:description>Subhasis Ray, Zane N Aldworth, Mark A Stopfer (2020)␣
↪Feedback inhibition and its control in an insect olfactory circuit eLife 9:e53281.</
↪dc:description>
            <dc:source>
              <rdf:Bag>
                <rdf:_1 rdf:resource="https://modeldb.science/262670"/>
              </rdf:Bag>
            </dc:source>
            <dc:source>
              <rdf:Bag>
                <rdf:_1 rdf:resource="https://github.com/OpenSourceBrain/262670"/>
              </rdf:Bag>
            </dc:source>
```

(continues on next page)

(continued from previous page)

```
            <dc:source>
              <rdf:Bag>
                <rdf:_1 rdf:resource="https://v1.opensourcebrain.org/projects/locust-
↪mushroom-body"/>
              </rdf:Bag>
            </dc:source>
            <bqmodel:isDescribedBy>
              <rdf:Bag>
                <rdf:_1 rdf:resource="https://doi.org/10.7554/eLife.53281"/>
              </rdf:Bag>
            </bqmodel:isDescribedBy>
            <dc:creator>
              <rdf:Bag>
                <rdf:_1>Subhasis Ray</rdf:_1>
                <rdf:_2>https://orcid.org/0000-0003-2566-7146</rdf:_2>
              </rdf:Bag>
            </dc:creator>
            <dc:contributor>
              <rdf:Bag>
                <rdf:_1>Ankur Sinha</rdf:_1>
                <rdf:_2>https://orcid.org/0000-0001-7568-7167</rdf:_2>
              </rdf:Bag>
            </dc:contributor>
            <dcterms:created>
              <rdf:Description>
                <dcterms:W3CDTF>2024-04-25</dcterms:W3CDTF>
              </rdf:Description>
            </dcterms:created>
          </rdf:Description>
        </rdf:RDF>

    </annotation>
```

pyNeuroML also includes functions to extract data from annotations.

CHAPTER

SEVEN

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.

7.1 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
```

7.2 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.

7.3 List of validation tests

These tests are made against the Schema document.

Test	Description
Check names	Check that names of all elements, attributes, parameters match those provided in the schema
Check types	Check that the types of all included elements
Check values	Check that values follow given restrictions
Check inclusion	Check that required elements are included
Check cardinality	Check the number of elements
Check hierarchy	Check that child/children elements are included in the correct parent elements
Check sequence order	Check that child/children elements are included in the correct order

These are additional validation tests that are run on models (defined here):

Test	Description
Check top level ids	Check that top level (root) elements have unique ids
Check *Network* level ids	Check that child/children of the *Network* element have unique ids
Check *Cell Segment* ids	Check that all *Segment*s in a *Cell* have unique ids
Check single *Segment* without parent	Check that only one *Segment* is without parents (the soma *Segment*)
Check *SegmentGroup* ids	Check that all *SegmentGroup*s in a *Cell* have unique ids
Check *Member* segment ids exist	Check that *Segment*s referred to in *SegmentGroup Member*s exist
Check *SegmentGroup* definition	Check that *SegmentGroup*s being referenced are defined
Check *SegmentGroup* definition order	Check that *SegmentGroup*s are defined before being referenced
Check included *SegmentGroup*s	Check that *SegmentGroup*s referenced by *Include* elements of other *SegmentGroup*s exist
Check `numberInternalDivisions`	Check that *SegmentGroup*s define `numberInternalDivisions` (used by simulators to discretize un-branched branches into compartments for simulation)
Check included model files	Check that model files included by other files exist
Check *Population* component	Check that a component id provided to a *Population* exists
Check ion channel exists	Check that an ion channel used to define a *ChannelDensity* element exists
Check concentration model species	Check that the species used in *ConcentrationModel* elements are defined
Check *Population* size	Check that the `size` attribute of a *PopulationList* matches the number of defined *Instance*s
Check *Projection* component	Check that *Population*s used in the *Projection* elements exist
Check *Connection Segment*	Check that the *Segment* used in *Connection* elements exist
Check *Connection* pre/post cells	Check that the pre- and post-synaptic cells used in *Connection* elements exist and are correctly specified
Check *Synapse*	Check that the *Synapse* component used in a *Projection* element exists
Check root id	Check that the root *Segment* in a *Cell* morphology has id (0)

CHAPTER

EIGHT

VISUALISING NEUROML MODELS

A number of the *NeuroML software tools* can be used to easily visualise models described in NeuroML.

8.1 Get a quick summary of your model

8.1.1 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: pyramidals_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 pyramidals_48, synapse: GABA_
↪syn
*       336 connections: [(Connection 0: 10:0(0.05824) -> 0:6(0.02628)), ...]
*     Projection: SmallNet_pyr_bask from pyramidals_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 pyramidals_48 to pyramidals_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 pyramidals_48, component␣
↪BackgroundRandomIClamps
*       14 inputs: [(Input 0: 37:0(0.500000)), ...]
*
******************************************************
```

8.1.2 Using pyNeuroML

You can also get a summary of your model from within your *pyNeuroML* script itself using the `summary` function:

```
import pyneuroml.pynml

...


pyneuroml.pynml.summary(nml2_doc)
```

8.2 View the 3D structure of your model

8.2.1 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 :

An visualiser is also included in pyneuroml as `pynml-plotmorph` which includes both 2D and 3D views:

```
Usage:
pynml-plotmorph <NeuroML file>
pynml-plotmorph -i <NeuroML file>
```

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:

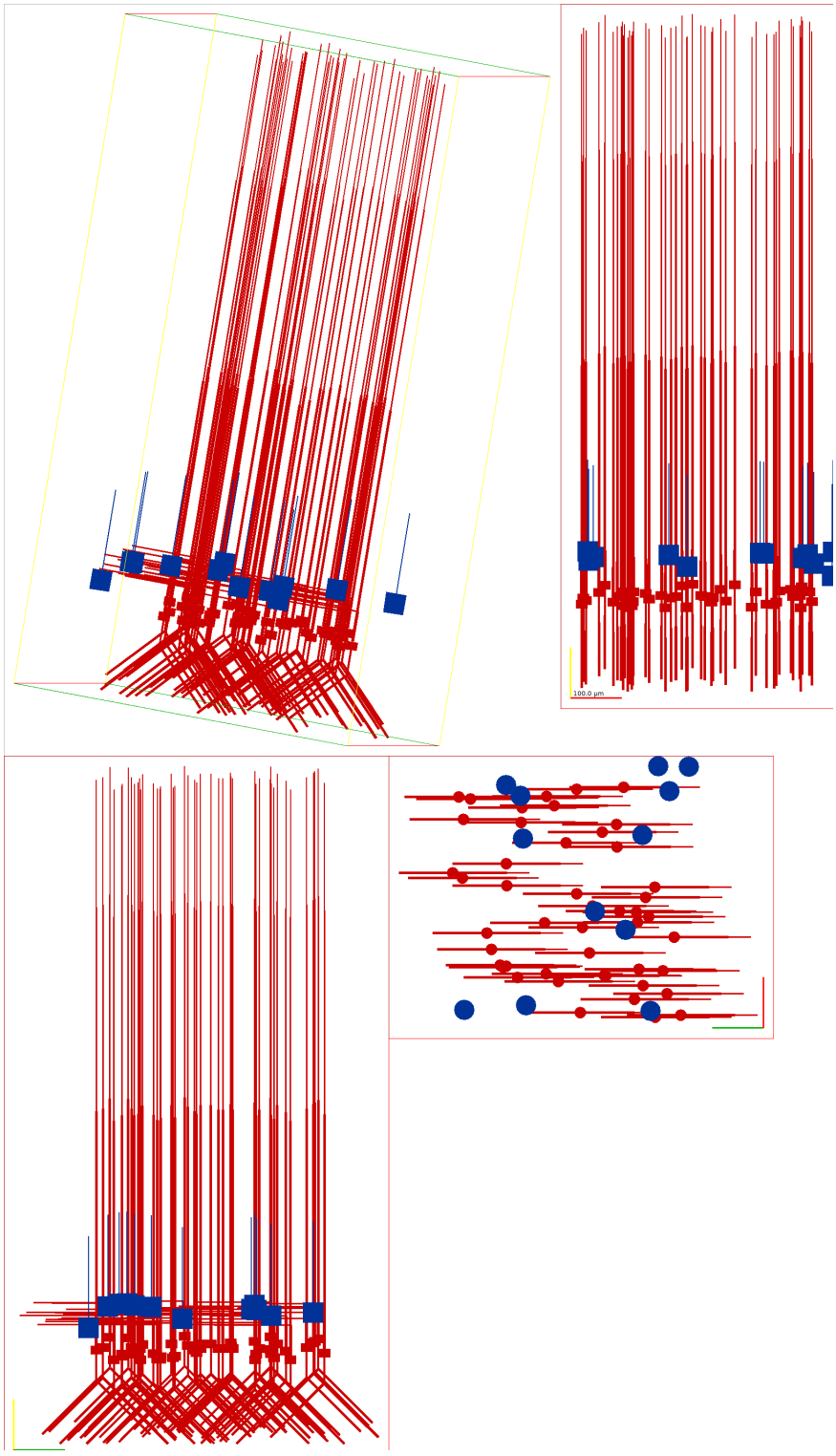


Fig. 8.1: Graphical view of the Auditory Cortex model generated with pynml

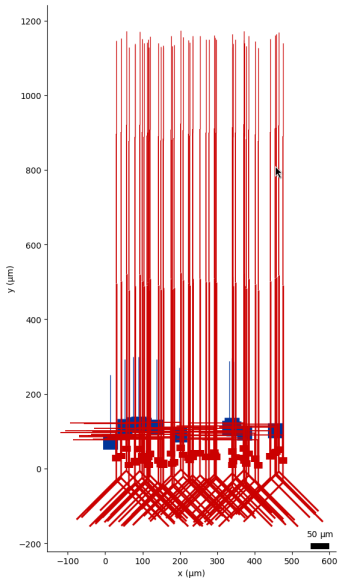


Fig. 8.2: Matplotlib based 2D visualisation of a network with `pynml-plotmorph`.

```
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:

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
```

8.2.2 Using pyNeuroML

These functions are also exposed as Python functions in *pyNeuroML*, so that you can use them directly in Python scripts:

```
import pyneuroml.pynml

pyneuroml.pynml.nml2_to_png(nml2_doc)
pyneuroml.pynml.nml2_to_svg(nml2_doc)


from pyneuroml.plot.PlotMorphology import plot_2D
from pyneuroml.plot.PlotMorphologyVispy import plot_interactive_3D

plot_2D(nml2_doc)
plot_interactive_3D(nml2_doc)
```

Open Source Brain uses NeuroML.

The Open Source Brain platform generates the interactive visualisations from NeuroML sources. See the Auditory

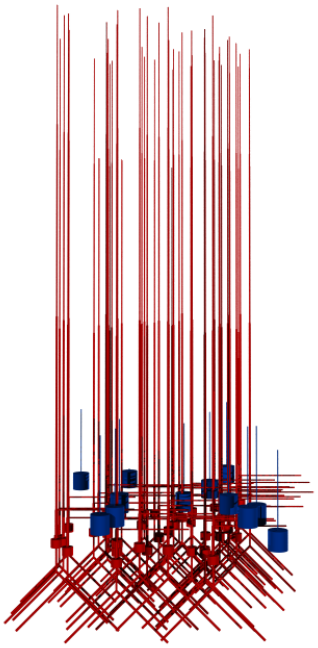


Fig. 8.3: Graphical view of the Auditory Cortex model generated with pynml-povray and POV-Ray

Cortex model on Open Source Brain here.

8.3 View the connectivity graph of your model

8.3.1 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 :



Fig. 8.4: Level 1 network graph generated by pynml

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
```

8.3.2 Using pyNeuroML

You can also generated these figures from within your *pyNeuroML* script itself using the `generate_nmlgraph` function:

```
import pyneuroml.pynml

...


pyneuroml.pynml.generate_nmlgraph(nml2_doc, level="1", engine="dot")
```

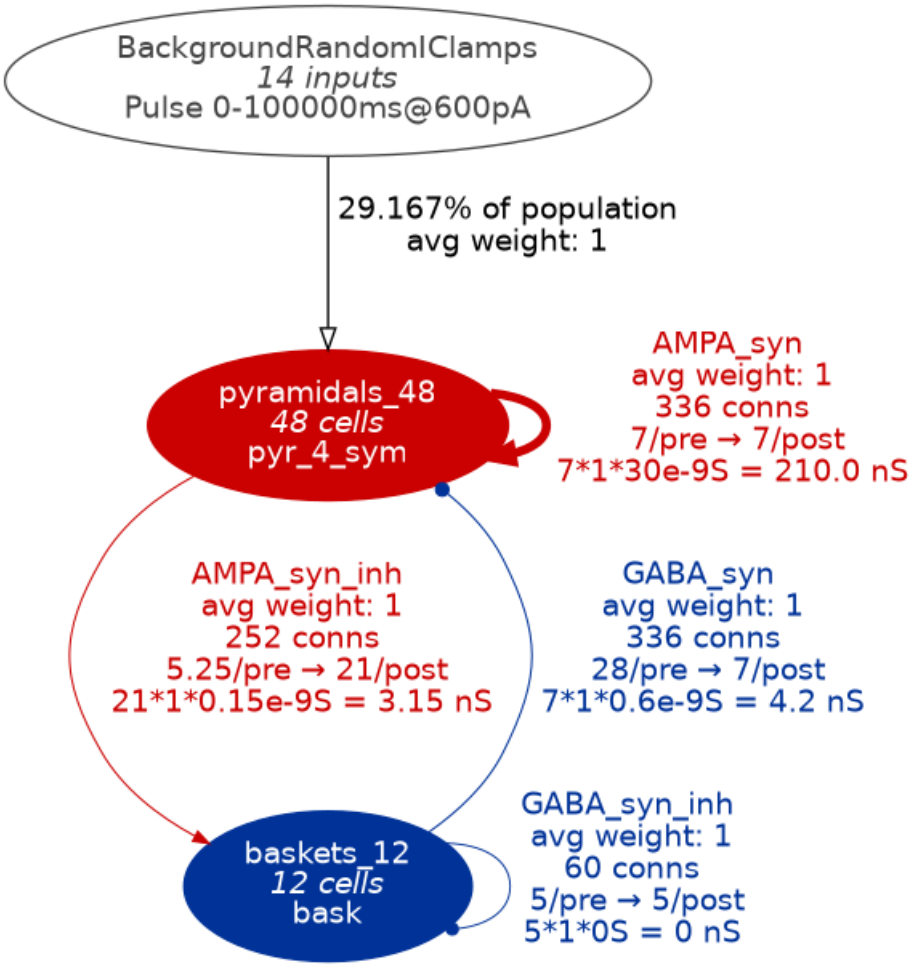


Fig. 8.5: Level 5 network graph generated by pynml

8.4 View the connectivity matrices of the model

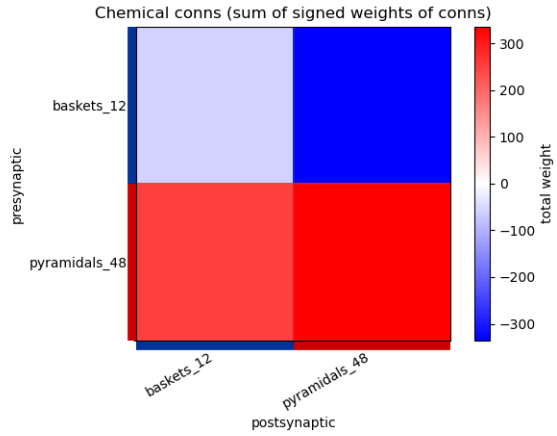
You can generate the connectivity matrices of projections between neuronal populations of the NeuroML model using `pynml`:

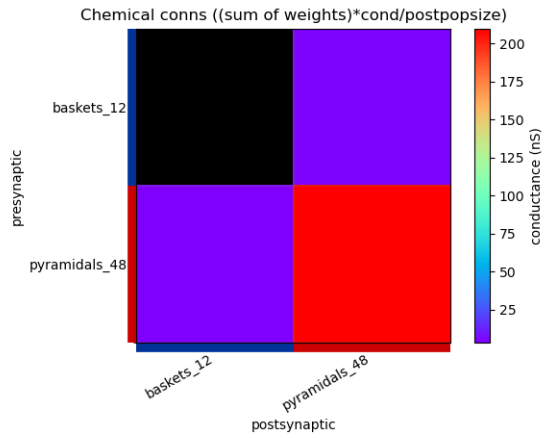
```
Usage:
pynml <NeuroML file> -matrix <level>
```

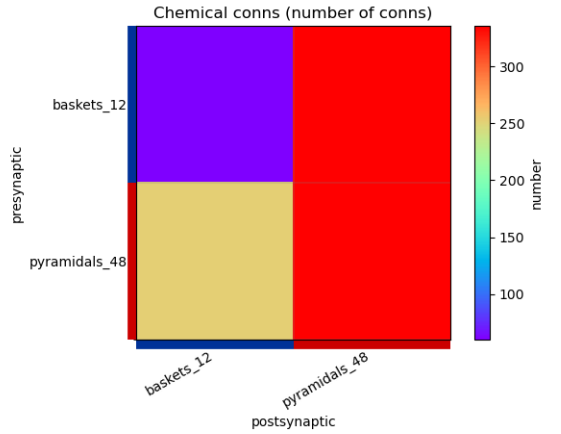
For example, to generate a PNG image of the connectivity matrices in the Auditory Cortex model used above, we can use:

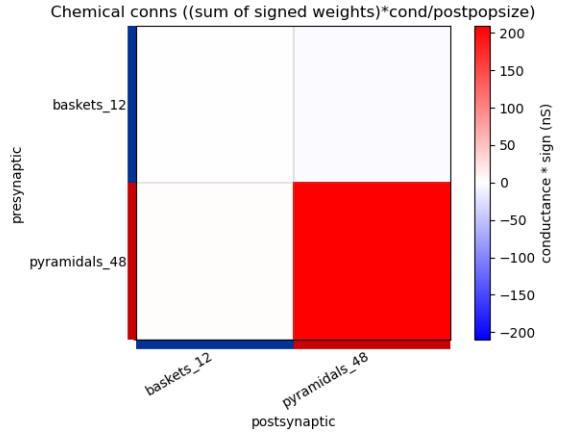
```
pynml MediumNet.net.nml -matrix 1
```

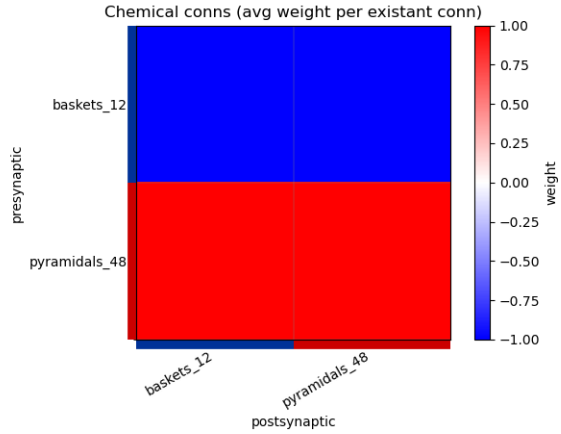
This generates the following images showing different views of the connectivity matrices in the network :






Chemical conns (number of conns)
baskets_12
pyramidals_48
presynaptic
baskets_12
pyramidals_48
postsynaptic
300
250
200
150
100
number


Chemical conns ((sum of signed weights)*cond/postpopsize)
baskets_12
pyramidals_48
presynaptic
baskets_12
pyramidals_48
postsynaptic
200
150
100
50
0
−50
−100
−150
−200
conductance * sign (nS)


Chemical conns (avg weight per existant conn)
baskets_12
pyramidals_48
presynaptic
baskets_12
pyramidals_48
postsynaptic
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
weight

8.5 View graph of the simulation instance of the model

8.5.1 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).

8.5.2 Using pyNeuroML

You can also generated these figures from within your *pyNeuroML* script itself using the `generate_lemsgraph` function:

```
import pyneuroml.pynml

...

pyneuroml.pynml.generate_lemsgraph(lems_file)
```

8.6 Viewing/analysing ion channel dynamics

There is a dedicated section on *visualising and analysing ion channel models*.

8.7 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.

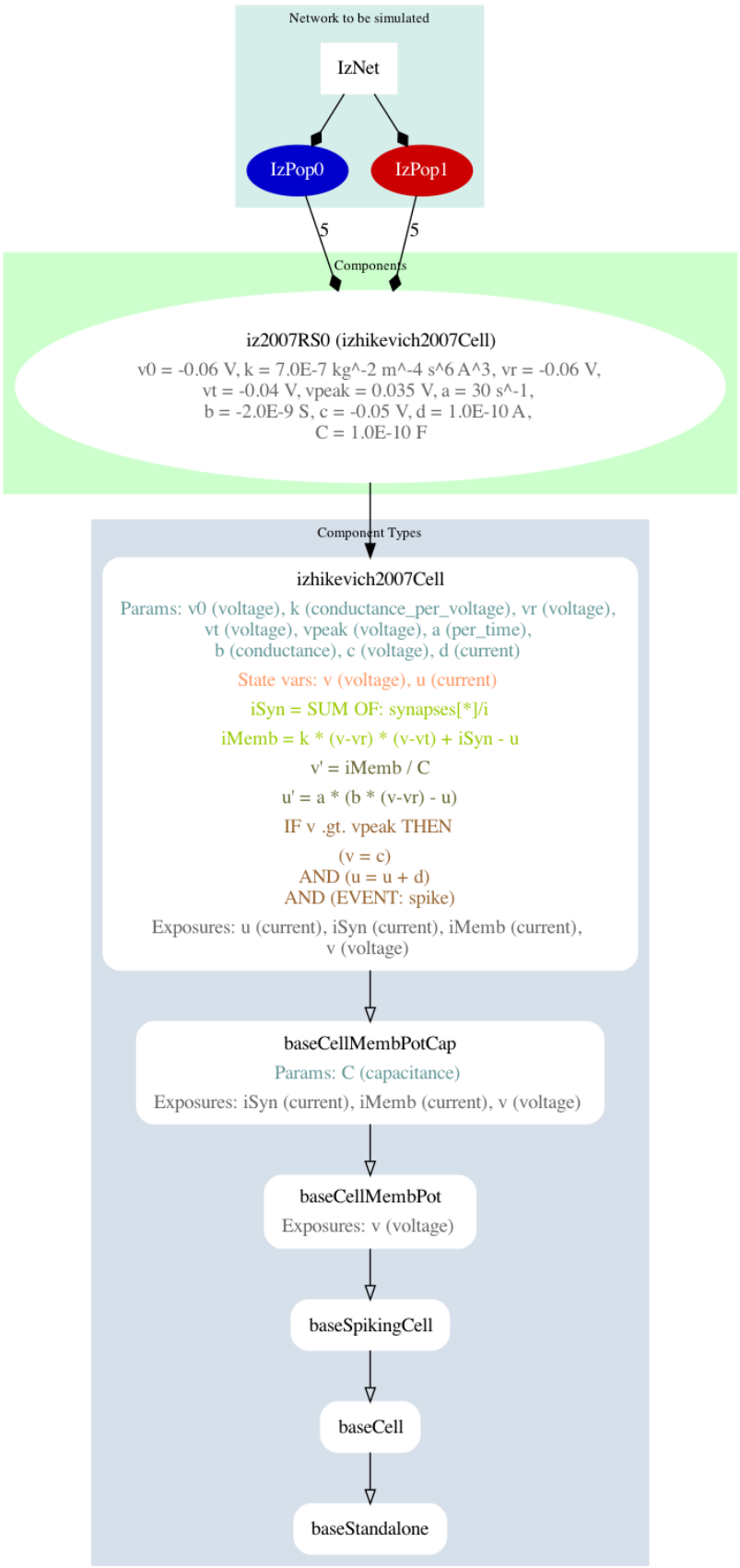


Fig. 8.6: A summary graph of the model generated using jnml.

8.7.1 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...)

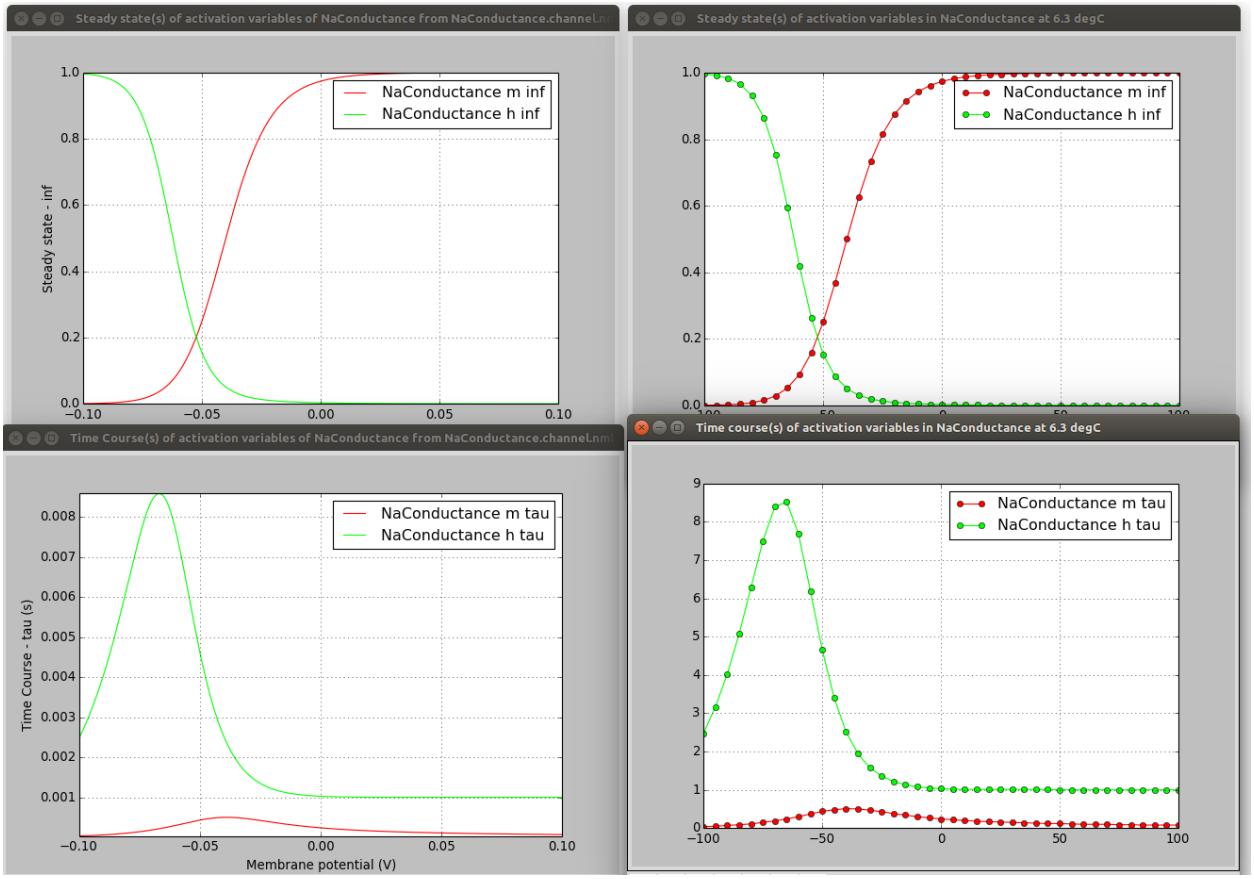
jNeuroML can be used on channel files for a quick summary of the contents.

```
> 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 utilities in pyNeuroML

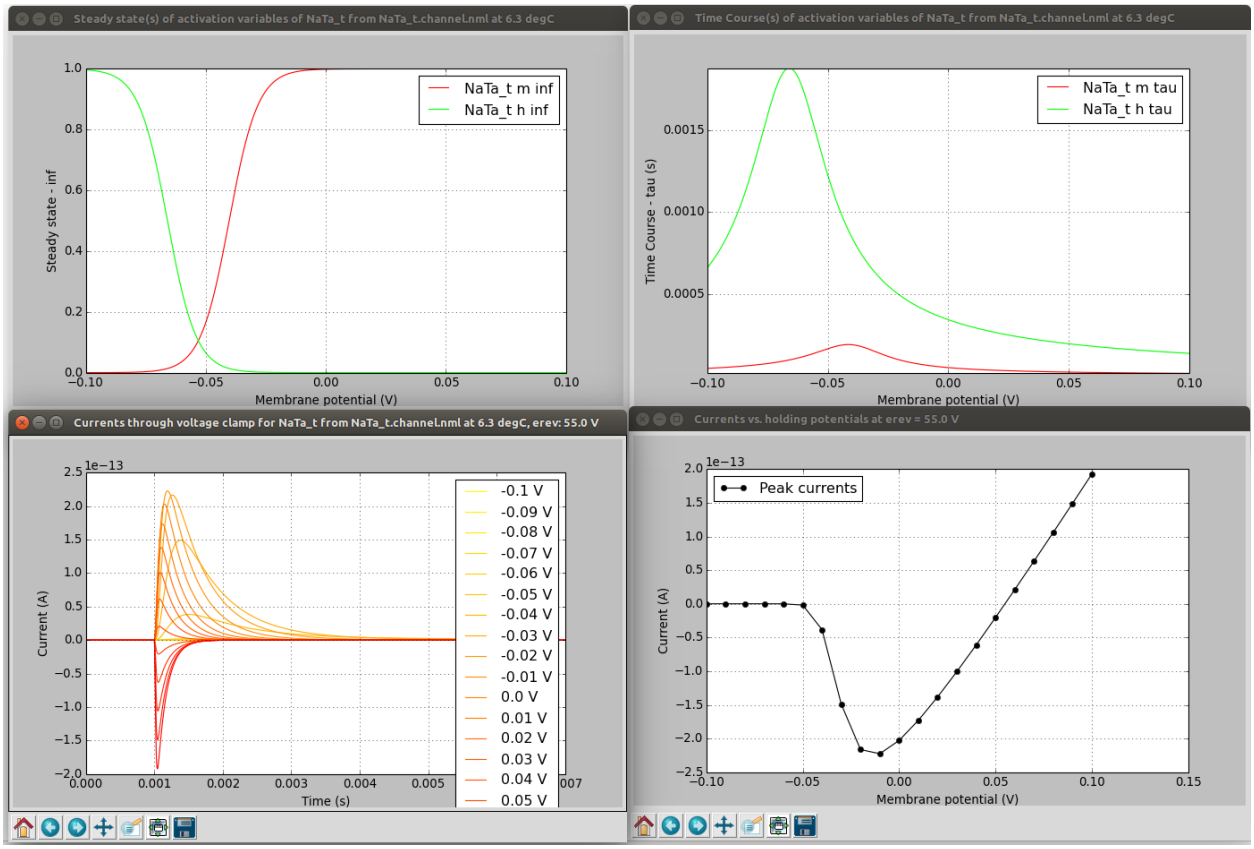
pyNeuroML comes with 3 utilities which help enable examination of the properties of ion channels, both based on NeuroML and NEURON mod files.

```
pynml-channelanalysis NaConductance.channel.nml    # Analyse a NeuroML 2 channel
pynml-plotchan cell.nml    # Plot distribution of peak channel conductances over cell␣
 ↪morphology
pynml-modchananalysis NaConductance                # Analyse a NEURON channel e.g.␣
 ↪from NaConductance.mod
```

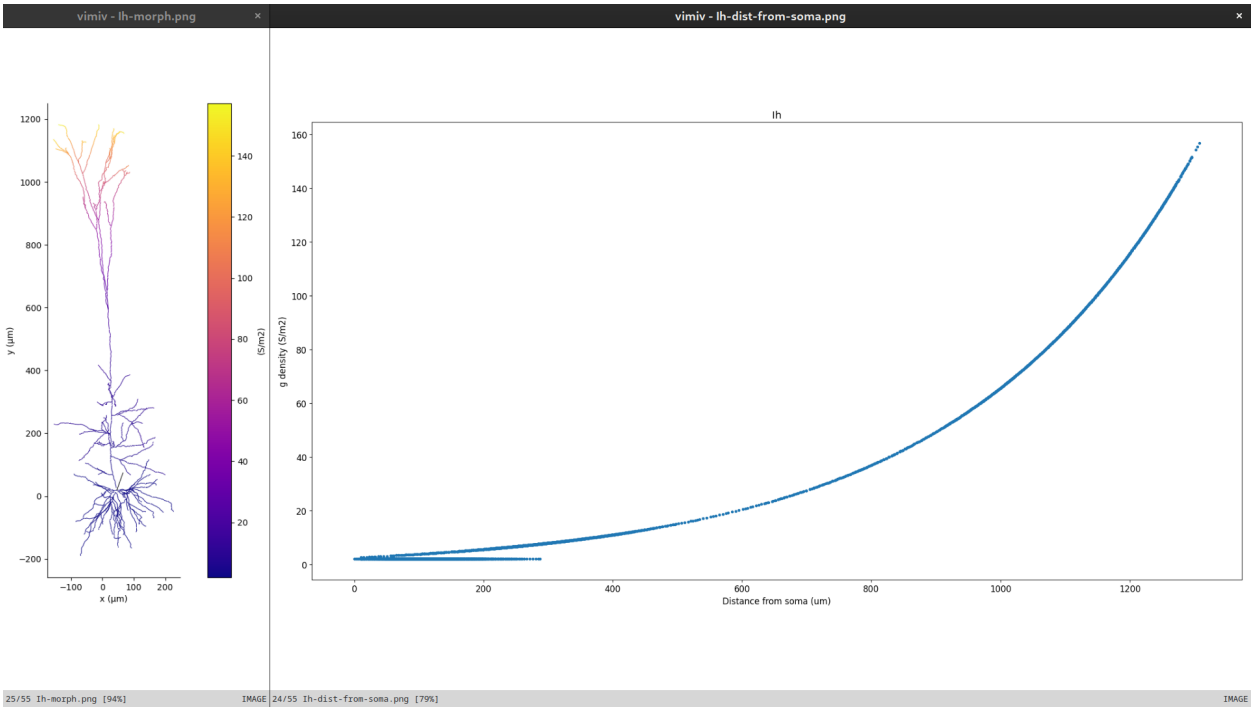


`pynml-channelanalysis` includes a number of options for generating graphs of channel activity under different conditions (see here for details).

```
pynml-channelanalysis NaTa_t.channel.nml  -erev 55 -stepTargetVoltage 10 -
 ↪clampDuration 5 -i -duration 7 -clampDelay 1
```



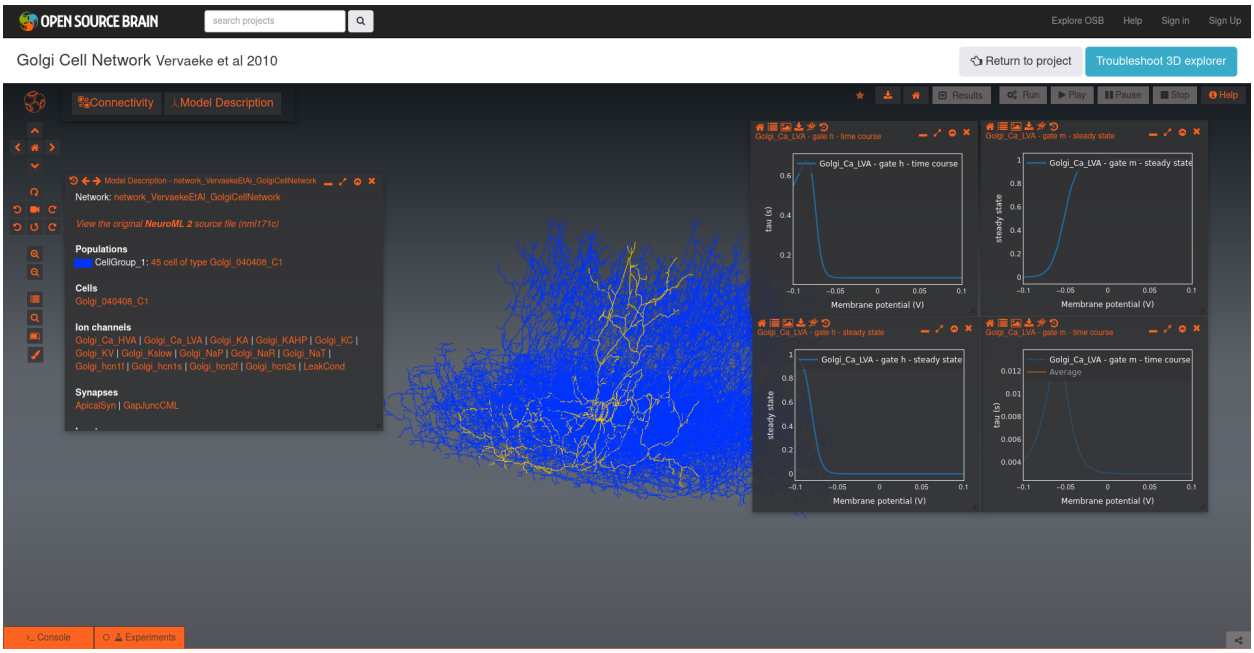
`pynml-plotchan` will plot the distribution of peak conductances of all channels in a cell over its morphology, and also show distribution as a function of distance from the soma. For example, the figure below shows the distribution of the Ih channel in the layer 5 pyramidal neuron model on the neuronal morphology on the left, and the value at different distances from the soma on the right.



This functionality is also available as a Python function for use in scripts.

4) Load cell model on to OSBv1 & analyse the channels

Open Source Brain (version 1) includes channel analysis functionalities.



5) 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
    }
...
```

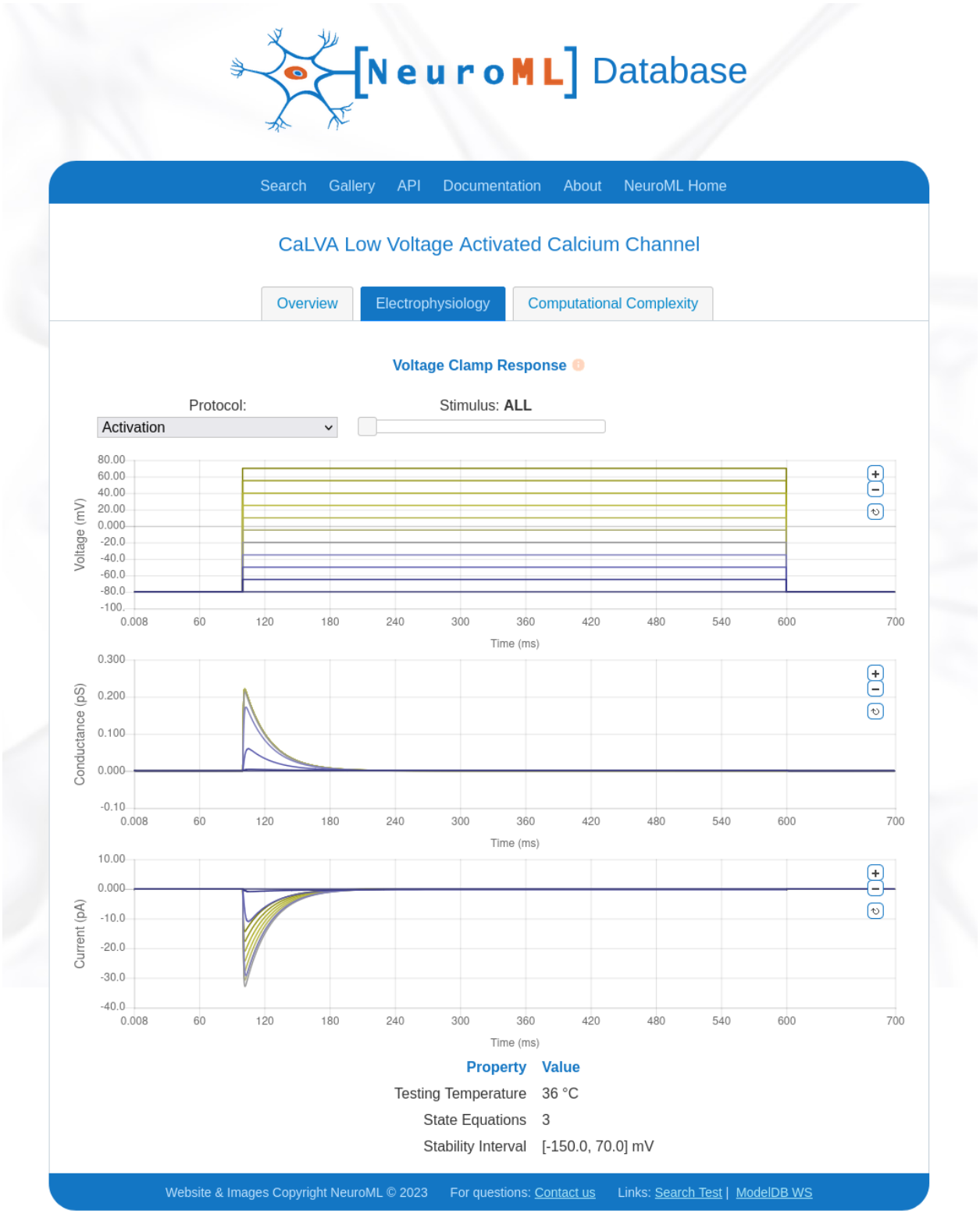
Exporting to Brian 2 (`jnml LEMS_NML2_Ex5_DetCell.xml -brian2`) will also produce a large file with the explicit expressions…

```
...
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
    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…

6) Use NeuroML-DB

NeuroML-DB also provides analysis features for Ion channels.


[NeuroML] Database
Search
Gallery
API
Documentation
About
NeuroML Home
CaLVA Low Voltage Activated Calcium Channel
Overview
Electrophysiology
Computational Complexity
Voltage Clamp Response
Protocol:
Activation
Stimulus: ALL
Voltage (mV)
Time (ms)
Conductance (pS)
Time (ms)
Current (pA)
Time (ms)
Property
Value
Testing Temperature
36 °C
State Equations
3
Stability Interval
[-150.0, 70.0] mV
Website & Images Copyright NeuroML © 2023
For questions: Contact us
Links: Search Test | ModelDB WS

8.8 Visualising and analysing cell models

The NeuroML ecosystem include a number of utilities for analysis and visualisation of cells. Cell morphologies can either be visualised programmatically using the core tools, or using the many advanced neuroinformatics tools in the ecosystem that support NeuroML. In addition to the resources listed below, you can also use the visualisation features of any other tools that read NeuroML. E.g., *NetPyNE and NetPyNE-UI*, *neuroConstruct*, *Arbor* and others.

8.8.1 Visualising morphology of multi-compartmental cell models

Multi-compartmental cells can be visualised using the plot_2D and plot_interactive_3D methods included in *pyNeuroML*. This functionality is also exposed via the `pynml-plotmorph` command line tool.

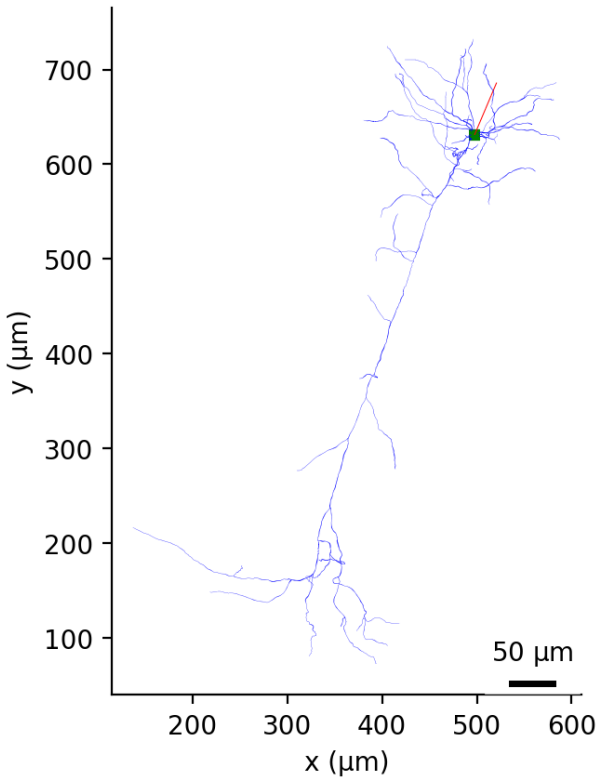


Fig. 8.7: Morphology of example cell plotted with `plot_2D` in the X-Y plane.

Visualising morphology of multi-compartmental cell models in NeuroML-db

The *NeuroML-DB* platform shows detailed cell morphologies of all cells included in its database.

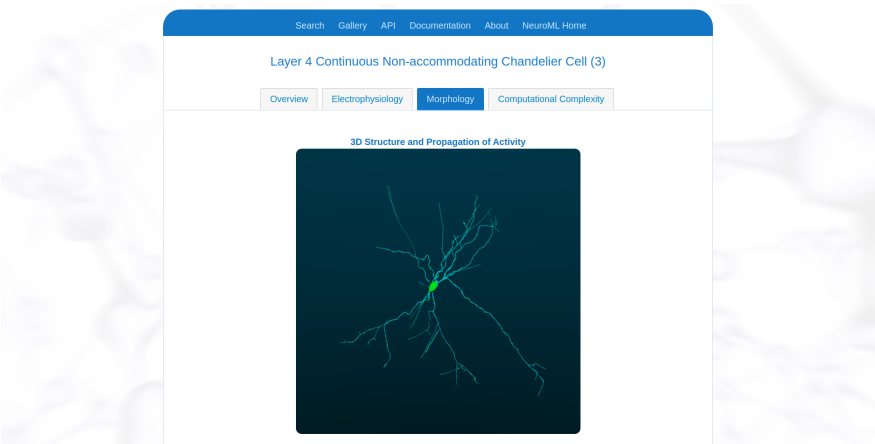


Fig. 8.8: Visualisation of morphology of an example cell on NeuroML-DB.

Visualising morphology of multi-compartmental cell models in Open Source Brain

The *Open Source Brain* platform also provides advanced visualisation capabilities that can be used to visualise the morphologies of NeuroML cells.

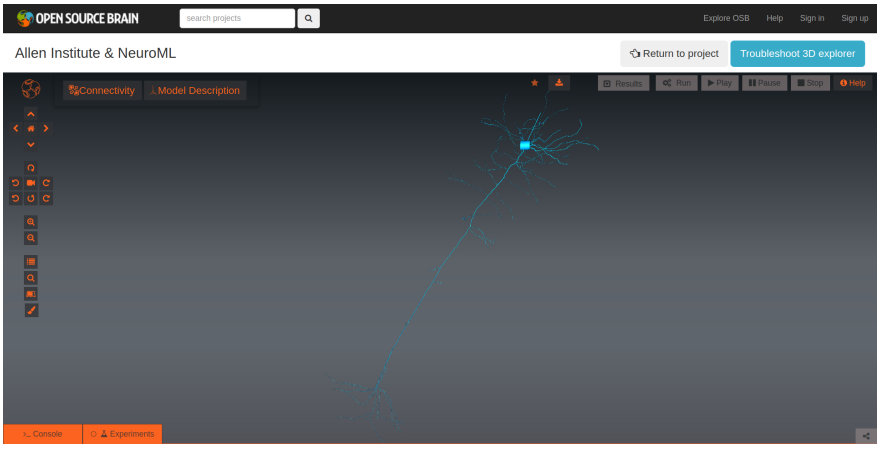


Fig. 8.9: Interactive visualisation of morphology of an example cell on Open Source Brain.

8.8.2 Analysing cell electrophysiology

The core tools also include utilities to aid in the analysis of cell electrophysiology. *pyNeuroML* includes the generate_current_vs_frequency_curve utility function that can be used to generate current-frequency, current-sub-threshold voltage, and to plot voltage traces generated at the soma for different current injections. For example, we can analyse the *OLM cell from our tutorial*:

```
generate_current_vs_frequency_curve("source/Userdocs/NML2_examples/olm.cell.nml", "olm
↪", simulator="jNeuroML_NEURON", plot_iv=True, plot_if=True, plot_voltage_
↪traces=True)
```

This will generate these figures:

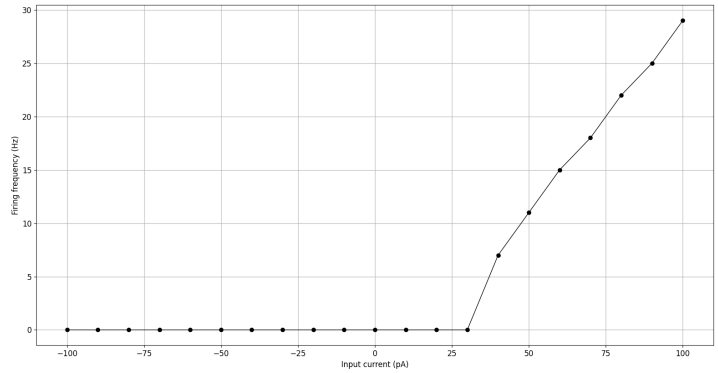


Fig. 8.10: F-I curve for OLM cell generated using `generate_current_vs_frequency_curve`.

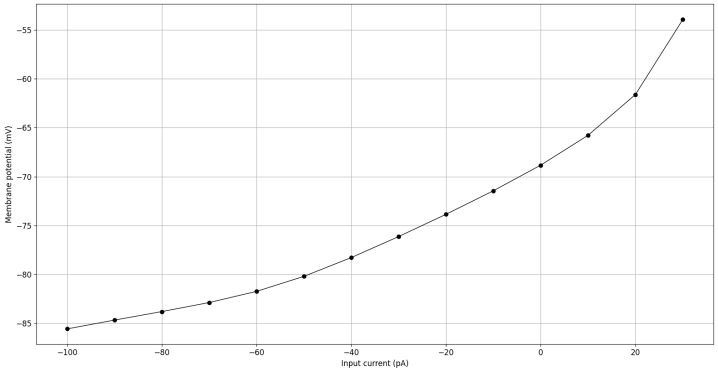


Fig. 8.11: Current vs sub-threshold voltage curve for OLM cell generated using `generate_current_vs_frequency_curve`.

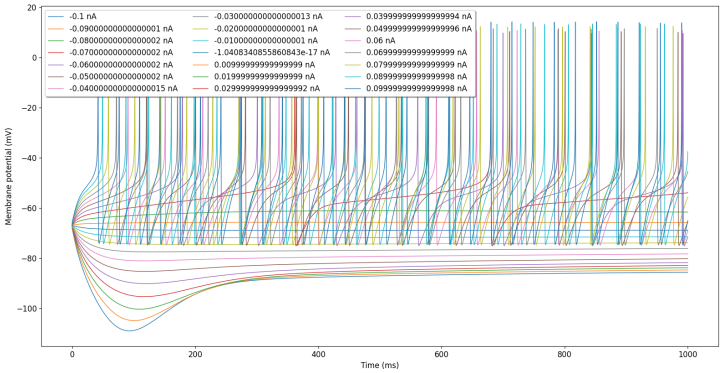


Fig. 8.12: Voltage traces for OLM cell with different injection currents generated using `generate_current_vs_frequency_curve`.

CHAPTER

NINE

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.

9.1 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.

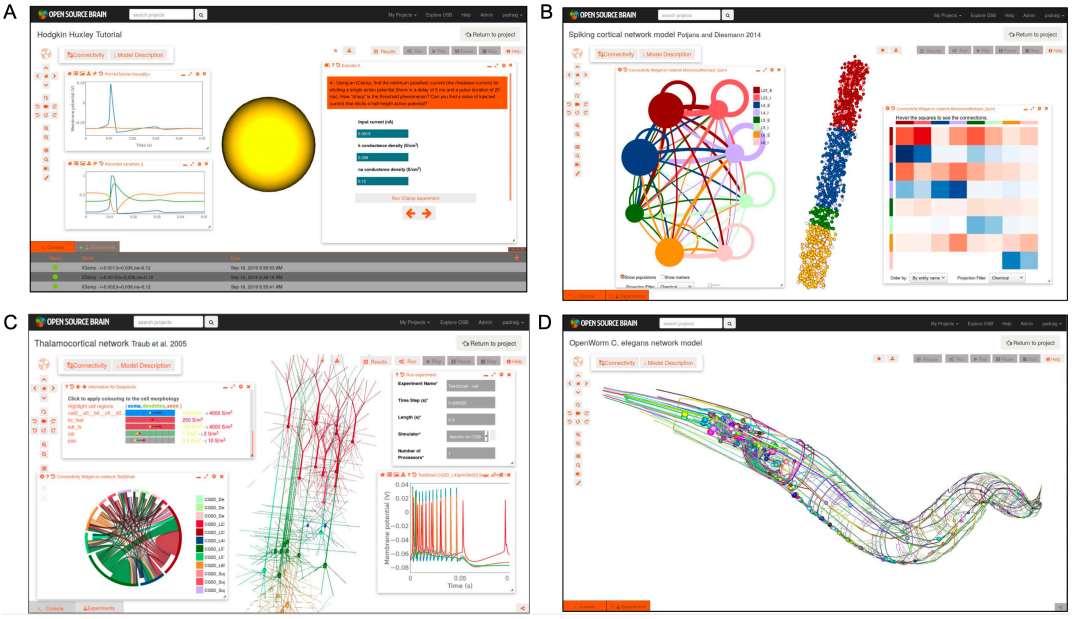


Fig. 9.1: Examples of NeuroML 2 models visualised on Open Source Brain. A) Hodgkin Huxley model interactive tutorial. B) Integrate and fire network model of cortical column (Potjans and Diesmann 2014), showing network connectivity. C) Cortical model with multicompartmental cells (Traub et al. 2005), 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*.

9.2 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. 9.2: 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)
```

9.3 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*.

9.4 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*.

9.5 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*.

9.6 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*.

9.7 Using EDEN

The EDEN simulator can load and simulate NeuroML v2 models.

There is a **dedicated page on EDEN/NeuroML interactions** *here*.

9.8 Using Arbor

You can import NeuroML models to the Arbor simulator.

There is a **dedicated page on Arbor/NeuroML interactions** *here*.

CHAPTER

TEN

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.

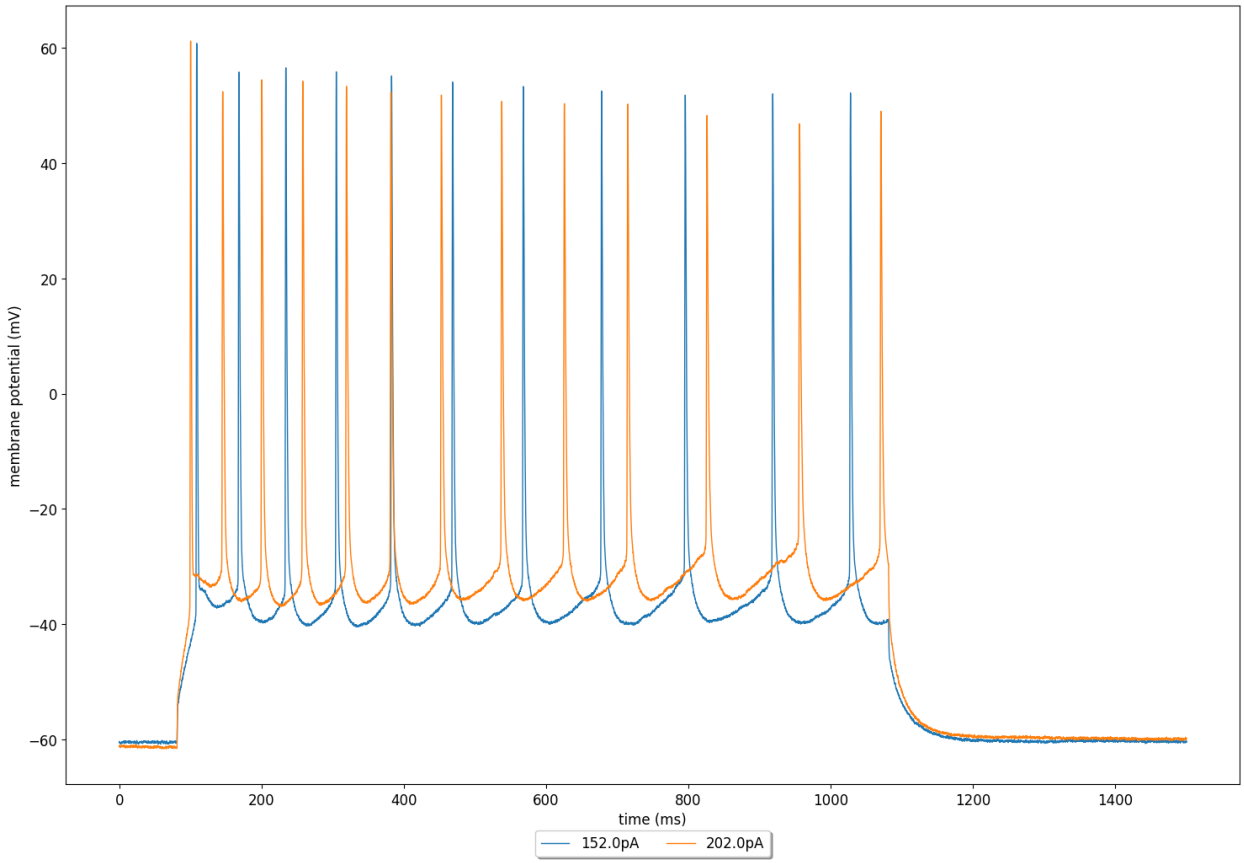


Fig. 10.1: Membrane potential from the experimental data.

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)

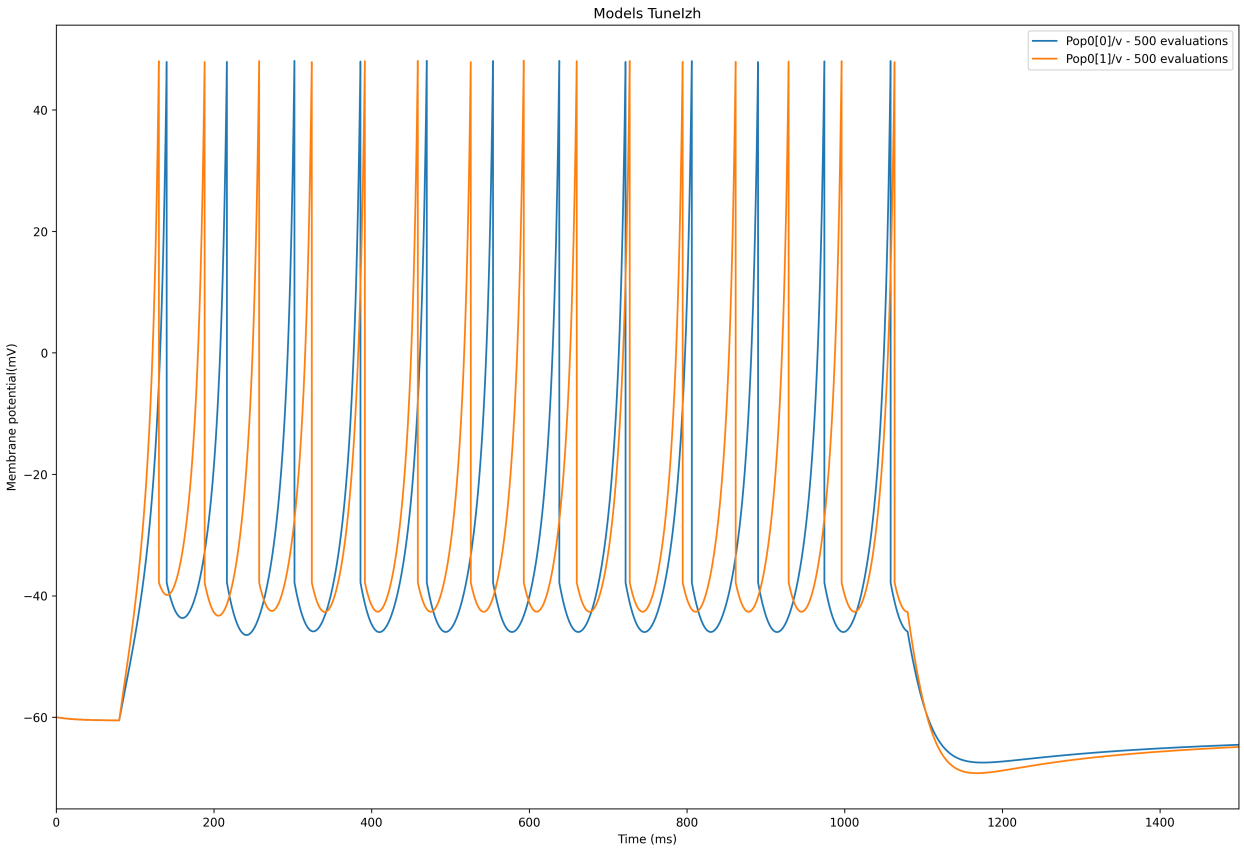


Fig. 10.2: Membrane potential obtained from the model with highest fitness.

(continued from previous page)

```
File: source/Userdocs/NML2_examples/tune-izhikevich.py

Copyright 2023 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_{:02d}".format(acq)].
 ↪data[:15000] * 1000.0
        )
        # get sampling rate from the data
        sampling_rate = datafile.acquisition[
            "CurrentClampSeries_{:02d}".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
```

(continues on next page)

(continued from previous page)

```
        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
    popsize = len(acq_list)
    template_doc.networks[0].populations.append(
```

(continues on next page)

(continued from previous page)

```
        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]]

    # Set up our target data and so on
    ctr = 0
    target_data = {}
```

(continues on next page)

(continued from previous page)

```
    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
        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,
    )


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"

    # 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
```

(continues on next page)

(continued from previous page)

```
    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)

    simulation = LEMSSimulation(
        sim_id=simulation_id,
        duration=sim_time,
        dt=0.1,
        target="Network0",
        simulation_seed=54321,
```

(continues on next page)

(continued from previous page)

```
    )
    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)",
        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(
```

(continues on next page)

(continued from previous page)

```
        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()
```

10.1 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 can be downloaded here on the Open Source Brain repository, and can also be viewed on the NWB Explorer web application:

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)
```

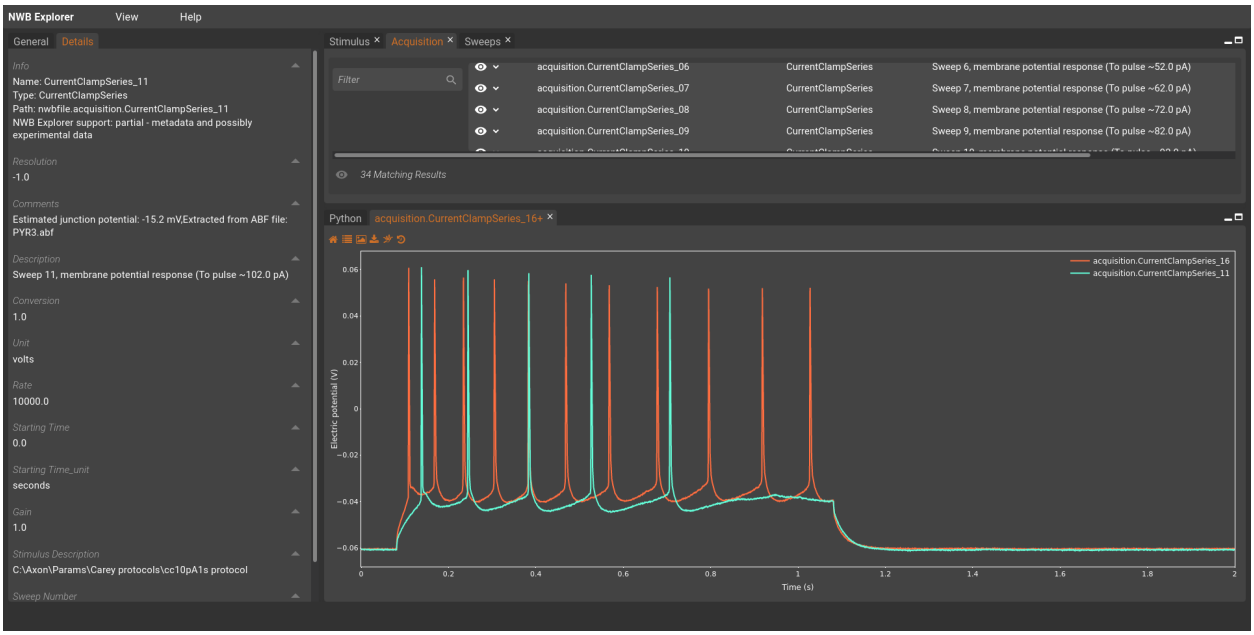


Fig. 10.3: Screenshot showing two recordings from FergusonEtAl2015_PYR3.nwb in NWB Explorer.

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

    """
    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_{:02d}".format(acq)].
 ↪data[:15000] * 1000.0
        )
        # get sampling rate from the data
        sampling_rate = datafile.acquisition[
            "CurrentClampSeries_{:02d}".format(acq)
        ].rate
```

(continues on next page)

(continued from previous page)

```
        # 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_{: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)
```

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,
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
```

(continues on next page)

(continued from previous page)

```
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': 482.20000000000005,
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': 192.10000000000002,
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,
pyelectro >>>     '9:min_peak_no': 3,
pyelectro >>>     '9:minimum': -60.91309,
pyelectro >>>     '9:peak_decay_exponent': -0.008125577959852965,
pyelectro >>>     '9:peak_linear_gradient': -0.007648055557429393,
pyelectro >>>     '9:spike_broadening': 0.891046031985908,
pyelectro >>>     '9:spike_frequency_adaptation': -0.05850411039850073,
pyelectro >>>     '9:spike_width_adaptation': 0.027679481742122 46,
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': 132.60000000000005,
```

(continues on next page)

(continued from previous page)

```
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': 106.50000000000003,
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': 127.40000000000002,
pyelectro >>>     '12:interspike_time_covar': 0.16275057704467177,
pyelectro >>>     '12:max_interspike_time': 136.5,
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,
```

(continues on next page)

(continued from previous page)

```
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': 74.40000000000002,
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,
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,
```

(continues on next page)

(continued from previous page)

```
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,
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': 53.09999999999998,
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': -0.027604513782098885,
pyelectro >>>     '18:trough_phase_adaptation': -0.005774543184281237}
Going over acquisition # 19
pyelectro >>> {   '19:average_last_1percent': -60.855106353759766,
```

(continues on next page)

(continued from previous page)

```
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': 49.099999999999994,
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,
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,
```

(continues on next page)

(continued from previous page)

```
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': 37.599999999999994,
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,
pyelectro >>>     '24:average_maximum': 50.66572,
pyelectro >>>     '24:average_minimum': -34.043533,
pyelectro >>>     '24:first_spike_time': 95.9,
pyelectro >>>     '24:interspike_time_covar': 0.2757061784222999,
pyelectro >>>     '24:max_interspike_time': 106.29999999999995,
pyelectro >>>     '24:max_peak_no': 14,
pyelectro >>>     '24:maximum': 61.2793,
pyelectro >>>     '24:mean_spike_frequency': 13.685651121170649,
pyelectro >>>     '24:min_interspike_time': 42.5,
pyelectro >>>     '24:min_peak_no': 13,
pyelectro >>>     '24:minimum': -61.03516,
pyelectro >>>     '24:peak_decay_exponent': -0.10291055243646331,
pyelectro >>>     '24:peak_linear_gradient': -0.008156801002617309,
pyelectro >>>     '24:spike_broadening': 0.647962840377368,
```

(continues on next page)

(continued from previous page)

```
pyelectro >>>     '24:spike_frequency_adaptation': -0.01033640102347251,
pyelectro >>>     '24:spike_width_adaptation': 0.0070544336550024695,
pyelectro >>>     '24:trough_decay_exponent': -0.04136814132208841,
pyelectro >>>     '24:trough_phase_adaptation': -0.007165702258804302}
Going over acquisition # 25
pyelectro >>> {   '25:average_last_1percent': -60.056766510009766,
pyelectro >>>     '25:average_maximum': 50.34093,
pyelectro >>>     '25:average_minimum': -33.228947,
pyelectro >>>     '25:first_spike_time': 95.60000000000001,
pyelectro >>>     '25:interspike_time_covar': 0.28833246313094774,
pyelectro >>>     '25:max_interspike_time': 100.80000000000007,
pyelectro >>>     '25:max_peak_no': 14,
pyelectro >>>     '25:maximum': 61.40137,
pyelectro >>>     '25:mean_spike_frequency': 14.023732470334416,
pyelectro >>>     '25:min_interspike_time': 39.10000000000001,
pyelectro >>>     '25:min_peak_no': 13,
pyelectro >>>     '25:minimum': -60.607914,
pyelectro >>>     '25:peak_decay_exponent': -0.1047695739806843,
pyelectro >>>     '25:peak_linear_gradient': -0.00879119329941481,
pyelectro >>>     '25:spike_broadening': 0.6394636967007732,
pyelectro >>>     '25:spike_frequency_adaptation': -0.011357738090467376,
pyelectro >>>     '25:spike_width_adaptation': 0.007198805900434394,
pyelectro >>>     '25:trough_decay_exponent': -0.03898706127522965,
pyelectro >>>     '25:trough_phase_adaptation': -0.005791081007349543}
Going over acquisition # 26
pyelectro >>> {   '26:average_last_1percent': -60.2272580464681,
pyelectro >>>     '26:average_maximum': 49.776356,
pyelectro >>>     '26:average_minimum': -32.534084,
pyelectro >>>     '26:first_spike_time': 94.89999999999999,
pyelectro >>>     '26:interspike_time_covar': 0.3174070553930572,
pyelectro >>>     '26:max_interspike_time': 115.70000000000005,
pyelectro >>>     '26:max_peak_no': 14,
pyelectro >>>     '26:maximum': 61.15723,
pyelectro >>>     '26:mean_spike_frequency': 14.697569248162802,
pyelectro >>>     '26:min_interspike_time': 35.60000000000001,
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,
```

(continues on next page)

(continued from previous page)

```
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,
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': 25.599999999999994,
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,
```

(continues on next page)

(continued from previous page)

```
pyelectro >>>     '30:min_interspike_time': 24.599999999999994,
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,
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,
```

(continues on next page)

(continued from previous page)

```
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

10.2 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")
    # 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]),
```

(continues on next page)

(continued from previous page)

```
            )
        )
        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]]

    # 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
```

(continues on next page)

(continued from previous page)

```
        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
        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,
```

(continues on next page)

(continued from previous page)

```
        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.

10.2.1 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
```

(continues on next page)

(continued from previous page)

```
    # 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)
```

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.

10.2.2 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)
        ]
```

(continues on next page)

(continued from previous page)

```
        # 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 the 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,
```

(continues on next page)

(continued from previous page)

```
'Pop0[1]/v:average_maximum': 1,
'Pop0[1]/v:average_minimum': 1
}
```

For simplicity, we set the weights for all as `1` here.

10.2.3 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
        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,
    '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,
```

(continues on next page)

(continued from previous page)

```
    'Pop0[0]/v:mean_spike_frequency': 6.957040276293886,
    'Pop0[1]/v:average_last_1percent': -59.28251401166662,
    'Pop0[1]/v:average_maximum': 47.242452454545464,
    '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 higher fitness

10.3 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.

10.3.1 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 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"
```

(continues on next page)

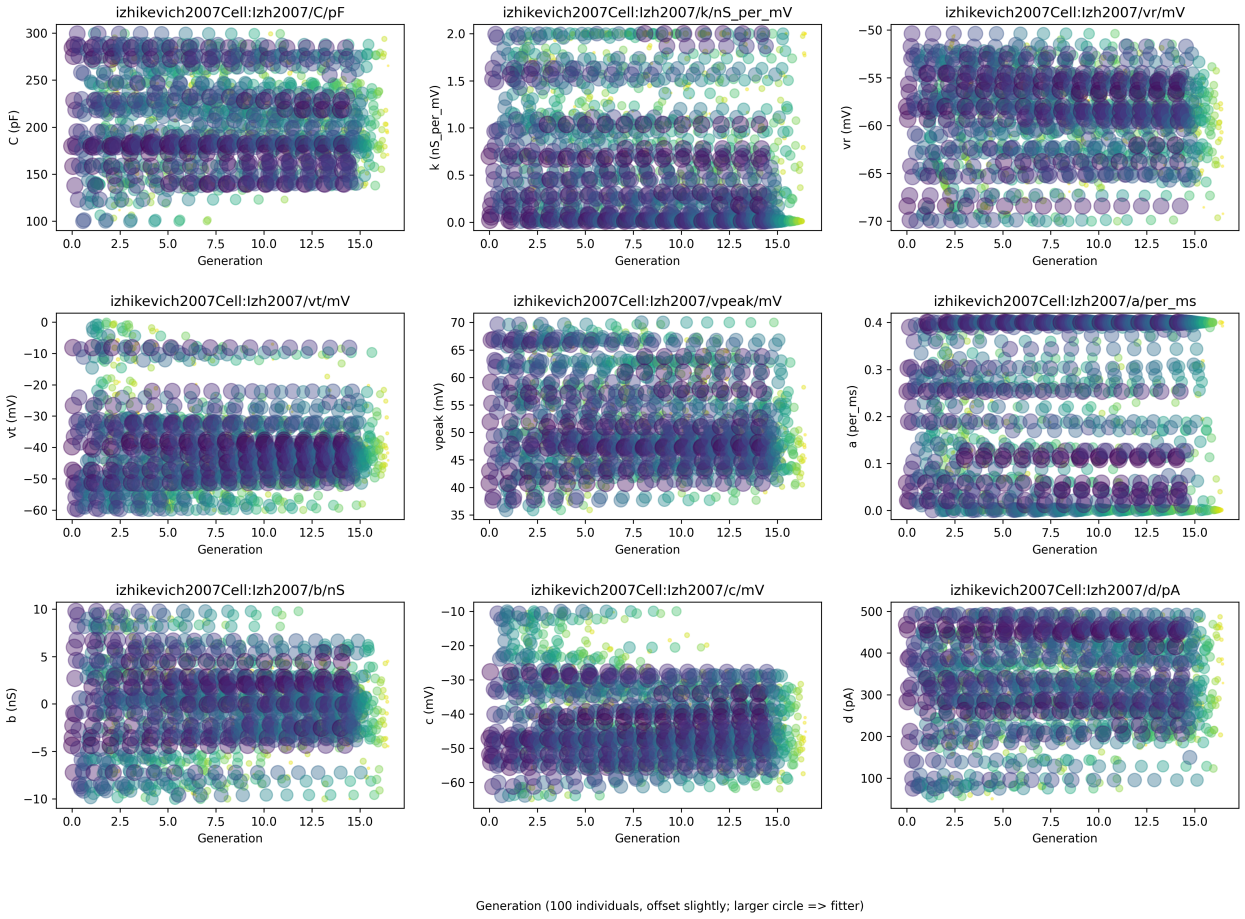


Fig. 10.4: The figure shows the values of various parameters throughout the evolution, with larger circles having higher values of fitness.

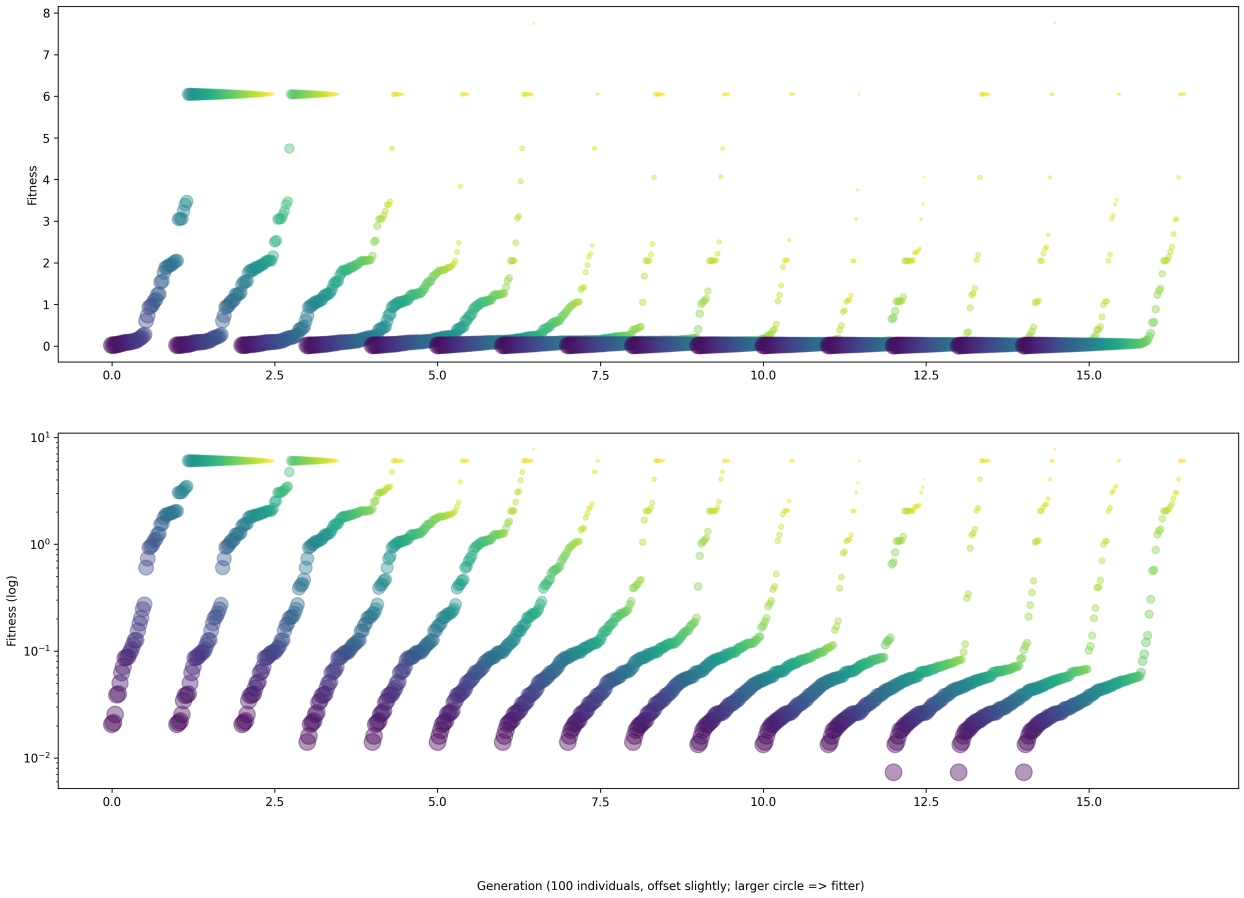


Fig. 10.5: The figure shows the trend of the fitness throughout the evolution.

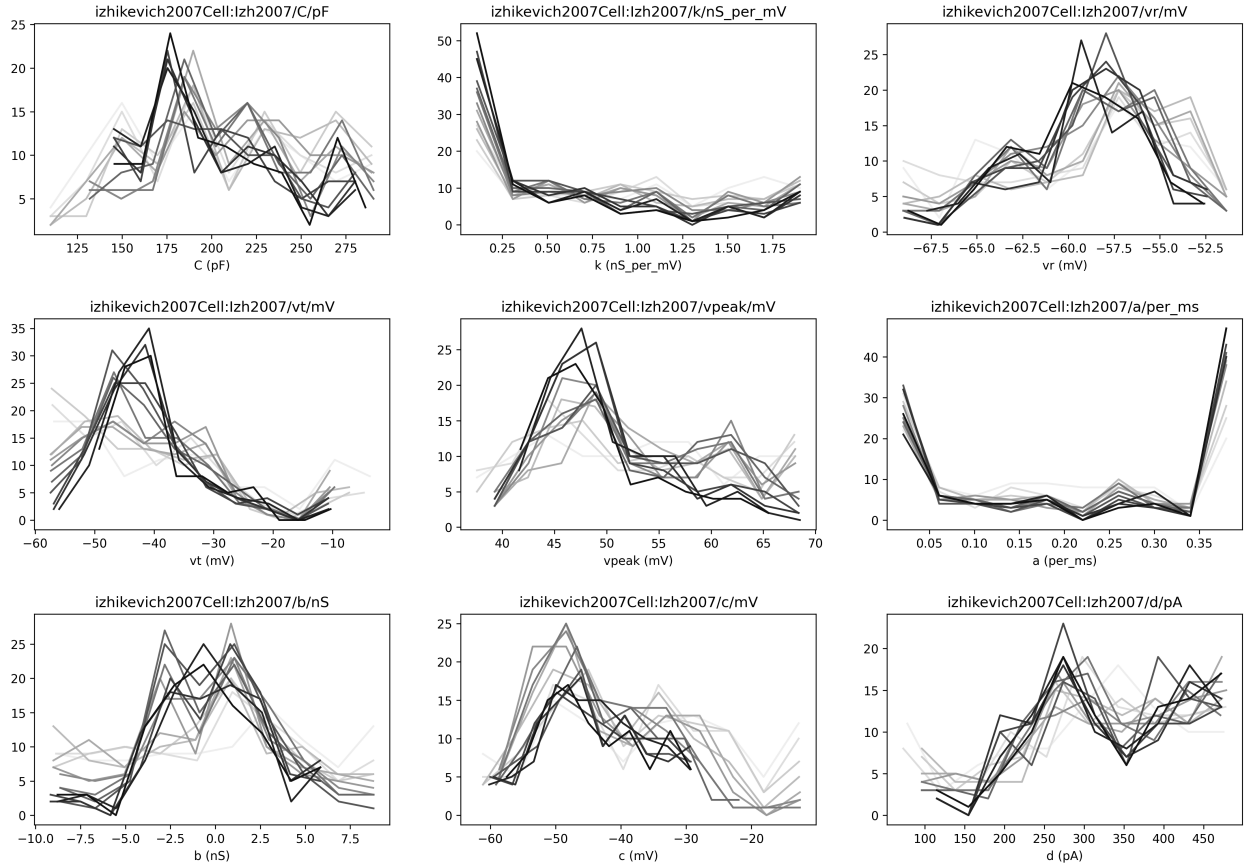


Fig. 10.6: The figure shows the distribution of values that for each parameter throughout the evolution. Darker lines have higher fitness values.

(continued from previous page)

```
    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
    # 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
```

(continues on next page)

(continued from previous page)

```
    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.

10.3.2 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 = []
```

(continues on next page)

(continued from previous page)

```
    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)",
    )
```

This generates the following figures:

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.

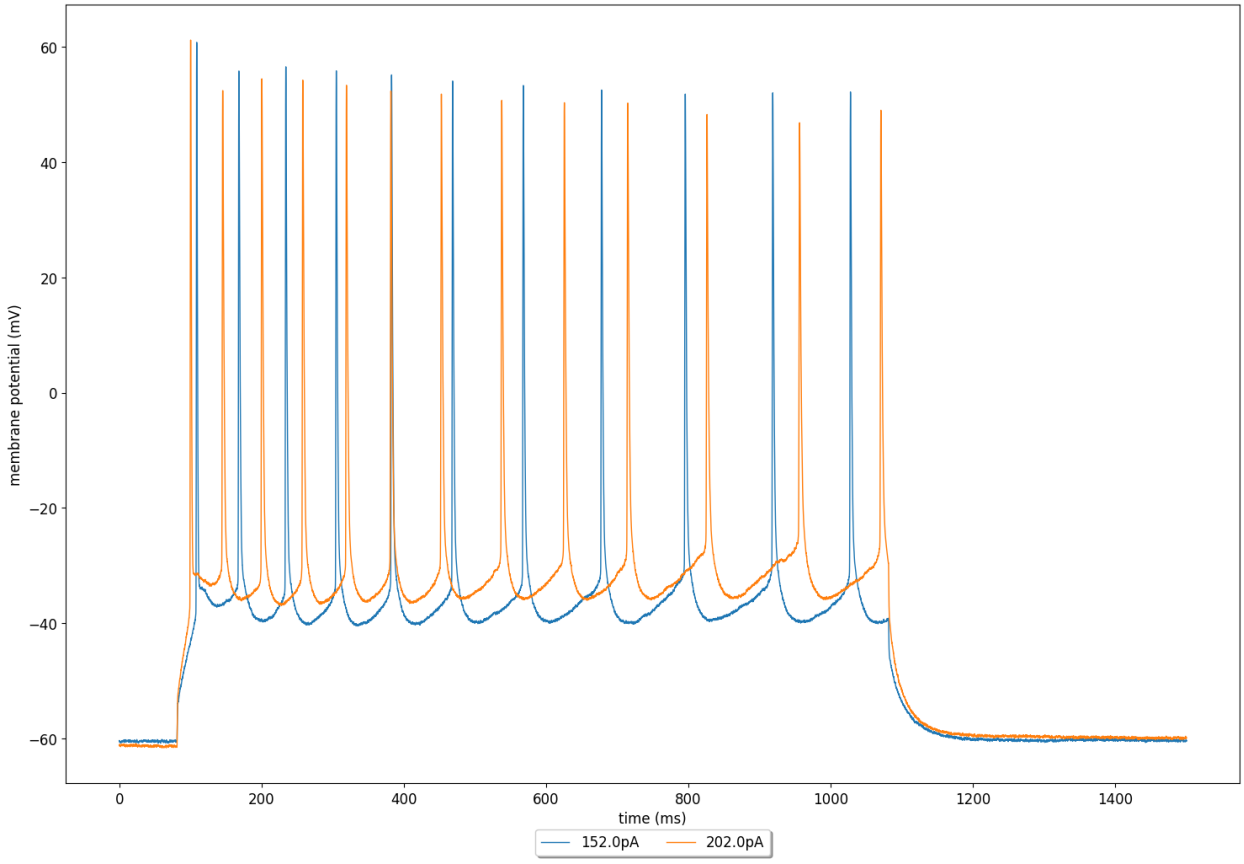


Fig. 10.7: Membrane potential from the experimental data.

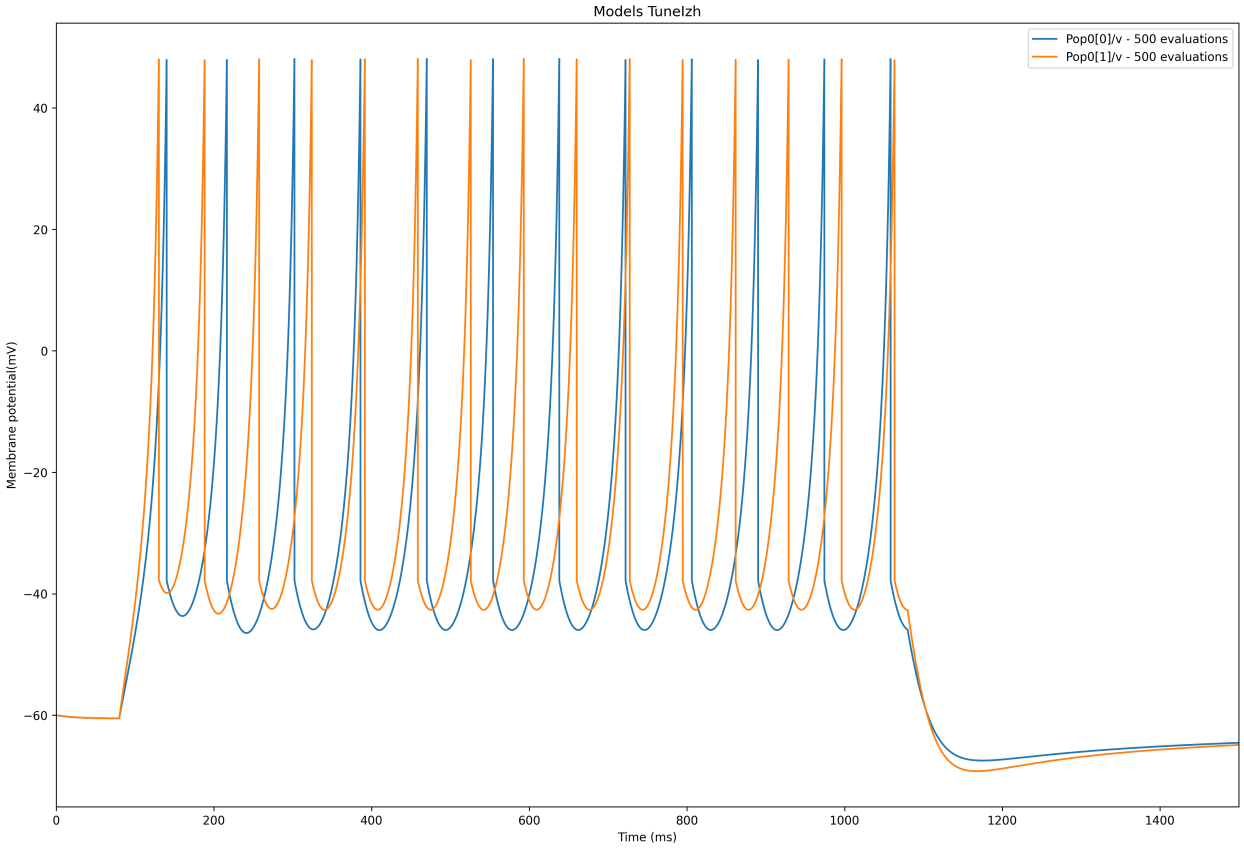


Fig. 10.8: Membrane potential obtained from the model with highest fitness.

CHAPTER

ELEVEN

TESTING/VALIDATING NEUROML MODELS

Models described in NeuroML can be run across multiple simulators, and it it essential that the activity (e.g. spike times) of the models are as close as possible across all of these independently developed platforms.

It is also important to validate that the behaviour of a given NeuroML model reproduces some recorded aspect of the biological equivalent.

11.1 Testing behaviour of NeuroML models across simulators

This type of testing addresses the question: **Does a given NeuroML model produce the same results when run across multiple simulators?**

11.1.1 OMV - Open Source Brain Model Validation framework

The OSB Model Validation framework was originally developed as an automated model validation package for Open Source Brain projects, which can be used for testing model behaviour across many simulation engines both:

- on your local machine when developing models
- on GitHub Actions, to ensure tests pass on every commit.

This framework has been used to test the 30+ NeuroML and PyNN models described in the Open Source Brain paper (Gleeson et al. 2019), and many more.

See https://github.com/OpenSourceBrain/osb-model-validation for more details.

11.2 Validating that NeuroML model reproduce biological activity

This type of testing addresses the question: **How well does a given NeuroML model replicate the activity as seen in real neurons/channels/networks?**

11.2.1 SciUnit/NeuronUnit

SciUnit is a Python framework for test-driven validation of scientific models, and NeuronUnit is a package based on this for data-driven validation of neuron and ion channel models. See also SciDash for more information.

Interactive Jupyter notebooks for running NeuronUnit examples can be found this repository.

TODO: Add details on using SciUnit and NeuronUnit with NeuroML models.

CHAPTER

TWELVE

LEMS: LOW ENTROPY MODEL SPECIFICATION

A language for specifying hierarchical models based on fundamental physical relationships

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 being developed to provide a compact, minimally redundant, human-readable, human-writable, declarative way of expressing models of biological systems.

It differs from other systems such as CellML or SBML in its requirement to be human writable and the inclusion of basic physical concepts such as dimensionality and physical nesting as part of the language. The main goal is to enable model developers to write declarative models in LEMS in much the same way as software developers write software applications in computer languages such as in C, Java or Python. The examples shown here use XML for expressing models as text, but LEMS is not primarily an XML language. Rather it defines a set of structures for representing models. The reference implementation also supports a more concise indentation-based format for representing models.

There are two independent implementations of LEMS: jLEMS, written in Java and pyLEMS written in Python. Both are hosted on the github.com/LEMS.

12.1 Capabilities

You can define ComponentTypes (e.g. a "HH channel" or "a bi-exponential synapse") which express the general properties of a particular type of thing that goes in a model. This includes saying what parameters they have, what child elements they are allowed, and how they behave (the equations).

You can then define Components based on these types by supplying values for the parameters and adding any child elements that are required, so, for example, a bi-exponential synapse model with rise time 1ms and decay 5ms would be a component.

ComponentTypes can extend other ComponentTypes to add extra parameters, fix certain values, and otherwise modify their behavior. Components can extend other Components to reuse specified parameter values. There is also a loose notion of abstract types, so a component can accept children with a particular lineage without needing to know exactly what type they are. This can be used, for example, to define cells that accept synaptic connections provided they have a particular signature.

Each ComponentType can have a Dynamics element that specifies how it behaves: what the state variables are, the equations that govern them, and what happens when events are sent or received. The interpreter takes a model consisting of type and component elements referenced from a network, builds an instance from them and runs it.

For those familiar with object oriented languages, the ComponentType/Component distinction is close to the normal Class/Instance distinction. When the model is run, the same pattern applies again, with the Components acting as class definitions, with their "instances" actually containing the state variables in the running mode.

12.2 Background

The March 2010 NeuroML meeting (minutes) identified a need to extend the capability within NeuroML for expressing a range of models of synapses. It was decided that the hitherto adopted approach of defining parameterized building blocks to construct models by combining blocks and setting parameters was unlikely to be flexible enough to cope with the needs for synapse models. This is not obvious a-priori, since, for example, the pre NeuroML 2.0 ion channel building blocks are fully adequate to describe the dynamics of a wide range existing channel models. But there appears to be no such commonality in models used for synapses, where the mechanisms used range from highly detailed biochemical models to much more abstract ones.

This work also has antecedents in Catacomb 3, which was essentially a GUI for a component definition system and model builder using a type system similar to that proposed here. Much of the XML processing code used in the interpreter was taken from PSICS which itself currently uses the "building block" approach to model specification. The need for user-defined types has been considered with respect to future PSICS development, and this proposal also reflects potential requirements for PSICS.

12.3 Example

Here is the XML for a simple integrate-and-fire cell definition:

```
<ComponentType name="refractiaf">
    <Parameter name="threshold" dimension="voltage"/>
    <Parameter name="refractoryPeriod" dimension="time"/>
    <Parameter name="capacitance" dimension="capacitance"/>
    <Parameter name="vleak" dimension="voltage"/>
    <Parameter name="gleak" dimension="conductance"/>

    <Parameter name="current" dimension="current"/>
    <Parameter name="vreset" dimension="voltage"/>
    <Parameter name="deltaV" dimension="voltage"/>
    <Parameter name="v0" dimension="voltage"/>

    <EventPort name="out" direction="out"/>
    <EventPort name="in" direction="in"/>

    <Exposure name="v" dimension="voltage"/>

    <Dynamics>
        <StateVariable name="v" exposure="v" dimension="voltage" />
        <StateVariable name="tin" dimension="time"/>
        <OnStart>
            <StateAssignment variable="v" value="v0"/>
        </OnStart>

        <Regime name="refr">
            <OnEntry>
                <StateAssignment variable="tin" value="t" />
                <StateAssignment variable="v" value="vreset" />
            </OnEntry>
```

(continues on next page)

(continued from previous page)

```
            <OnCondition test="t .gt. tin + refractoryPeriod">
                <Transition regime="int" />
            </OnCondition>
        </Regime>

        <Regime name="int" initial="true">
            <TimeDerivative variable="v" value="(current + gleak * (vleak - v)) /␣
↪capacitance" />
            <OnCondition test="v .gt. threshold">
                <EventOut port="out" />
                <Transition regime="refr" />
            </OnCondition>

        </Regime>
    </Dynamics>

</ComponentType>
```

Once this definition is available, a particular model using this structure can be specified with the following XML:

```
<refractiaf threshold="-40mV" refractoryPeriod="5ms" capacitance="1nF" vleak="-80mV"␣
↪gleak="100pS" vreset="-70mV" v0="-70mV" deltaV="10mV" />
```

More complex models will have nested components and other types of parameters, but the basic principle of separating out the equations and parameters for reusable model components, such that the equations are only stated once, remains the same.

12.4 Model structure overview

Models are based on user-defined types (the term ComponentType is used in the XML) that contain parameter declarations, reference declarations and specification of what children an instance of a type can have. Typically they also contain a Dynamics specification which can contain build-time and run-time declarations. Build-time declarations apply when a simulation is set up, for example to connect cells. Run-time declarations specify the state variables, equations and events that are involved.

An instance of a ComponentType is a model Component It specifies a particular set of parameters for a given ComponentType. It says nothing about state variables: in a simulation, typically many run-time instances will correspond to a single model component definition, and several model component definitions will use the same type. A run-time instance holds its own set of state variables as defined by the Type definition and a reference to its component for the parameter values specific to that particular model component. The update rules come from the type definition. As such, neither the ComponentType nor the Component is properly a "prototype" for the runtime instance.

12.4.1 Defining ComponentTypes

ComponentTypes are declared as, for example:

```
<ComponentType name="myCell">
   <Parameter name="threshold" dimension="voltage" />
</ComponentType>
```

A Component based on such a type is expressed as:

```
<Component type="myCell" threshold="dimensional_quantity" />
```

The quoted value for 'threshold' here is a rich quantity with size and dimensions, typically consisting of a numerical value and a unit symbol. Assignments like this are the only place unit symbols can occur. Equations and expressions relate rich types, independent of any particular unit system.

An equivalent way of writing the above in shorthand notation (using an example of a string with size and dimension for threshold) is:

```
<myCell threshold="-30 mV" />
```

A type can contain elements for specifying the following aspects of the structure and parameters of a model component:

- Parameter - dimensional quantities that remain fixed within a model
- Child - a required single sub-component of a given type
- Children - variable number of sub-components of the given type
- ComponentRef - a reference to a top-level component definition.
- Link - a reference to a component definition relative to the referrer
- Attachments - for build-time connections
- EventPort - for run-time discrete event communication
- Exposure - quantities that can be accessed from other components
- Requirement - quantities that must be accessible to the component for it to make sense
- DerivedParameter - like parameters, but derived from some other quantity in the model

The "EventPort" and "Attachments" declarations don't have any corresponding elements in their model component specification. They only affect how the component can be used when a model is instantiated. EventPorts specify that a model can send or receive events, and should match up with declarations in its Dynamics specification. An "Attachments" declaration specifies that a run-time instance can have dynamically generated attachments as, for example, when a new synapse run-time instance is added to a cell for each incoming connection.

12.4.2 Inheritance

A type can extend another type, adding new parameters, or supplying values (SetParam) for inherited parameters. As well as reducing duplication, the key application of this is with the Child and Children declarations, where a type can specify that it needs a child or children of a particular supertype, but doesn't care about which particular sub-type is used in a model. This applies, for example, where a cell requires synapses that compute a quantity with dimensions current, but doesn't need access to any other parts of the synapse Dynamics.

12.4.3 Run-time Dynamics

Run time Dynamics are included within a Dynamics block in a type specification. They include declaration of:

- state variables
- first order differential equations with respect to time of state variables
- derived quantities - things computed in terms of other local quantities or computed from other run-time instances

Run time Dynamics can be grouped into Regimes, where only one regime is operative at a given time for a particular run-time instance. Regimes have access to all the variables in the parent instance and can define their own local variables.

Dynamics can also contain event blocks:

- OnStart blocks contain any initialization declarations needed when a run-time state is instantiated
- OnEvent blocks specify what happens when an event is received on a specified port
- OnEntry blocks (only within regimes) specify things that should happen each time the system enters that regime.
- OnCondition blocks have a test condition and specify what should happen when it is met.

Blocks may contain state variable assignments, event sending directives and transition directives to indicate that the system should change from one regime to another.

12.4.4 Build-time Structure

Build-time Structure defines the structure of a multi-component model. Currently there are:

- MultiInstantiate - for declaring that a component yields multiple run-time instances corresponding to a particular model component. Eg, for defining populations of cells.
- ForEach - for iterating over multiple instances in the run-time structure
- EventConnection - for connecting ports between run-time instances

12.4.5 Other

There are also Run, Show and Record Dynamics for creating type definitions that define simulations and what should be recorded or displayed from such a simulation.

12.4.6 Observations

The numerous references to "run-time instances" above is problematic, since the structures do not dictate any particular way of building a simulator or running a model. In particular, there is no requirement that a component or Dynamics declaration should give rise to any particular collection of state variables that could be interpreted as a run-time instance in the state of a simulator.

So, it is convenient to think of eventual state instances, and that is indeed how the reference interpreter works, but the model specification structure should avoid anything that is specific to this picture.

12.4.7 Type specification examples

Examples of type definitions using the various types of child element:

```
<ComponentType name="synapse">
   <EventPort direction="in" />
</ComponentType>
```

says that instances of components using this type can receive events.

```
<ComponentType name="HHChannel">
   <Children name="gates" type="HHGate" />
</ComponentType>
```

says that a HHChannel can have gates.

```
<ComponentType name="HHGate">
   <Child name="Forward" type="HHRate" />
   <Child name="Reverse" type="HHRate" />
</ComponentType>
```

says that a HHGate has two children called Forward and Reverse, each of type HHRate.

```
<ComponentType name="synapseCell">
   <Attachments name="synapses" type="synapse" />
</ComponentType>
```

says that instances of components based on the synapseCell type can have instances of component based on the synapse type attached to them at build time.

```
<ComponentType name="Population">
   <ComponentRef name="component" type="Component" />
</ComponentType>
```

says that components based on the Population type need a reference to a component of type Component (ie, anything) (which would then be used as the thing to be repeated in the population, but it doesn't say that here).

```
<ComponentType name="EventConnectivity">
   <Link name="source" type="Population" />
</ComponentType>
```

says that EventConnectivity components need a relative path to a local component of type Population which will be accessed via the name "source". The model component declarations corresponding to the channel and gate types would be:

```
<Component type="HHChanne">
   <Component type="HHGate">
      <Component type="some_hh_gate_type" role="Forward" />
      <Component type="some_hh_gate_type" role="Reverse" />
   </Component>
</Component>
```

or, in the shorthand notation:

```
<HHChannel>
   <HHGate>
      <Forward type="some_hh_gate_type" />
      <Reverse type="some_hh_gate_type" />
   </HHGate>
</HHChannel>
```

For the population type it would be:

```
<Component id="myPopulation" type="population" component="myCellModel" />
```

And for the connections:

```
<Component type="EventConnectivity" source="myPopulation" />
```

12.5 Example 1: Dimensions, Units, ComponentTypes and Components

This page is structured as a walk-through of a single example explaining the various elements as they occur.

```
<Lems>

    <Target component="sim1" />

    <Dimension name="voltage" m="1" l="2" t="-3" i="-1" />
    <Dimension name="time" t="1" />
    <Dimension name="per_time" t="-1" />
    <Dimension name="conductance" m="-1" l="-2" t="3" i="2" />
    <Dimension name="capacitance" m="-1" l="-2" t="4" i="2" />
    <Dimension name="current" i="1" />


    <ComponentType name="iaf1">
        <Parameter name="threshold" dimension="voltage" />
        <Parameter name="refractoryPeriod" dimension="time" />
        <Parameter name="capacitance" dimension="capacitance" />
    </ComponentType>


    <Unit symbol="mV" dimension="voltage" power="-3" />
    <Unit symbol="ms" dimension="time" power="-3" />
    <Unit symbol="pS" dimension="conductance" power="-12" />
    <Unit symbol="nS" dimension="conductance" power="-9" />
    <Unit symbol="uF" dimension="capacitance" power="-6" />
    <Unit symbol="nF" dimension="capacitance" power="-9" />
    <Unit symbol="pF" dimension="capacitance" power="-12" />
    <Unit symbol="per_ms" dimension="per_time" power="3" />
    <Unit symbol="pA" dimension="current" power="-12" />

    <iaf1 id="celltype_a" threshold="-30 mV" refractoryPeriod="2 ms" capacitance="3uF
↪" />
    <!-- or -->
    <Component id="ctb" type="iaf1" threshold="-30 mV" refractoryPeriod="2 ms"␣
↪capacitance="1uF" />


    <ComponentType name="iaf2" extends="iaf1">
        <Fixed parameter="threshold" value="-45mV" />
    </ComponentType>

    <ComponentType name="iaf3" extends="iaf1">
        <Parameter name="leakConductance" dimension="conductance" />
        <Parameter name="leakReversal" dimension="voltage" />
        <Parameter name="deltaV" dimension="voltage" />

        <EventPort name="spikes-in" direction="in" />
        <Exposure name="v" dimension="voltage" />

        <Dynamics>
            <StateVariable name="v" exposure="v" dimension="voltage" />
            <TimeDerivative variable="v" value="leakConductance * (leakReversal - v) /
```

(continues on next page)

(continued from previous page)

```
↪ capacitance" />

            <OnEvent port="spikes-in">
                <StateAssignment variable="v" value="v + deltaV" />
            </OnEvent>
        </Dynamics>

    </ComponentType>


    <ComponentType name="spikeGenerator">
        <Parameter name="period" dimension="time" />
        <EventPort name="a" direction="out" />
        <Exposure name="tsince" dimension="time" />
        <Dynamics>
            <StateVariable name="tsince" exposure="tsince" dimension="time" />
            <TimeDerivative variable="tsince" value="1" />
            <OnCondition test="tsince .gt. period">
                <StateAssignment variable="tsince" value="0" />
                <EventOut port="a" />
            </OnCondition>
        </Dynamics>
    </ComponentType>


    <ComponentType name="spikeGenerator2" extends="spikeGenerator">
        <Dynamics>
            <StateVariable name="tlast" dimension="time" />
            <DerivedVariable name="tsince" dimension="time" exposure="tsince" value=
↪"t - tlast" />
            <OnCondition test="t - tlast .gt. period">
                <StateAssignment variable="tlast" value="t" />
                <EventOut port="a" />
            </OnCondition>
        </Dynamics>
    </ComponentType>


    <ComponentType name="HHRate">
        <Parameter name="rate" dimension="per_time" />
        <Parameter name="midpoint" dimension="voltage" />
        <Parameter name="scale" dimension="voltage" />
        <Requirement name="v" dimension="voltage" />
        <Exposure name="r" dimension="per_time" />
    </ComponentType>


    <ComponentType name="HHExpRate" extends="HHRate">
        <Dynamics>
            <DerivedVariable name="r" dimension="per_time" exposure="r" value="rate *␣
↪exp((v - midpoint)/scale)" />
        </Dynamics>
    </ComponentType>


    <ComponentType name="HHSigmoidRate" extends="HHRate">
        <Dynamics>
```

(continues on next page)

(continued from previous page)

```
            <DerivedVariable name="r" dimension="per_time" exposure="r" value="rate /␣
↪(1 + exp( -(v - midpoint)/scale))" />
        </Dynamics>
    </ComponentType>


    <ComponentType name="HHExpLinearRate" extends="HHRate">
        <Dynamics>
            <DerivedVariable name="x" dimension="none" value="(v - midpoint) / scale"␣
↪/>
            <DerivedVariable name="r" dimension="per_time" exposure="r" value="rate *␣
↪x / (1 - exp(-x))" />
        </Dynamics>
    </ComponentType>


    <ComponentType name="HHGate0">
        <Parameter name="power" dimension="none" />
        <Child name="Forward" type="HHRate" />
        <Child name="Reverse" type="HHRate" />
        <Requirement name="v" dimension="voltage" />
        <Exposure name="fcond" dimension="none" />
        <Dynamics>
            <StateVariable name="q" dimension="none" />
            <DerivedVariable dimension="per_time" name="rf" select="Forward/r" />
            <DerivedVariable dimension="per_time" name="rr" select="Reverse/r" />
            <TimeDerivative variable="q" value="rf * (1 - q) - rr * q" />
            <DerivedVariable name="fcond" dimension="none" exposure="fcond" value="q^
↪power" />
        </Dynamics>
    </ComponentType>


    <ComponentType name="HHGate">
        <Parameter name="power" dimension="none" />
        <Child name="Forward" type="HHRate" />
        <Child name="Reverse" type="HHRate" />
        <Requirement name="v" dimension="voltage" />
        <Exposure name="fcond" dimension="none" />
        <Dynamics>
            <StateVariable name="x" dimension="none" />
            <DerivedVariable name="ex" dimension="none" value="exp(x)" />
            <DerivedVariable name="q" dimension="none" value="ex / (1 + ex)" />
            <DerivedVariable name="rf" dimension="per_time" select="Forward/r" />
            <DerivedVariable name="rr" dimension="per_time" select="Reverse/r" />
            <TimeDerivative variable="x" value="(1 + ex)^2 / ex * (rf * (1 - q) - rr␣
↪* q)" />
            <DerivedVariable name="fcond" dimension="none" exposure="fcond" value="q^
↪power" />
        </Dynamics>
    </ComponentType>


    <ComponentType name="HHChannel">
        <Parameter name="conductance" dimension="conductance" />
        <Children name="gates" type="HHGate" />
        <Exposure name="g" dimension="conductance" />
```

(continues on next page)

(continued from previous page)

```
        <Dynamics>
            <DerivedVariable name="gatefeff" dimension="none"
                             select="gates[*]/fcond" reduce="multiply" />
            <DerivedVariable name="g" exposure="g"
                             dimension="conductance" value="conductance * gatefeff" />
        </Dynamics>
    </ComponentType>


    <HHChannel id="na" conductance="20pS">
        <HHGate id="m" power="3">
            <Forward type="HHExpLinearRate" rate="1.per_ms"
                     midpoint="-40mV" scale="10mV" />
            <Reverse type="HHExpRate" rate="4per_ms" midpoint="-65mV"
                     scale="-18mV" />
        </HHGate>
        <HHGate id="h" power="1">
            <Forward type="HHExpRate" rate="0.07per_ms"
                     midpoint="-65.mV" scale="-20.mV" />
            <Reverse type="HHSigmoidRate" rate="1per_ms"
                     midpoint="-35mV" scale="10mV" />
        </HHGate>
    </HHChannel>


    <HHChannel id="k" conductance="20pS">
        <HHGate id="n" power="4">
            <Forward type="HHExpLinearRate" rate="0.1per_ms"
                     midpoint="-55mV" scale="10mV" />
            <Reverse type="HHExpRate" rate="0.125per_ms"
                     midpoint="-65mV" scale="-80mV" />
        </HHGate>
    </HHChannel>


    <ComponentType name="ChannelPopulation">
        <ComponentReference name="channel" type="HHChannel" />
        <Parameter name="number" dimension="none" />
        <Parameter name="erev" dimension="voltage" />
        <Requirement name="v" dimension="voltage" />
        <Exposure name="current" dimension="current" />

        <Dynamics>
            <DerivedVariable name="channelg" dimension="conductance" select="channel/g
↪" />
            <DerivedVariable name="geff" dimension="conductance" value="channelg *␣
↪number" />
            <DerivedVariable name="current" dimension="current" exposure="current"␣
↪value="geff * (erev - v)" />
        </Dynamics>

        <Structure>
            <ChildInstance component="channel" />
        </Structure>
    </ComponentType>
```

(continues on next page)

(continued from previous page)

```
    <ComponentType name="HHCell">
        <Parameter name="capacitance" dimension="capacitance" />
        <Children name="populations" type="ChannelPopulation" />
        <Parameter name="injection" dimension="current" />
        <Parameter name="v0" dimension="voltage" />
        <Exposure name="v" dimension="voltage" />
        <Dynamics>
            <OnStart>
                <StateAssignment variable="v" value="v0" />
            </OnStart>

            <DerivedVariable name="totcurrent"
                             dimension="current" select="populations[*]/current"
                             reduce="add" />
            <StateVariable name="v" exposure="v"
                           dimension="voltage" />
            <TimeDerivative variable="v"
                            value="(totcurrent + injection) / capacitance" />
        </Dynamics>
    </ComponentType>



    <HHCell id="hhcell_1" capacitance="1pF" injection="4pA" v0="-60mV">
        <ChannelPopulation channel="na" number="6000"  erev="50mV" />
        <ChannelPopulation channel="k" number="1800" erev="-77mV" />
    </HHCell>


    <Component id="celltype_c" type="iaf3" leakConductance="3 pS" refractoryPeriod="3␣
↪ms" threshold="45 mV" leakReversal="-50 mV" deltaV="5mV" capacitance="1uF" />

    <Component id="gen1" type="spikeGenerator" period="30ms" />

    <Component id="gen2" type="spikeGenerator2" period="32ms" />

    <Component id="iaf3cpt" type="iaf3" leakReversal="-50mV" deltaV="50mV" threshold=
↪"-30mV" leakConductance="50pS" refractoryPeriod="4ms" capacitance="1pF" />


    <Include file="SimpleNetwork.xml" />


    <Network id="net1">
        <Population id="p1" component="gen1" size="1" />
        <Population id="p2" component="gen2" size="1" />
        <Population id="p3" component="iaf3cpt" size="1" />

        <Population id="hhpop" component="hhcell_1" size="1" />

        <EventConnectivity id="p1-p3" source="p1" target="p3">
            <Connections type="AllAll" />
        </EventConnectivity>
    </Network>
```

(continues on next page)

(continued from previous page)

```
    <Include file="SingleSimulation.xml" />

    <Simulation id="sim1" length="80ms" step="0.01ms" target="net1">
        <Display id="d0" title="Example 1: Dimensions, Units, ComponentTypes and␣
↪Components"
                 timeScale="1ms" xmin="-10" xmax="90" ymin="-90" ymax="60">
            <Line id="tsince" quantity="p1[0]/tsince" scale="1ms" timeScale="1ms"␣
↪color="#00c000" />
            <Line id="p3v" quantity="p3[0]/v" scale="1mV" timeScale="1ms" color="
↪#0000f0" />
            <Line id="p0v" quantity="hhpop[0]/v" scale="1mV" timeScale="1ms" color="
↪#ff4040" />
        </Display>
    </Simulation>

</Lems>
```

The whole model is wrapped in a block which, for now, is called “Lems” (Low Entropy Model Specification). Then we define the dimensions that will be used in this model. Typically these would be loaded from an external file along with various other stuff, not repeated in each model, but it is included here in the interests of having a single file for everything.

```
<Dimension name="voltage" m="1" l="2" t="3" i="-1" />
<Dimension name="time" t="1" />
<Dimension name="per_time" t="-1" />
<Dimension name="conductance" m="-1" l="-2" t="3" i="2" />
<Dimension name="capacitance" m="-1" l="-2" t="4" i="2" />
<Dimension name="current" i="1" />
```

Each dimension element just associates a dimension name with the exponents for mass, length, time and current.

At this stage, one can begin defining component types. This is done with the ComponentType element and child Parameter elements. A simple cell model with three parameters could be defined as:

```
<ComponentType name="cell1">
   <Parameter name="threshold" dimension="voltage" />
   <Parameter name="refractoryPeriod" dimension="time" />
   <Parameter name="capacitance" dimension="capacitance" />
</ComponentType>
```

Each of the Parameter elements defines a parameter that should be supplied when a component is defined based on this type. Before we can define a component though, we need some units to use in setting those values.

Defining a unit involves supplying the symbol, dimension and the power of ten by which it is scaled from the IS base unit. Note that units have a symbol, not a name. This is because they occur as a component of an assignment expression such as ‘threshold=“-45mV”’ not as a reference such as ‘dimension=“voltage”’. In general, where one component refers to another, then the attribute value is the name of the thing being referred to, and the attribute name is the lower case version of the type of the thing being referred to. Thus when a dimension is declared with <Dimension name=“voltage”…/> then it is referred to from a Parameter as . This holds for all references to components of a particular type.

Returning to the units, this model will use the following, which normally would also be loaded from an external file of standard settings.

```
<Unit symbol="mV" dimension="voltage" powTen="-3" />
<Unit symbol="ms" dimension="time" powTen="-3" />
<Unit symbol="pS" dimension="conductance" powTen="-12" />
<Unit symbol="nS" dimension="conductance" powTen="-9" />
<Unit symbol="uF" dimension="capacitance" powTen="-6" />
```

(continues on next page)

(continued from previous page)

```
<Unit symbol="nF" dimension="capacitance" powTen="-9" />
<Unit symbol="pF" dimension="capacitance" powTen="-12" />
<Unit symbol="per_ms" dimension="per_time" powTen="3" />
<Unit symbol="pA" dimension="current" powTen="-12" />
```

Once the units are available it is possible to define a component. There are two equivalent ways of doing this: either by using the Component element and setting its type, or by using the type as a new XML element. The latter may be a little more readable, but for a simple component like this it doesn't make much difference. For more complicated components with nested children though, the second form is definitely clearer (eg see the HHChannel examples later).

```
<Component id="ctb" type="cell1" threshold="-30 mV" refractoryPeriod="2 ms"␣
 ↪capacitance="1uF" />
<cell1 id="celltype_a" threshold="-30 mV" refractoryPeriod="2 ms" capacitance="3uF" />
```

In specifying a component, a value must be supplied for each of the parameters defined in the corresponding type. The value is composed of a number and a unit. It can't include expressions with multiple units for the values so, for example, to express an acceleration you couldn't write "3 m s^-2". Instead would need to define a unit element for the compound unit (and a dimension element for acceleration) and use that.

Specifying all the parameters for each component can lead to duplication. Suppose, for example, you want to build a range of cell models all based on cell1, but you don't want to change the threshold. You could define a new type without the threshold, but it is neater to still use the same type but specify that you are restricting attention to the set that all have a particular value for the threshold. This can be done by creating a new type that extends the cell1 type and includes a Fixed element to fix the threshold:

```
<ComponentType name="cell2" extends="cell1">
   <Fixed parameter="threshold" value="-45mV" />
</ComponentType>
```

The cell2 type can now be used by only setting the remaining two parameters.

As well as restricting types when you extend them, you can also add new parameters as shown in the next type. This also introduces an EventPort, to indicate that instances of components built from this type can receive events, and, finally, a Dynamics block. This is where the Dynamics of instances of the component can be specified. The phrase "instances of the component" is intentional. The type itself doesn't "behave": it is just a definition. A component built from the type doesn't "behave" either: it is just a set of parameter values linked back to the type. The thing that "behaves" is an instance in a runnable model that actually contains state variables. In general, many components may be based on one type, and one component may give rise to many instances in a running model.

Here is a basic capacitative cell with a leaking potential and a simple event handler.

```
<ComponentType name="cell3" extends="cell1">
   <Parameter name="leakConductance" dimension="conductance" />
   <Parameter name="leakReversal" dimension="voltage" />
   <Parameter name="deltaV" dimension="voltage" />
   <EventPort name="spikes-in" direction="in" />
   <Exposure name="v" dimension="voltage" />
   <Dynamics>
      <StateVariable name="v" exposure="v" dimension="voltage" />
      <TimeDerivative variable="v" value="leakConductance * (leakReversal - v) /␣
 ↪capacitance" />
      <OnEvent port="spikes-in">
         <StateAssignment variable="v" value="v + deltaV" />
      </OnEvent>
   </Dynamics>
</ComponentType>
```

The Dynamics involves a single state variable, a voltage called "v", and one equation, expressing how v drifts towards the leak reversal potential. The event block specifies what happens when an instance receives an event. In this case the state variable v is bumped up by deltaV. The value attribute in the TimeDerivative element is an expression involving the parameters and the state variables. It gives the right hand side of a first order differential equation dv/dt = (…). Expressions follow normal operator precedence rules with "^" for general powers and exp(x) for exponentials.

Below is another example of a Dynamics, this time with an output port and a condition testing block that sends an event when the condition becomes true. The test attribute in the OnCondition element is a boolean valued expression. These use Fortran style operators (.gt. and .lt. for > and <) to avoid confusion with xml angle brackets.

```
<ComponentType name="spikeGenerator">
   <Parameter name="period" dimension="time" />
   <EventPort name="a" direction="out" />
   <Exposure name="tsince" dimension="time" />
   <Dynamics>
      <StateVariable name="tsince" exposure="tsince" dimension="time" />
      <TimeDerivative variable="tsince" value="1" />
      <OnCondition test="tsince .gt. period">
         <StateAssignment variable="tsince" value="0" />
         <EventOut port="a" />
      </OnCondition>
   </Dynamics>
</ComponentType>
```

The above model is one way of writing a regular event generator. It has a state variable that grows in sync with t until it reaches a threshold when the event fires and it is reset. The model below achieves the same effect without solving a differential equation. Instead, it asks for access to the global time variable ("t" is the one global variable that is always available) and uses that in the test condition. [aside - there's a slight problem here since t exists even if the model doesn't define a dimension called time].

```
<ComponentType name="spikeGenerator2" extends="spikeGenerator">
   <Dynamics>
      <GlobalVariable name="t" dimension="time" />
      <StateVariable name="tlast" dimension="time" />
      <DerivedVariable name="tsince" exposure="tsince" value="t - tlast" />
      <OnCondition test="t - tlast .gt. period">
         <StateAssignment variable="tlast" value="t" />
         <EventOut port="a" />
      </OnCondition>
   </Dynamics>
</ComponentType>
```

The examples so far have all been of very simple components which just had a single set of parameters. Real models however require rather more structure than this with components having children of various types and possibly multiple children of certain types. To illustrate this, the next example shows how the concept of an ion channel using Hodgkin-Huxley Dynamics can be defined.

Starting from the bottom, we define the different types of rate equations that occur. These will supply terms in the equations for the derivatives of the gating particles. There are three different expressions used in the HH equations, but they can all be expressed with three parameters, rate, midpoint and scale. We first define a general rate class, and then extend it for the three cases.

The HHRate Dynamics shows two new constructs. An Exposure declares that the component makes a quantity available to other components. A Requirement specifies that the component needs to know about a variable that it doesn't define itself. When it is used in a model, the specified variable must be available (and have the right dimension) in the parent component or one of its more remote ancestors.

Note that the general HHRate class defines an Exposure without a Dynamics block to actually set its value. This is

analogous to an abstract class in java: you can't actually make a component out of the HHRate element directly (the interpreter will complain) but any component using a HHRate will know it has an exposed variable called "r". The types that extend HHRate have to supply a value for "r" before they are fully defined and ready to be used.

So here is the basic HHRate and its three extensions:

```
<ComponentType name="HHRate">
   <Parameter name="rate" dimension="per_time" />
   <Parameter name="midpoint" dimension="voltage" />
   <Parameter name="scale" dimension="voltage" />
   <Exposure name="r" dimension="per_time" />
</ComponentType>
<ComponentType name="HHExpRate" extends="HHRate">
   <Dynamics>
      <DerivedVariable name="r" exposure="r" value="rate * exp((v - midpoint)/scale)"␣
 ↪/>
   </Dynamics>
</ComponentType>
<ComponentType name="HHSigmoidRate" extends="HHRate">
   <Dynamics>
      <DerivedVariable name="r" exposure="r" value="rate / (1 + exp(0 - (v -␣
 ↪midpoint)/scale))" />
   </Dynamics>
</ComponentType>
<ComponentType name="HHExpLinearRate" extends="HHRate">
   <Dynamics>
      <DerivedVariable name="x" value="(v - midpoint) / scale" />
      <DerivedVariable name="r" exposure="r" value="rate * x / (1 - exp(0 - x))" />  ␣
 ↪
   </Dynamics>
</ComponentType>
```

Now the rate elements are available, they can be used to define a component for a gate in a HH model. This introduces the Child element which says that components built using this type must include a subcomponent of the specified type. A HH gate needs subcomponents for the forward and reverse rates.

```
<ComponentType name="HHGate0">
   <Parameter name="power" dimension="none" />
   <Child name="Forward" type="HHRate" />
   <Child name="Reverse" type="HHRate" />
   <Exposure name="fcond" dimension="none" />
   <Requirement name="v" dimension="voltage" />
   <Dynamics>
      <StateVariable name="q" dimension="none" />
      <DerivedVariable name="rf" select="Forward/r" />
      <DerivedVariable name="rr" select="Reverse/r" />
      <TimeDerivative variable="q" value="rf * (1 - q) - rr * q" />
      <DerivedVariable name="fcond" exposure="fcond" value="q^power" />
   </Dynamics>
</ComponentType>
```

The above is a perfectly reasonable way to define a HH gate but unfortunately it needs smarter numerics than the simple forward Euler rule used in the proof of concept interpreter. Running this model with the Euler method leads to numerical instabilities. Happily, this problem can be circumvented without improving the numerics by changing the state variable. Instead of q which is defined on [0, 1] you can use x defined on (-infinity, infinity) which works much better with a naive integration scheme. This is what it looks like with x instead of q:

```
<ComponentType name="HHGate">
   <Parameter name="power" dimension="none" />
   <Child name="Forward" type="HHRate" />
   <Child name="Reverse" type="HHRate" />
   <Exposure name="fcond" dimension="none" />
   <Requirement name="v" dimension="voltage" />
   <Dynamics>
      <StateVariable name="x" dimension="none" />
      <DerivedVariable name="ex" dimension="none" value="exp(x)" />
      <DerivedVariable name="q" dimension="none" value="ex / (1 + ex)" />
      <DerivedVariable name="rf" select="Forward/r" />
      <DerivedVariable name="rr" select="Reverse/r" />
      <TimeDerivative variable="x" value="(1 + ex)^2 / ex * (rf * (1 - q) - rr * q)" /
↪>
      <DerivedVariable name="fcond" exposure="fcond" value="q^power" />
   </Dynamics>
</ComponentType>
```

Now the gate type has been defined, it can be used to say what a HH Channel actually is. In this picture, a channel just has a conductance and one or more gates:

```
<ComponentType name="HHChannel">
   <Parameter name="conductance" dimension="conductance" />
   <Children name="gates" type="HHGate" min="0" max="4" />
   <Requirement name="v" dimension="voltage" />
   <Exposure name="g" dimension="conductance" />
   <Dynamics>
      <DerivedVariable name="gatefeff" select="gates[*]/fcond" reduce="multiply" /> ␣
↪
      <DerivedVariable name="g" exposure="g" value="conductance * gatefeff" />
   </Dynamics>
</ComponentType>
```

This introduces one new construct, the Children element, that allows for an indeterminate number of children of a given type. This means that the same type can be used for potassium channels with only one gate, sodium channels with two gates or indeed other channels with more gates. The first derived variable in the Dynamics block uses a xpath-style selection function to process the indeterminate number of children. In this case it computes the produce of the fcond variables from the different gates.

With these definitions in place, it is now possible to define some channel models. The classic Hodgkin-Huxley sodium channel can be represented as:

```
<HHChannel id="na" conductance="20pS">
   <HHGate id="m" power="3">
      <Forward type="HHExpLinearRate" rate="1.per_ms" midpoint="-40mV" scale="10mV" />
↪
      <Reverse type="HHExpRate" rate="4per_ms" midpoint="-65mV" scale="-18mV" />
   </HHGate>
   <HHGate id="h" power="1">
      <Forward type="HHExpRate" rate="0.07per_ms" midpoint="-65.mV" scale="-20.mV" />␣
↪
      <Reverse type="HHSigmoidRate" rate="1per_ms" midpoint="-35mV" scale="10mV" /> ␣
↪
   </HHGate>
</HHChannel>
```

The potassium channel uses exactly the same types, but has only one gate:

```
<HHChannel id="k" conductance="20pS">
   <HHGate id="n" power="4">
      <Forward type="HHExpLinearRate" rate="0.1per_ms" midpoint="-55mV" scale="10mV" /
 ↪>
      <Reverse type="HHExpRate" rate="0.125per_ms" midpoint="-65mV" scale="-80mV" /> ␣
 ↪
   </HHGate>
</HHChannel>
```

These channel models are an example where the ability to use the type name as the XML tag makes the model much clearer: the alternative just with three levels of Component elements would look rather unhelpful.

Although the channel models have now been defined, they still need to be used in a cell before anything can be run. For this we'll just define a basic channel population type. There is one new construct here: the ComponentRef element which in this case says that a channel population needs a reference to a component of type HHChannel. This is much like a Child element, but instead of the component being defined then and there inside the channel population, there is just a reference to it.

The Dynamics block for a cannel population just computes the total conductance and then the current, in this case using Ohm's law.

```
<ComponentType name="ChannelPopulation">
   <ComponentRef name="channel" type="HHChannel" />
   <Parameter name="number" dimension="none" />
   <Parameter name="erev" dimension="voltage" />
   <Requirement name="v" dimension="voltage" />
   <Exposure name="current" dimension="current" />
   <Dynamics>
      <DerivedVariable name="channelg" select="channel/g" />
      <DerivedVariable name="geff" value="channelg * number" />
      <DerivedVariable name="current" exposure="current" value="geff * (erev - v)" />␣
 ↪
   </Dynamics>
</ComponentType>
```

To use these populations, they need inserting in a cell. The following type represents a simple cell with a number of populations and an option to inject a current so it does something interesting.

```
<ComponentType name="HHCell">
   <Parameter name="capacitance" dimension="capacitance" />
   <Children name="populations" type="ChannelPopulation" />
   <Parameter name="injection" dimension="current" />
   <Parameter name="v0" dimension="voltage" />
   <Exposure name="v" dimension="voltage" />
   <Dynamics>
      <OnStart>
         <StateAssignment variable="v" value="v0" />
      </OnStart>
      <DerivedVariable name="totcurrent" select="populations[*]/current" reduce="add"␣
 ↪/>
      <StateVariable name="v" dimension="voltage" />
      <TimeDerivative variable="v" value="(totcurrent + injection) / capacitance" /> ␣
 ↪
   </Dynamics>
</ComponentType>
```

This Dynamics block introduces the OnStart element which is much like the OnEvent elements earlier, except the block applies only when the simulation starts. In this case it just sets the voltage to a value supplied as a parameter. The

Dynamics block uses another selector function "sum(…" to sum the currents delivered by the various populations.

Now all the definitions are in place to define a cell model with a couple of channel populations:

```
<HHCell id="hhcell_1" capacitance="1pF" injection="4pA" v0="-60mV">
   <ChannelPopulation channel="na" number="6000" erev="50mV" />
   <ChannelPopulation channel="k" number="1800" erev="-77mV" />
</HHCell>
```

To go with this cell type, we can define some components using the other types defined earlier. Note how celltype_d is based on an existing component via the "extends" attribute and only replaces one parameter value.

```
<Component id="celltype_c" type="iaf3" leakConductance="3 pS" refractoryPeriod="3 ms"␣
 ↪threshold="45 mV" leakReversal="-50 mV" deltaV="5mV" capacitance="1uF" />
<Component id="celltype_d" extends="celltype_c" leakConductance="5 pS" />
<Component id="gen1" type="spikeGenerator" period="30ms" />
<Component id="gen2" type="spikeGenerator2" period="32ms" />
<Component id="cell3cpt" type="cell3" leakReversal="-50mV" deltaV="50mV" threshold="-
 ↪30mV" leakConductance="50pS" refractoryPeriod="4ms" capacitance="1pF" />
```

Finally a simulation element says what component is to be run and for how long. It also contains an embedded display element so the results of the simulation can be visualized. These are also user-defined types: their definitions will be presented in example 6.

```
<Simulation length="80ms" step="0.05ms" target="hhcell_1">
   <Display unit="ms">
      <Line quantity="v" unit="mV" color="#0000f0" />
   </Display>
</Simulation>
```

That's it. When this model is run it produces the figure shown below (after rescaling a bit).

12.6 Example 2: tidying up example 1

This models is the same as in example 1, except that the definitions have been split out into several self-contained files.

The main file, included below, uses the Include element to include definitions from other files. Each file is only read once, even if several files include it. Because some of these files, such as the HH channel definitions, are intended to be used on their own, they include all the dimension definitions they need. These may also occur in other files with the same dimension names. This is fine as long as the dimensions being declared are the same. An error will be reported if a new definition is supplied that changes any of the values. The same applies for Unit definitions. For other element types names and ids must be unique. An id or name can't appear twice, even if the content of the elements is the same.

12.7 Main model

This defines a few components, then a network that uses them and a simulation to run it all. The HHCell component refers to channel types coming from the included hhmodels.xml file which in turn depends on hhcell.xml and hhchannel.xml.

```
<Lems>

    <Target component="sim1"/>

    <Include file="ex2dims.xml"/>
```

(continues on next page)

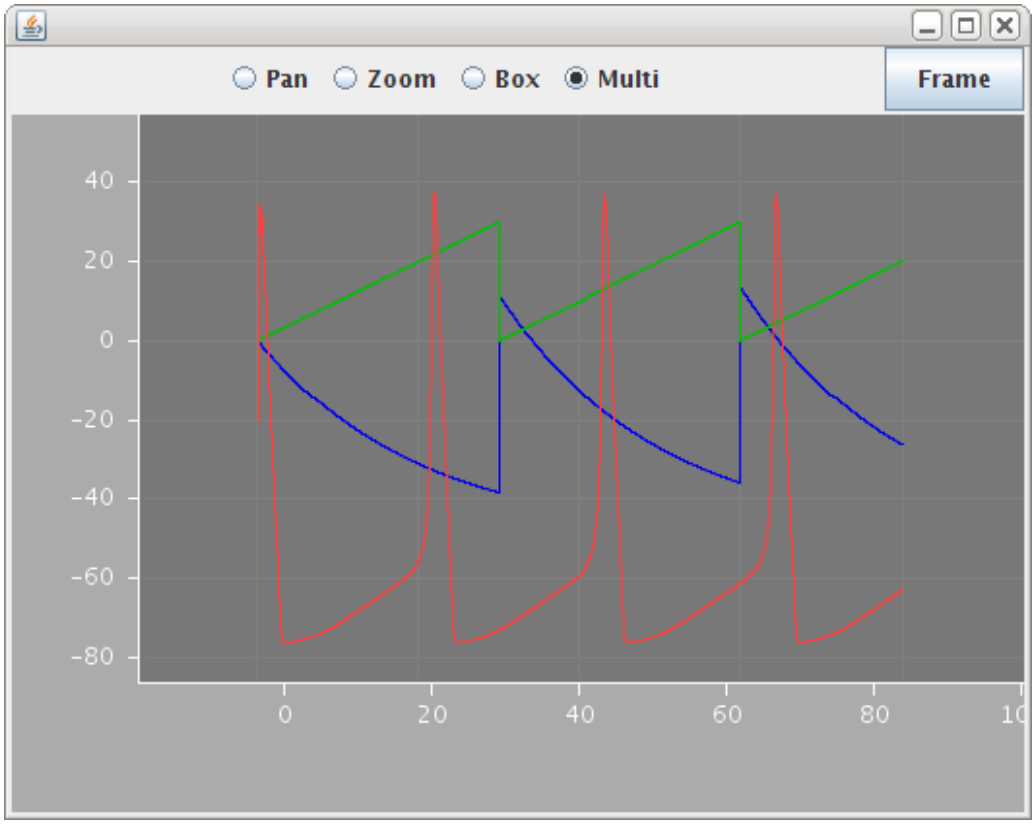


Fig. 12.1: LEMS GUI showing simulation output graphs

(continued from previous page)

```
    <Include file="hhchannel.xml"/>

    <Include file="hhcell.xml"/>
    <Include file="spikegenerators.xml"/>
    <Include file="hhmodels.xml"/>
    <Include file="misciaf.xml"/>

    <Include file="SimpleNetwork.xml"/>

    <HHCell id="hhcell_1" capacitance="1pF" injection="4pA" v0="-60mV">
        <ChannelPopulation channel="na" number="6000" erev="50mV"/>
        <ChannelPopulation channel="k" number="1800" erev="-77mV"/>
    </HHCell>

    <Component id="gen1" type="spikeGenerator" period="30ms"/>

    <Component id="gen2" type="spikeGenerator2" period="32ms"/>

    <Component id="iaf3cpt" type="iaf3" leakReversal="-50mV" deltaV="50mV" threshold=
↪"-30mV" leakConductance="50pS"
           refractoryPeriod="4ms" capacitance="1pF"/>


    <Network id="net1">
        <Population id="p1" component="gen1" size="1"/>
        <Population id="p2" component="gen2" size="1"/>
        <Population id="p3" component="iaf3cpt" size="1"/>

        <Population id="hhpop" component="hhcell_1" size="1"/>


        <EventConnectivity id="p1-p3" source="p1" target="p3">
            <Connections type="AllAll"/>
        </EventConnectivity>
    </Network>

    <Include file="SingleSimulation.xml" />

    <Simulation id="sim1" length="80ms" step="0.01ms" target="net1">
        <Display id="d0" title="Example 2" timeScale="1ms" xmin="-10" xmax="90" ymin=
↪"-90" ymax="60">
            <Line id="tsince" quantity="p1[0]/tsince" scale="1ms" timeScale="1ms"␣
↪color="#00c000" />
            <Line id="p3v" quantity="p3[0]/v" scale="1mV" timeScale="1ms" color="
↪#0000f0" />
            <Line id="p0v" quantity="hhpop[0]/v" scale="1mV" timeScale="1ms" color="
↪#ff4040" />
        </Display>
    </Simulation>

</Lems>
```

12.8 Included files

```
<Lems>

    <Dimension name="voltage" m="1" l="2" t="-3" i="-1"/>
    <Dimension name="time" t="1"/>
    <Dimension name="per_time" t="-1"/>
    <Dimension name="conductance" m="-1" l="-2" t="3" i="2"/>
    <Dimension name="capacitance" m="-1" l="-2" t="4" i="2"/>
    <Dimension name="current" i="1"/>
    <Dimension name="temperature" k="1"/>

    <Unit symbol="mV" dimension="voltage" power="-3"/>
    <Unit symbol="ms" dimension="time" power="-3"/>
    <Unit symbol="pS" dimension="conductance" power="-12"/>
    <Unit symbol="nS" dimension="conductance" power="-9"/>
    <Unit symbol="uF" dimension="capacitance" power="-6"/>
    <Unit symbol="nF" dimension="capacitance" power="-9"/>
    <Unit symbol="pF" dimension="capacitance" power="-12"/>
    <Unit symbol="per_ms" dimension="per_time" power="3"/>
    <Unit symbol="pA" dimension="current" power="-12"/>
    <Unit symbol="nA" dimension="current" power="-9"/>
    <Unit symbol="degC" dimension="temperature" offset="273.15"/>

</Lems>
```

The file hhchannel.xml contains complete definitions of a fairly general HH-style channel model with any number of gates based on the three standard types used in the original HH work.

```
<Lems>

    <Dimension name="voltage" m="1" l="2" t="-3" i="-1"/>
    <Dimension name="time" t="1"/>
    <Dimension name="per_time" t="-1"/>
    <Dimension name="conductance" m="-1" l="-2" t="3" i="2"/>
    <Dimension name="capacitance" m="-1" l="-2" t="4" i="2"/>
    <Dimension name="current" i="1"/>


    <ComponentType name="HHRate">
        <Parameter name="rate" dimension="per_time"/>
        <Parameter name="midpoint" dimension="voltage"/>
        <Parameter name="scale" dimension="voltage"/>
        <Requirement name="v" dimension="voltage"/>
        <Exposure name="r" dimension="per_time"/>
    </ComponentType>


    <ComponentType name="HHExpRate" extends="HHRate">
        <Dynamics>
            <DerivedVariable name="r" exposure="r"  dimension="per_time" value="rate␣
↪* exp((v - midpoint)/scale)"/>
        </Dynamics>
    </ComponentType>


    <ComponentType name="HHSigmoidRate" extends="HHRate">
```

(continues on next page)

(continued from previous page)

```
        <Dynamics>
            <DerivedVariable name="r" dimension="per_time" exposure="r" value="rate /␣
↪(1 + exp( -(v - midpoint)/scale))"/>
        </Dynamics>
    </ComponentType>


    <ComponentType name="HHExpLinearRate" extends="HHRate">
        <Dynamics>
            <DerivedVariable name="x" dimension="none" value="(v - midpoint) / scale"/
↪>
            <DerivedVariable name="r" dimension="per_time" exposure="r" value="rate *␣
↪x / (1 - exp(-x))"/>
        </Dynamics>
    </ComponentType>


    <ComponentType name="HHGate0">
        <Parameter name="power" dimension="none"/>
        <Child name="Forward" type="HHRate"/>
        <Child name="Reverse" type="HHRate"/>
        <Requirement name="v" dimension="voltage"/>
        <Exposure name="fcond" dimension="none"/>

        <Dynamics simultaneous="true">
            <StateVariable name="q" dimension="none"/>
            <DerivedVariable dimension="per_time" name="rf" select="Forward/r"/>
            <DerivedVariable dimension="per_time" name="rr" select="Reverse/r"/>
            <TimeDerivative variable="q" value="rf * (1 - q) - rr * q"/>
            <DerivedVariable name="fcond" dimension="none" exposure="fcond" value="q^
↪power"/>
        </Dynamics>
    </ComponentType>


    <Include file="hhaltgate.xml"/>


    <ComponentType name="HHChannel">
        <Parameter name="conductance" dimension="conductance"/>
        <Children name="gates" type="HHGate"/>
        <Exposure name="g" dimension="conductance"/>
        <Dynamics simultaneous="false">
            <DerivedVariable name="gatefeff" dimension="none" select="gates[*]/fcond"␣
↪reduce="multiply"/>
            <DerivedVariable name="g" exposure="g" dimension="conductance" value=
↪"conductance * gatefeff"/>
        </Dynamics>
    </ComponentType>


</Lems>
```

As mentioned in example1, the numerics are too feeble to cope with this gate definition though, so a change of variables is employed instead:

```
<Lems>

    <ComponentType name="HHGate">
        <Parameter name="power" dimension="none"/>
        <Child name="Forward" type="HHRate"/>
        <Child name="Reverse" type="HHRate"/>
        <Requirement name="v" dimension="voltage"/>
        <Exposure name="fcond" dimension="none"/>


        <Dynamics simultaneous="false">
            <StateVariable name="x" dimension="none"/>
            <DerivedVariable name="ex" dimension="none" value="exp(x)"/>
            <DerivedVariable name="q" dimension="none" value="ex / (1 + ex)"/>
            <DerivedVariable name="rf" dimension="per_time" select="Forward/r"/>
            <DerivedVariable name="rr" dimension="per_time" select="Reverse/r"/>

            <TimeDerivative variable="x" value="(1 + ex)^2 / ex * (rf * (1 - q) - rr␣
↪* q)"/>

            <DerivedVariable name="fcond" dimension="none" exposure="fcond" value="q^
↪power"/>
        </Dynamics>
    </ComponentType>

</Lems>
```

The file hhcell.xml defines a simple cell model with some populations of HH channels.

```
<Lems>

    <Include file="hhchannel.xml"/>

    <Dimension name="voltage" m="1" l="2" t="-3" i="-1"/>
    <Dimension name="capacitance" m="-1" l="-2" t="4" i="2"/>
    <Dimension name="current" i="1"/>


    <ComponentType name="ChannelPopulation">
        <ComponentReference name="channel" type="HHChannel"/>
        <Parameter name="number" dimension="none"/>
        <Parameter name="erev" dimension="voltage"/>
        <Requirement name="v" dimension="voltage"/>
        <Exposure name="current" dimension="current"/>
        <Exposure name="geff" dimension="conductance"/>

        <Structure>
            <ChildInstance component="channel"/>
        </Structure>

        <Dynamics simultaneous="false">
            <DerivedVariable name="channelg" dimension="conductance" select="channel/g
↪"/>
            <DerivedVariable name="geff" exposure="geff" value="channelg * number"/>
            <DerivedVariable name="current" exposure="current" value="geff * (erev -␣
↪v)"/>
        </Dynamics>
```

(continues on next page)

(continued from previous page)

```
    </ComponentType>


    <ComponentType name="HHCell">
        <Parameter name="capacitance" dimension="capacitance"/>
        <Children name="populations" type="ChannelPopulation"/>
        <Parameter name="injection" dimension="current"/>
        <Parameter name="v0" dimension="voltage"/>
        <Exposure name="v" dimension="voltage"/>

        <Dynamics simultaneous="true">
            <OnStart>
                <StateAssignment variable="v" value="v0"/>
            </OnStart>

            <DerivedVariable name="totcurrent" dimension="current" select=
↪"populations[*]/current" reduce="add"/>
            <StateVariable name="v" exposure="v" dimension="voltage"/>
            <TimeDerivative variable="v" value="(totcurrent + injection) / capacitance
↪"/>
        </Dynamics>
    </ComponentType>

</Lems>
```

A couple of spike generators.

```
<Lems>

    <Dimension name="time" t="1"/>


    <ComponentType name="spikeGenerator">
        <Parameter name="period" dimension="time"/>
        <EventPort name="a" direction="out"/>
        <Exposure name="tsince" dimension="time"/>
        <Dynamics>
            <StateVariable name="tsince" exposure="tsince" dimension="time"/>
            <TimeDerivative variable="tsince" value="1"/>
            <OnCondition test="tsince .gt. period">
                <StateAssignment variable="tsince" value="0"/>
                <EventOut port="a"/>
            </OnCondition>
        </Dynamics>
    </ComponentType>


    <ComponentType name="spikeGenerator2" extends="spikeGenerator">
        <Dynamics>
            <DerivedVariable name="tsince" exposure="tsince" value="t - tlast"/>
            <StateVariable name="tlast" dimension="time"/>
            <OnCondition test="t - tlast .gt. period">
                <StateAssignment variable="tlast" value="t"/>
                <EventOut port="a"/>
            </OnCondition>
        </Dynamics>
```

(continues on next page)

(continued from previous page)

```
    </ComponentType>


</Lems>
```

And now the components themselves. These are the standard HH sodium and potassium channels (as used in Rallpack3).

```
<Lems>
    <Include file="hhchannel.xml"/>

    <Unit symbol="mV" dimension="voltage" power="-3"/>
    <Unit symbol="per_ms" dimension="per_time" power="3"/>
    <Unit symbol="pS" dimension="conductance" power="-12"/>


    <HHChannel id="na" conductance="20pS">
        <HHGate id="m" power="3">
            <Forward type="HHExpLinearRate" rate="1.per_ms" midpoint="-40mV" scale=
↪"10mV"/>
            <Reverse type="HHExpRate" rate="4per_ms" midpoint="-65mV" scale="-18mV"/>
        </HHGate>

        <HHGate id="h" power="1">
            <Forward type="HHExpRate" rate="0.07per_ms" midpoint="-65.mV" scale="-20.
↪mV"/>
            <Reverse type="HHSigmoidRate" rate="1per_ms" midpoint="-35mV" scale="10mV
↪"/>
        </HHGate>
    </HHChannel>


    <HHChannel id="k" conductance="20pS">
        <HHGate id="n" power="4">
            <Forward type="HHExpLinearRate" rate="0.1per_ms" midpoint="-55mV" scale=
↪"10mV"/>
            <Reverse type="HHExpRate" rate="0.125per_ms" midpoint="-65mV" scale="-80mV
↪"/>
        </HHGate>
    </HHChannel>

</Lems>
```

Some miscellaneous iaf models.

```
<Lems>

    <Include file="elecdims.xml"/>


    <ComponentType name="iaf1">
        <Parameter name="threshold" dimension="voltage"/>
        <Parameter name="refractoryPeriod" dimension="time"/>
        <Parameter name="capacitance" dimension="capacitance"/>
    </ComponentType>


    <ComponentType name="iaf3" extends="iaf1">
```

(continues on next page)

(continued from previous page)

```
        <Parameter name="leakConductance" dimension="conductance"/>
        <Parameter name="leakReversal" dimension="voltage"/>
        <Parameter name="deltaV" dimension="voltage"/>
        <EventPort name="spikes-in" direction="in"/>
        <Exposure name="v" dimension="voltage"/>

        <Dynamics>
            <StateVariable name="v" exposure="v" dimension="voltage"/>
            <TimeDerivative variable="v" value="leakConductance * (leakReversal - v) /
↪ capacitance"/>

            <OnEvent port="spikes-in">
                <StateAssignment variable="v" value="v + deltaV"/>
            </OnEvent>
        </Dynamics>

    </ComponentType>


</Lems>
```

Finally, a small collection of dimension definitions useful for things like the miscellaneous iaf cell definitions.

```
<Lems>
    <Dimension name="voltage" m="1" l="2" t="-3" i="-1"/>
    <Dimension name="time" t="1"/>
    <Dimension name="conductance" m="-1" l="-2" t="3" i="2"/>
    <Dimension name="capacitance" m="-1" l="-2" t="4" i="2"/>
    <Dimension name="current" i="1"/>

</Lems>
```

12.9 Example 3: Connection dependent synaptic components

In many models, a synapse is only created where a connection exists. This means that the model of the receiving cell should only declare that particular types of synapse can be added to it, not the actual synapse sub-components themselves.

Not much is needed beyond the elements described in example 1 except for some extensions to the component that declares the connectivity and a new child element in the component that the synapses are attached to. The full example is shown below. The synapse type includes an EventPort just like the previously defined cell type. The cell type however includes a new child element: Attachments defined as:

```
<Attachments name="synapses" type="synapse" />
```

This operates rather like the Children element except that when a component is defined using this type the sub-elements are not included in the component definition. Instead it indicates that instances of components of the particular type may be attached later when the model is actually run.

12.10 Example 4: Kinetic schemes

The existing components provide everything necessary to define types that allow a model to specify a kinetic scheme (Markov model). The missing ingredient is the Dynamics element to actually expresses how instances of the components develop through time.

First then, the following definitions can be used to express ion channel models where the channel state is represented by an occupancy vector among a number of distinct states with rates for the transitions between states.

```
<ComponentType name="KSGate">
   <Parameter name="power" dimension="none" />
   <Parameter name="deltaV" dimension="voltage" />
   <Children name="states" type="KSState" />
   <Children name="transitions" type="KSTransition" />
</ComponentType>
<ComponentType name="KSState">
   <Parameter name="relativeConductance" dimension="none" />
   <Dynamics>
      <StateVariable name="occupancy" dimension="none" />
      <DerivedVariable name="q" value="relativeConductance * occupancy" />
   </Dynamics>
</ComponentType>
<ComponentType name="KSClosedState" extends="KSState">
   <Fixed parameter="relativeConductance" value="0" />
</ComponentType>
<ComponentType name="KSOpenState" extends="KSState">
   <Fixed parameter="relativeConductance" value="1" />
</ComponentType>
<ComponentType name="KSTransition">
   <Link name="from" type="KSState" />
   <Link name="to" type="KSState" />
   <Requirement name="v" dimension="voltage" />
   <Exposure name="rf" dimension="per_time" />
   <Exposure name="rr" dimension="per_time" />
</ComponentType>
<ComponentType name="KSChannel">
   <Parameter name="conductance" dimension="conductance" />
   <Children name="gates" type="KSGate" />
   <Exposure name="g" dimension="conductance" />
   <Dynamics>
      <DerivedVariable name="fopen" dimension="none" select="gates[*]/fopen" reduce=
 ↪"multiply" />
      <DerivedVariable name="g" exposure="g" dimension="conductance" value="fopen *␣
 ↪conductance" />
   </Dynamics>
</ComponentType>
```

This says that a gate can contain any number of states and transitions. A state has an occupancy variable, and a transition has links to two states giving the source and target states for the transition.

The transition element here is an abstract element because it doesn't provide a Dynamics block but just specifies what quantities transitions should privide via the two exposures. One of the most useful forms of transition is a damped Boltzman equation which can be parameterizd as follows:

```
<ComponentType name="VHalfTransition" extends="KSTransition">
   <Parameter name="vHalf" dimension="voltage" />
   <Parameter name="z" dimension="none" />
```

(continues on next page)

(continued from previous page)

```
   <Parameter name="gamma" dimension="none" />
   <Parameter name="tau" dimension="time" />
   <Parameter name="tauMin" dimension="time" />
   <Constant name="kte" dimension="voltage" value="25.3mV" />
   <Requirement name="v" dimension="voltage" />
   <Dynamics>
      <DerivedVariable name="rf0" dimension="per_time" value="exp(z * gamma * (v -␣
↪vHalf) / kte) / tau" />
      <DerivedVariable name="rr0" dimension="per_time" value="exp(-z * (1 - gamma) *␣
↪(v - vHalf) / kte) / tau" />
      <DerivedVariable name="rf" exposure="rf" dimension="per_time" value="1 / (1/rf0␣
↪+ tauMin)" />
      <DerivedVariable name="rr" exposure="rr" dimension="per_time" value="1 / (1/rr0␣
↪+ tauMin)" />
   </Dynamics>
</ComponentType>
```

Given these definitions, we can express a couple of simple channel models that use kinetic schemes. There is nothing special about these models. They are just examples used in PSICS that produce spikes (albeit rather unnatural looking ones) when used together.

```
<KSChannel id="na1" conductance="20pS">
   <KSGate power="1" deltaV="0.1mV">
      <KSClosedState id="c1" />
      <KSClosedState id="c2" />
      <KSOpenState id="o1" relativeConductance="1" />
      <KSClosedState id="c3" />
      <VHalfTransition from="c1" to="c2" vHalf="-35mV" z="2.5" gamma="0.8" tau="0.15ms
↪" tauMin="0.001ms" />
      <VHalfTransition from="c2" to="o1" vHalf="-35mV" z="2.5" gamma="0.8" tau="0.15ms
↪" tauMin="0.001ms" />
      <VHalfTransition from="o1" to="c3" vHalf="-70mV" z="1.1" gamma="0.90" tau="8.0ms
↪" tauMin="0.01ms" />
   </KSGate>
</KSChannel>
<KSChannel id="k1" conductance="30pS">
   <KSGate power="1" deltaV="0.1mV">
      <KSClosedState id="c1" />
      <KSOpenState id="o1" />
      <VHalfTransition from="c1" to="o1" vHalf="0mV" z="1.5" gamma="0.75" tau="3.2ms"␣
↪tauMin="0.3ms" />
   </KSGate>
</KSChannel>
```

This has all been done with the existing components. They allow types to be defined for expressing kinetic schemes, and models can be expressed that use these types, but there is nothing so far that says that the model actually is governed by a kinetic scheme. In particular, there is an "occupancy" state variable in each state element for which there is no governing equation and the rates generate "rf" and "rr" quantities that are not unused anywhere.

What is needed is a new element in the Dynamics block to link these together and say that the rates apply to the changes of occupancy among the state elements. This is done by adding a KineticScheme element to the Dynamics block for a gate as follows (this now shows the full definition of the KSGate element):

```
<ComponentType name="KSGate">
   <Parameter name="power" dimension="none" />
   <Parameter name="deltaV" dimension="voltage" />
```

(continues on next page)

(continued from previous page)

```
   <Children name="states" type="KSState" />
   <Children name="transitions" type="KSTransition" />
   <Dynamics>
      <KineticScheme name="ks">
         <Nodes children="states" variable="occupancy" />
         <Edges children="transitions" sourceNodeName="from" targetNodeName="to"␣
↪forwardRate="rf" reverseRate="rr" />
         <Tabulable variable="v" increment="deltaV" />
      </KineticScheme>
      <DerivedVariable name="q" dimension="none" select="states[*]/q" reduce="add" />
      <DerivedVariable name="fopen" exposure="fopen" dimension="none" value="q^power"␣
↪/>
   </Dynamics>
</ComponentType>
```

The new part here is the KineticScheme element and its children Nodes, Edges and Tabulable. The Nodes element says which elements in the parent container are goverened by the scheme, and which variable in those elements represents the relative occupancy.

The Edges element is a little more complicated. It has to say not only which elements define the transitions, but how the fields in the transitions map to things the scheme knows about. For a transition in a kinetic scheme, you need to know which state the transition comes from, which it goes to, and how fast it goes. It is possible (as here) that a single transition defines both directions, in which case it must also say which variable in the target objects provides the reverse transition rates. This is what the last four attributes of the Edges element do.

The Tabulable element is a temporary convenience for implementation purposes. In this case it says that the rates depend only on v and that the transition matrices can be cached an reused on a grid of spacing deltaV rather than recomputed every time. This is not used in the 0.2.1 version of the interpreter.

Note that the KineticScheme element doesn't say anything about what the outputs are. All it does is control the occupancy state variable in the state elements. The interpretation of these quantities is specified in the normal way with the two DerivedVaraible declarations. No special elements are needed in the scheme itself.

To actually use these models we need cell and population elements to link them all together. There is nothing new here - it all works just as for HH channels. The rest of the example4.xml file is:

```
<ComponentType name="ChannelPopulation">
   <ComponentRef name="channel" type="KSChannel" />
   <Parameter name="number" dimension="none" />
   <Parameter name="erev" dimension="voltage" />
   <Requirement name="v" dimension="voltage" />
   <Dynamics>
      <DerivedVariable name="channelg" dimension="conductance" select="channel/g" />
      <DerivedVariable name="geff" value="channelg * number" />
      <DerivedVariable name="current" value="geff * (erev - v)" />
   </Dynamics>
</ComponentType>
<ComponentType name="KSCell">
   <Parameter name="capacitance" dimension="capacitance" />
   <Children name="populations" type="ChannelPopulation" />
   <Parameter name="injection" dimension="current" />
   <Parameter name="v0" dimension="voltage" />
   <Dynamics>
      <OnStart>
         <StateAssignment variable="v" value="v0" />
      </OnStart>
      <DerivedVariable name="totcurrent" dimension="current" select=
```

(continues on next page)

(continued from previous page)

```
 ↪"sum(populations[*]/current)" />
      <StateVariable name="v" dimension="voltage" />
      <TimeDerivative variable="v" value="(totcurrent + injection) / capacitance" />
   </Dynamics>
</ComponentType>
<KSCell id="kscell_1" capacitance="1pF" injection="1pA" v0="-60mV">
   <ChannelPopulation channel="na1" number="600" erev="50mV" />
   <ChannelPopulation channel="k1" number="180" erev="-77mV" />
</KSCell>

<Network id="net1">
   <XPopulation id="kspop" component="kscell_1" size="1" />
</Network>

<Simulation length="80ms" step="0.07ms" target="net1">
   <Display timeScale="ms">
      <Line quantity="kspop[0]/v" scale="mV" color="#ff4040" />
   </Display>
</Simulation>
```

When run, this produces:

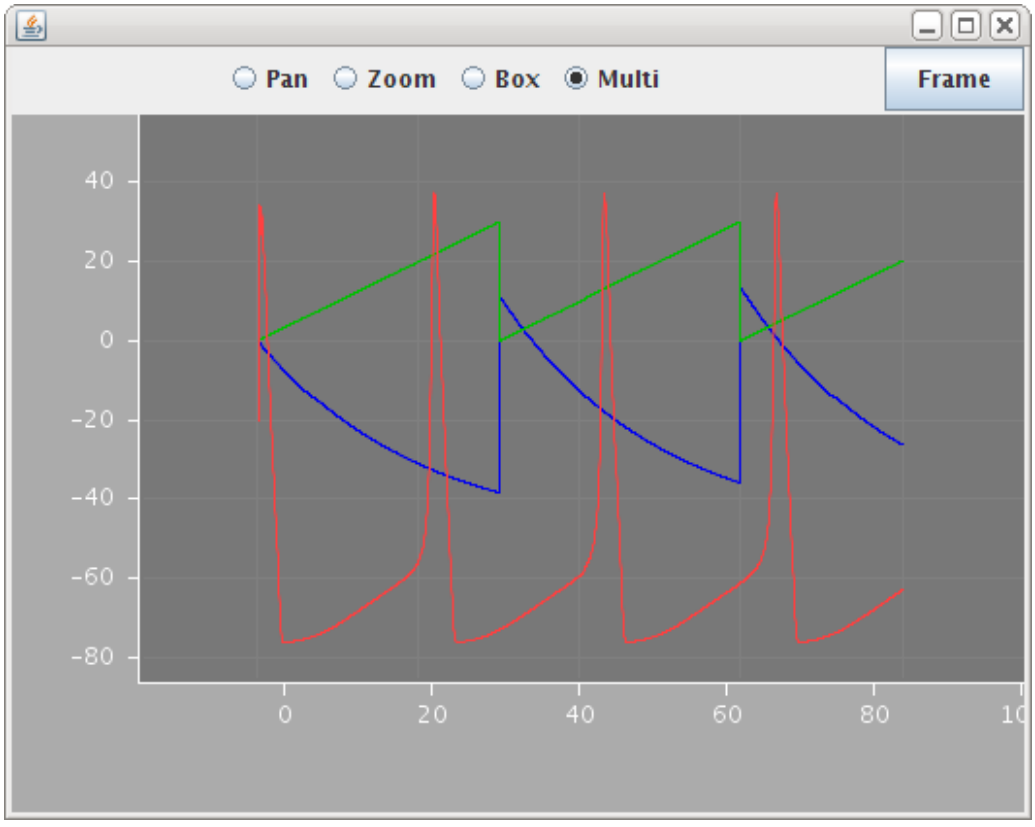


Fig. 12.2: LEMS GUI showing simulation output graphs

There are clearly some initialization issues but the basic Dynamics is the same as the PSICS version of this model.

12.11 Example 5: References and paths

The ChannelPopulation type used earlier repeats a common model specification error in that it makes the reversal potential of a population of channels a parameter of the population (often it is made a parameter of the channel specification, which is equally bad):

```
<ComponentType name="ChannelPopulation">
   <ComponentRef name="channel" type="KSChannel" />
   <Parameter name="number" dimension="none" />
   <Parameter name="erev" dimension="voltage" />
</ComponentType>
```

In fact, of course, the reversal potential is not a property of a channel population, or of a channel. It depends on the environment the channel is put in and the ions it is permeable to. But, it is needed in the Dynamics specification for the population so just putting it in as a parameter solves the immediate problem. In the process, however, it introduces the potential for easily creating contradictory models, by, for example setting different reversals for populations of the same type of channel.

A much better approach is to let the channel just say what it is permeable to. Some other element in the model can define the membrane reversal potentials for different channels, and the channel population object should then look up the relevant value for the permeant ion of its channel. This provides a cleaner expression of what is there, removes redundancy and lowers the entropy of the model specification.

The following three types are sufficient to provide a simple framework to centralize the definitions of species and reversal potentials on one place:

```
<ComponentType name="Species">
   <Text name="name" />
   <Parameter name="charge" dimension="none" />
</ComponentType>
<ComponentType name="Environment">
   <Children name="membranePotentials" type="MembranePotential" />
</ComponentType>
<ComponentType name="MembranePotential">
   <ComponentRef name="species" type="Species" />
   <Parameter name="reversal" dimension="voltage" />
</ComponentType>
```

Once these are available, they can be used to define some species, and to create an environment component that sets their reversal potentials:

```
<Species id="Na" name="Sodium" charge="1" />
<Species id="K" name="Potassium" charge="1" />
<Species id="Ca" name="Calcium" charge="1" />
<Environment id="env1">
   <MembranePotential species="Na" reversal="50mV" />
   <MembranePotential species="K" reversal="-80mV" />
</Environment>
```

The next step is to add a species reference to the channel type, so that channel definitions can say what species they are permeant to.

```
<ComponentType name="KSChannel">
   <Parameter name="conductance" dimension="conductance" />
   <ComponentRef name="species" type="Species" />
   <Children name="gates" type="KSGate" />
   <Dynamics>
```

(continues on next page)

(continued from previous page)

```
      <DerivedVariable name="fopen" dimension="none" select="gates[*]/fopen" reduce=
↪"multiply" />
      <DerivedVariable name="g" dimension="conductance" value="fopen * conductance" />
   </Dynamics>
</ComponentType>
```

Finally the channel population type needs modifying to add a derived parameter that addresses the reversal potential from the membrane properties:

```
<ComponentType name="ChannelPopulation">
   <ComponentRef name="channel" type="KSChannel" />
   <Parameter name="number" dimension="none" />
   <Requirement name="v" dimension="voltage" />
   <DerivedParameter name="erev" dimension="voltage" select="//
↪MenbranePotential[species = channel/species]/reversal" />
   <Dynamics>
      <DerivedVariable name="channelg" dimension="conductance" select="channel/g" />
      <DerivedVariable name="geff" value="channelg * number" />
      <DerivedVariable name="current" value="geff * (erev - v)" />
   </Dynamics>
</ComponentType>
```

This introduces a new construct, the DerivedParameter specification that defines a local parameter "erev" to hold the quantity from the specified path:

```
<DerivedParameter name="erev" dimension="voltage" select="//MenbranePotential[species␣
↪= channel/species]/reversal" />
```

The path here uses XPath like syntax operating on the component tree in the model. In this case, it finds all the elements of thpe MembranePotential in the model. The predicate selects the one for which the species is the same as the species referred to from the channel used for this population. Finally, it takes the "reversal" parameter from the membrane potential component. This is made locally available as the parameter "erev".

The Dynamics of this model is exactly the same as example 4. The full model including both the type definitions and the components is included below.

```
<Lems>

    <Target component="sim1"/>

    <Include file="ex2dims.xml"/>

    <ComponentType name="Species">
        <Text name="name"/>
        <Parameter name="charge" dimension="none"/>
    </ComponentType>


    <ComponentType name="Environment">
        <Children name="membranePotentials" type="MembranePotential"/>
    </ComponentType>


    <ComponentType name="MembranePotential">
        <ComponentReference name="species" type="Species"/>
        <Parameter name="reversal" dimension="voltage"/>
```

(continues on next page)

(continued from previous page)

```
    </ComponentType>

    <Species id="Na" name="Sodium" charge="1"/>
    <Species id="K" name="Potassium" charge="1"/>
    <Species id="Ca" name="Calcium" charge="1"/>


    <Environment id="env1">
        <MembranePotential species="Na" reversal="50mV"/>
        <MembranePotential species="K" reversal="-80mV"/>
    </Environment>


    <ComponentType name="KSChannel">
        <Parameter name="conductance" dimension="conductance"/>
        <ComponentReference name="species" type="Species"/>
        <Children name="gates" type="KSGate"/>
        <Exposure name="g" dimension="conductance"/>

        <Dynamics>
            <DerivedVariable name="fopen" dimension="none" select="gates[*]/fopen"␣
↪reduce="multiply"/>
            <DerivedVariable name="g" exposure="g" dimension="conductance" value=
↪"fopen * conductance"/>
        </Dynamics>
    </ComponentType>



    <ComponentType name="KSGate">
        <Parameter name="power" dimension="none"/>
        <Parameter name="deltaV" dimension="voltage"/>
        <Children name="states" type="KSState"/>
        <Children name="transitions" type="KSTransition"/>
        <Exposure name="fopen" dimension="none"/>

        <Dynamics>
            <KineticScheme name="ks" nodes="states" stateVariable="occupancy"
                           edges="transitions" edgeSource="from" edgeTarget="to"
                           forwardRate="rf" reverseRate="rr" dependency="v" step=
↪"deltaV"/>

            <DerivedVariable name="q" dimension="none" select="states[*]/q" reduce=
↪"add"/>
            <DerivedVariable name="fopen" exposure="fopen" dimension="none" value="q^
↪power"/>
        </Dynamics>
    </ComponentType>


    <ComponentType name="KSState">
        <Parameter name="relativeConductance" dimension="none"/>
        <Exposure name="q" dimension="none"/>
        <Exposure name="occupancy" dimension="none"/>

        <Dynamics>
            <StateVariable name="occupancy" exposure="occupancy" dimension="none"/>
```

(continues on next page)

(continued from previous page)

```
            <DerivedVariable name="q" exposure="q" value="relativeConductance *␣
↪occupancy"/>
        </Dynamics>
    </ComponentType>

    <ComponentType name="KSClosedState" extends="KSState">
        <Fixed parameter="relativeConductance" value="0"/>
    </ComponentType>


    <ComponentType name="KSOpenState" extends="KSState">
        <Fixed parameter="relativeConductance" value="1"/>
    </ComponentType>


    <ComponentType name="KSTransition">
        <Link name="from" type="KSState"/>
        <Link name="to" type="KSState"/>
        <Exposure name="rf" dimension="per_time"/>
        <Exposure name="rr" dimension="per_time"/>

    </ComponentType>


    <ComponentType name="VHalfTransition" extends="KSTransition">
        <Parameter name="vHalf" dimension="voltage"/>
        <Parameter name="z" dimension="none"/>
        <Parameter name="gamma" dimension="none"/>
        <Parameter name="tau" dimension="time"/>
        <Parameter name="tauMin" dimension="time"/>
        <Constant name="kte" dimension="voltage" value="25.3mV"/>
        <Requirement name="v" dimension="voltage"/>

        <Dynamics>
            <DerivedVariable name="rf0" dimension="per_time" value="exp(z * gamma *␣
↪(v - vHalf) / kte) / tau"/>
            <DerivedVariable name="rr0" dimension="per_time" value="exp(-z * (1 -␣
↪gamma) * (v - vHalf) / kte) / tau"/>
            <DerivedVariable name="rf" exposure="rf" dimension="per_time" value="1 /␣
↪(1/rf0 + tauMin)"/>
            <DerivedVariable name="rr" exposure="rr" dimension="per_time" value="1 /␣
↪(1/rr0 + tauMin)"/>
        </Dynamics>
    </ComponentType>







    <KSChannel id="na1" conductance="20pS" species="Na">
        <KSGate power="1" deltaV="0.1mV">
            <KSClosedState id="c1"/>
            <KSClosedState id="c2"/>
            <KSOpenState id="o1" relativeConductance="1"/>
            <KSClosedState id="c3"/>
            <VHalfTransition from="c1" to="c2" vHalf = "-35mV" z="2.5" gamma="0.8"␣
```

(continues on next page)

(continued from previous page)

```
↪tau="0.15ms" tauMin="0.001ms"/>
            <VHalfTransition from="c2" to="o1" vHalf = "-35mV" z="2.5" gamma="0.8"␣
↪tau="0.15ms" tauMin="0.001ms"/>
            <VHalfTransition from="o1" to="c3" vHalf = "-70mV" z="1.1" gamma="0.90"␣
↪tau="8.0ms" tauMin="0.01ms"/>
        </KSGate>
    </KSChannel>


    <KSChannel id="k1" conductance="30pS" species="K">
        <KSGate power="1" deltaV="0.1mV">
            <KSClosedState id="c1"/>
            <KSOpenState id="o1"/>
            <VHalfTransition from="c1" to="o1" vHalf = "0mV" z="1.5" gamma="0.75" tau=
↪"3.2ms" tauMin="0.3ms"/>
        </KSGate>
    </KSChannel>




    <ComponentType name="ChannelPopulation">
        <ComponentReference name="channel" type="KSChannel"/>
        <Parameter name="number" dimension="none"/>
        <Requirement name="v" dimension="voltage"/>
        <Exposure name="current" dimension="current"/>
        <DerivedParameter name="erev" dimension="voltage" select="//
↪MembranePotential[species=channel/species]/reversal"/>
        <Dynamics>

            <DerivedVariable name="channelg" dimension="conductance" select="channel/g
↪"/>
            <DerivedVariable name="geff" value="channelg * number"/>
            <DerivedVariable name="current" exposure="current" value="geff * (erev -␣
↪v)"/>
        </Dynamics>


        <Structure>
            <ChildInstance component="channel"/>
        </Structure>
    </ComponentType>




    <ComponentType name="KSCell">
        <Parameter name="capacitance" dimension="capacitance"/>
        <ComponentReference name="environment" type="Environment"/>
        <Children name="populations" type="ChannelPopulation"/>
        <Parameter name="injection" dimension="current"/>
        <Parameter name="v0" dimension="voltage"/>
        <Exposure name="v" dimension="voltage"/>
        <Dynamics>
            <OnStart>
                <StateAssignment variable="v" value="v0"/>
            </OnStart>
```

(continues on next page)

(continued from previous page)

```
            <DerivedVariable name="totcurrent" dimension="current" select=
↪"populations[*]/current" reduce="add"/>
            <StateVariable name="v" exposure="v" dimension="voltage"/>
            <TimeDerivative variable="v" value="(totcurrent + injection) / capacitance
↪"/>
        </Dynamics>
    </ComponentType>



    <KSCell id="kscell_1" capacitance="0.4pF" injection="1pA" v0="-60mV" environment=
↪"env1">
        <ChannelPopulation channel="na1" number="400"/>
        <ChannelPopulation channel="k1" number="180"/>
    </KSCell>


    <Include file="SingleSimulation.xml" />

    <Simulation id="sim1" length="80ms" step="0.05ms" target="kscell_1">
        <Display id="d0" title="Example 5: References and paths" timeScale="1ms" xmin=
↪"-10" xmax="90" ymin="-90" ymax="60">
            <Line id="v" quantity="v" scale="1mV" timeScale="1ms" color="#0000f0"/>
        </Display>
    </Simulation>

</Lems>
```

12.12 Example 6: User defined types for simulation and display

Up until now, the examples have used a set of simple Simulation, Display and Line constructs without explaining how they are defined. This shows what is needed in the Dynamics block to let the user defined types to specify that they actually define a runnable simulation or settings that can be used to display results.

This means that the user can select their own names for the different parameters required for a simulation, and, more importantly, simulation and display attributes can be added to existing type definitions to make multi-faceted type definitions that can both be run on their own or as part of a larger simulation.

Example 6 shows two new elements that can be used in the Dynamics block, Run and Show as illustrated in the following user-defined type that defines a simulation:

```
<ComponentType name="Simulation">
   <Parameter name="length" dimension="time" />
   <Parameter name="step" dimension="time" />
   <ComponentRef name="target" type="HHCell" />
   <Children name="displays" type="Display" />
   <Dynamics>
      <StateVariable name="t" dimension="time" />
      <Run component="target" variable="t" increment="step" total="length" />
      <Show src="displays" />
   </Dynamics>
</ComponentType>
```

The 'component' attribute of the Run element specifies which parameter of the type contains the reference to the com-

ponent that should actually be run. The 'step' and 'increment' attributes specify the parameters that hold the timestep and total runtime. The 'variable' attribute is for future use - at present, the independent variable is always 't'.

A Run element can be added to the Dynamics block in any type definition to make it independently runnable.

Running a simulation without any output is rarely much use, so there are two futher elements that can be included in the Dynamics block: Show and Record. The 'src' attribute of the Show element points to the components that should be shown. These in turn can contain other Show elements but eventually everything pointed to by a Show element should contain one or more Record elements. These specify what will actually be sent as output. They have the path to the variable as the 'quantity' attribute, its scale as the 'scale' attribute and the line color for plotting.

The following two types show one way that these can be combined to allow the user to express a display object containing one or more lines.

```
<ComponentType name="Display">
   <Parameter name="timeScale" dimension="time" />
   <Children name="lines" type="Line" />
   <Dynamics>
      <Show src="lines" scale="timeScale" />
   </Dynamics>
</ComponentType>
<ComponentType name="Line">
   <Parameter name="scale" dimension="*" />
   <Text name="color" />
   <Path name="quantity" />
   <Dynamics>
      <Record quantity="quantity" scale="scale" color="color" />
   </Dynamics>
</ComponentType>
```

Once these have been defined, a component can be constructed that uses them as follows:

```
<Simulation id="sim1" length="80ms" step="0.05ms" target="hhcell_1">
   <Display timeScale="1ms">
      <Line id="V" quantity="v" scale="1mV" color="#0000f0" />
      <Line id="Na_q" quantity="NaPop/geff" scale="1nS" color="#f00000" />
      <Line id="K_q" quantity="KPop/geff" scale="1nS" color="#00f000" />
   </Display>
</Simulation>
```

When run, this produces the output shown below:

Note how the scale attributes are set to 1mV and 1nS for the different lines so that they show up on the same axes.

12.13 Example 7: User defined types for networks and populations

This example shows how the standard component type structures can be used to declare components for simple networks. The following three definitions allow networks to be constructed containing fixed size populations of a particular component type.

```
<ComponentType name="Network">
   <Children name="populations" type="Population" />
   <Children name="connectivities" type="EventConnectivity" />
</ComponentType>
<ComponentType name="Population">
   <ComponentRef name="component" type="Component" />
```

(continues on next page)

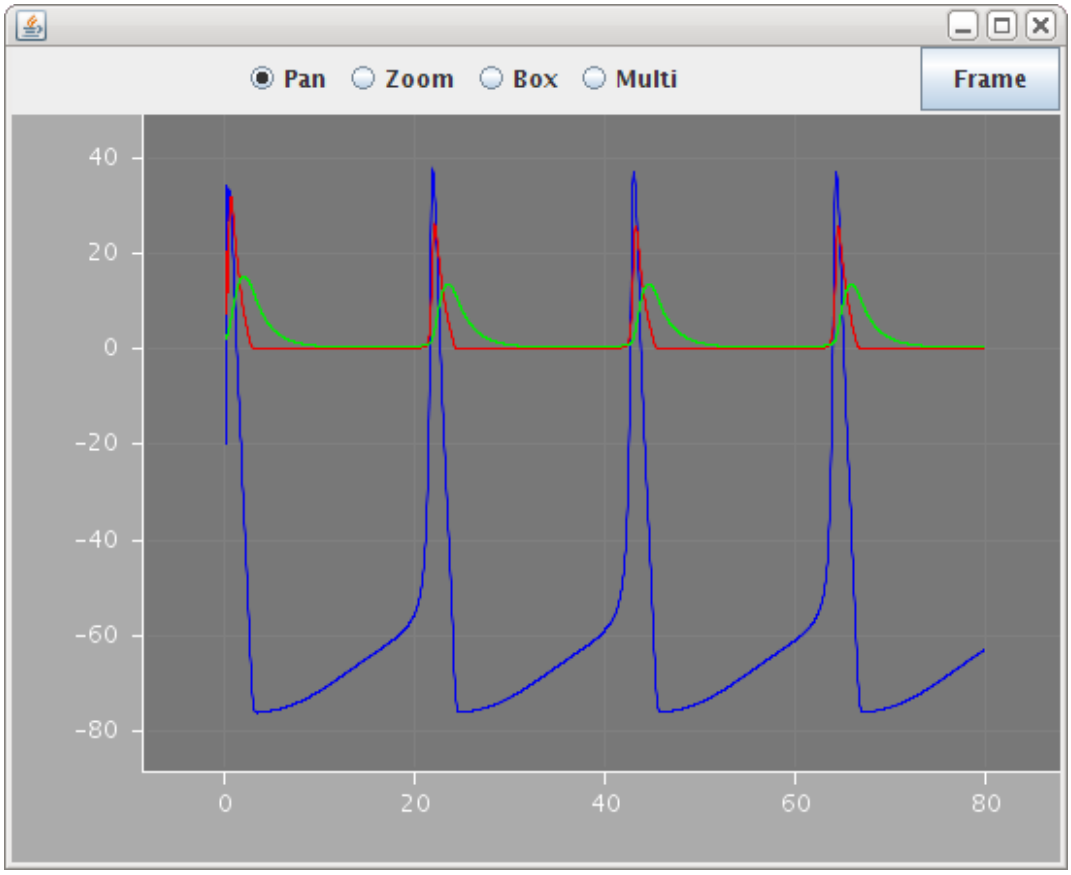


Fig. 12.3: LEMS GUI showing simulation output graphs

(continued from previous page)

```
   <Parameter name="size" dimension="none" />
</ComponentType>
<ComponentType name="EventConnectivity">
   <Link name="source" type="Population" />
   <Link name="target" type="Population" />
   <Child name="Connections" type="ConnectionPattern" />
</ComponentType>
```

The harder part is to provide elements in the Dynamics blocks to express what should be done with components based on these types. The Network element doesn't pose any problems because the default behavior on instantiation will do the right thing: it will instantiate each of the child populations and EventConnectivity elements.

But the population element needs to say that its instantiation involves making 'size' instances of the component referred to by the 'component' reference, where 'size' is the value supplied for the size parameter in a component specification. This can be done by including a Build element inside the Dynamics block:

```
<ComponentType name="Population">
   <ComponentRef name="component" type="Component" />
   <Parameter name="size" dimension="none" />
   <Dynamics>
      <Build>
         <MultiInstantiate number="size" component="component" />
      </Build>
   </Dynamics>
</ComponentType>
```

The MultiInstantiate specification says that there should be 'size' instances of the component referred to in the 'component' parameter created when the model is built. This overrides the default behavior. [TODO: what is the Build element content corresponding to the default behavior?].

This serves to create some rather simple populations. More complex specifications, such as putting one instance at each point of a grid satisfying a particular constraint could be handled via first declaring elements to form the grid, and then using selectors that pick the points in the population element to actually put the cells at [its not clear to me how much more would be required to make this work, other than implementing proper xpath-like selectors].

The following three types define a general connectivity structure with an abstract ConnectionPattern type, and a specific instance for All-All connectivity.

```
<ComponentType name="EventConnectivity">
   <Link name="source" type="Population" />
   <Link name="target" type="Population" />
   <Child name="Connections" type="ConnectionPattern" />
</ComponentType>
<ComponentType name="ConnectionPattern">
</ComponentType>
<ComponentType name="AllAll" extends="ConnectionPattern">
   <Dynamics>
      <Build>
         <ForEach instances="../source" as="a">
            <ForEach instances="../target" as="b">
               <EventConnection from="a" to="b" />
            </ForEach>
         </ForEach>
      </Build>
   </Dynamics>
</ComponentType>
```

The Build element in the AllAll pattern uses a new ForEach construct and the EventConnectin element from before. The ForEach element operates selects each instance matching its 'instances' attribute, and applies the enclosing directives, much in the same way as for-each in XSL. The proof of concept interpreter also has Choose, When and Otherwise elements that operate much like their XSL equivalents, although these are not used in this example.

With these definitions in place, a network simulation can be defined with the following:

```
<Network id="net1">
   <Population id="p1" component="gen1" size="2" />
   <Population id="p3" component="iaf3cpt" size="3" />
   <EventConnectivity id="p1-p3" source="p1" target="p3">
      <Connections type="AllAll" />
   </EventConnectivity>
</Network>

<Simulation id="sim1" length="80ms" step="0.05ms" target="net1">
   <Display timeScale="1ms">
      <Line id="gen_v" quantity="p3[0]/v" scale="1mV" color="#0000f0" />
      <Line id="gen_tsince" quantity="p1[0]/tsince" scale="1ms" color="#00c000" />
   </Display>
</Simulation>
```

The output when the model is run is shown below, followed by the full listing.

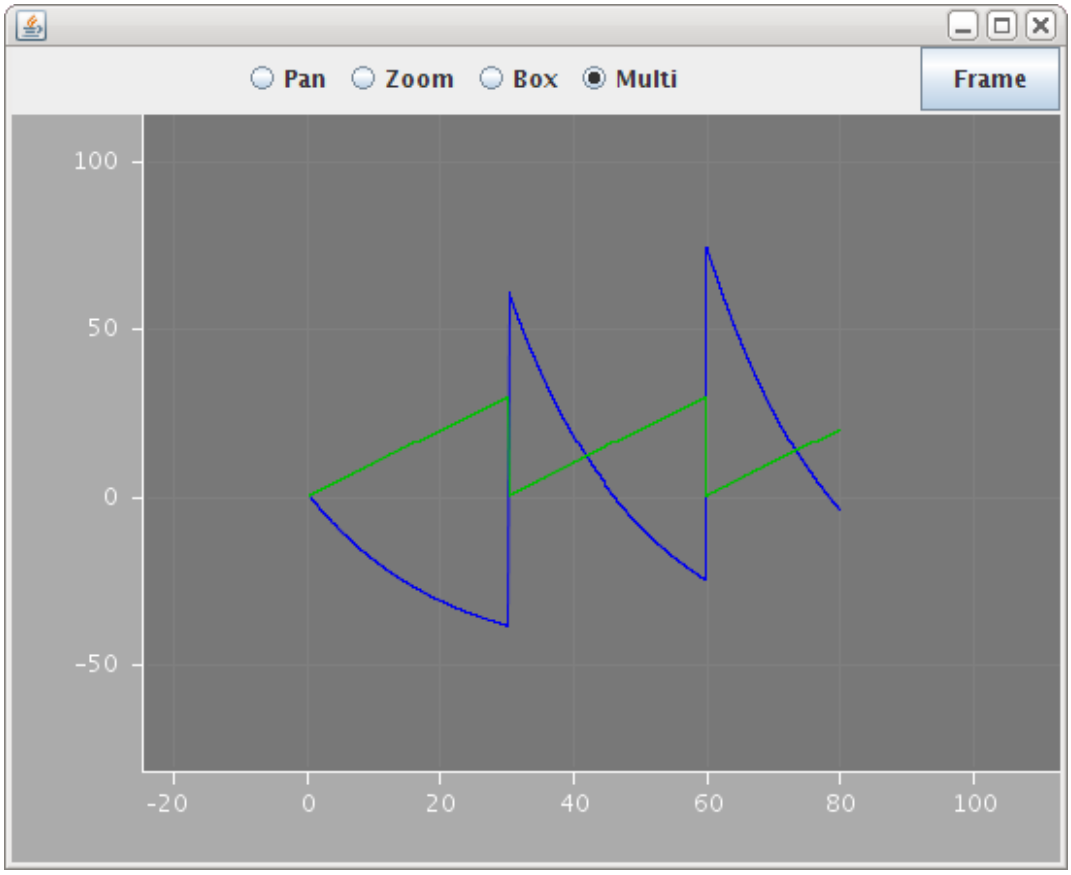


Fig. 12.4: LEMS GUI showing simulation output graphs

```
<Lems>

    <Target component="sim1"/>


    <Include file="ex2dims.xml"/>
    <Include file="spikegenerators.xml"/>
    <Include file="misciaf.xml"/>


    <Component id="gen1" type="spikeGenerator" period="30ms"/>

    <Component id="gen2" type="spikeGenerator2" period="32ms"/>

    <Component id="iaf3cpt" type="iaf3" leakReversal="-50mV" deltaV="50mV" threshold=
↪"-30mV" leakConductance="50pS"
            refractoryPeriod="4ms" capacitance="1pF"/>


    <ComponentType name="Network">
        <Children name="populations" type="Population"/>
        <Children name="connectivities" type="EventConnectivity"/>
    </ComponentType>


    <ComponentType name="Population">
        <ComponentReference name="component" type="Component"/>
        <Parameter name="size" dimension="none"/>

        <Structure>
            <MultiInstantiate number="size" component="component"/>
        </Structure>

    </ComponentType>


    <ComponentType name="EventConnectivity">
        <Link name="source" type="Population"/>
        <Link name="target" type="Population"/>
        <Child name="Connections" type="ConnectionPattern"/>
    </ComponentType>


    <ComponentType name="ConnectionPattern"/>


    <ComponentType name="AllAll" extends="ConnectionPattern">
        <Structure>
            <ForEach instances="../source" as="a">
                <ForEach instances="../target" as="b">
                    <EventConnection from="a" to="b"/>
                </ForEach>
            </ForEach>
        </Structure>
    </ComponentType>
```

(continues on next page)

(continued from previous page)

```
    <Network id="net1">
        <Population id="p1" component="gen1" size="2"/>
        <Population id="p3" component="iaf3cpt" size="3"/>

        <EventConnectivity id="p1-p3" source="p1" target="p3">
            <Connections type="AllAll"/>
        </EventConnectivity>
    </Network>


    <Include file="SingleSimulation.xml" />

    <Simulation id="sim1" length="80ms" step="0.05ms" target="net1">
        <Display id="d0" title="Example 7: User defined types for networks and␣
↪populations" timeScale="1ms" xmin="-10" xmax="90" ymin="-50" ymax="90">
            <Line id="gen_v" quantity="p3[0]/v" scale="1mV" timeScale="1ms" color="
↪#0000f0"/>
            <Line id="gen_tsince" quantity="p1[0]/tsince" scale="1ms" timeScale="1ms"␣
↪color="#00c000"/>
        </Display>
    </Simulation>

</Lems>
```

12.14 Example 8: Regimes in Dynamics definitions

This example introduces the Regime, Transition and OnEntry elements within a Dynamics block. Rather than having a single state instance, the entity can be on one of the defined regimes at any given time. The Transition element occurring inside a condition block serves to move it from one regime to another. The OnEntry block inside a regime can contain initialization directives that apply each time the entity enters that regime.

```
<ComponentType name="refractiaf">
   <Parameter name="threshold" dimension="voltage" />
   <Parameter name="refractoryPeriod" dimension="time" />
   <Parameter name="capacitance" dimension="capacitance" />
   <Parameter name="vleak" dimension="voltage" />
   <Parameter name="gleak" dimension="conductance" />
   <Parameter name="current" dimension="current" />
   <Parameter name="vreset" dimension="voltage" />
   <Parameter name="deltaV" dimension="voltage" />
   <Parameter name="v0" dimension="voltage" />
   <EventPort name="out" direction="out" />
   <EventPort name="in" direction="in" />
   <Dynamics>
      <StateVariable name="v" dimension="voltage" />
      <OnStart>
         <StateAssignment variable="v" value="v0" />
      </OnStart>
      <Regime name="refr">
         <StateVariable name="tin" dimension="time" />
         <OnEntry>
            <StateAssignment variable="tin" value="t" />
            <StateAssignment variable="v" value="vreset" />
         </OnEntry>
```

(continues on next page)

(continued from previous page)

```
        <OnCondition test="t .gt. tin + refractoryPeriod">
           <Transition regime="int" />
        </OnCondition>
     </Regime>
     <Regime name="int" initial="true">
        <TimeDerivative variable="v" value="(current + gleak * (vleak - v)) /␣
↪capacitance" />
        <OnCondition test="v .gt. threshold">
           <EventOut port="out" />
           <Transition regime="refr" />
        </OnCondition>
        <OnEvent port="in">
           <StateAssignment variable="v" value="v + deltaV" />
        </OnEvent>
     </Regime>
  </Dynamics>
</ComponentType>
```

Full listing:

```
<Lems>

    <Target component="sim1"/>


    <Include file="ex2dims.xml"/>
    <Include file="spikegenerators.xml"/>
    <Include file="misciaf.xml"/>


    <ComponentType name="refractiaf">
        <Parameter name="threshold" dimension="voltage"/>
        <Parameter name="refractoryPeriod" dimension="time"/>
        <Parameter name="capacitance" dimension="capacitance"/>
        <Parameter name="vleak" dimension="voltage"/>
        <Parameter name="gleak" dimension="conductance"/>

        <Parameter name="current" dimension="current"/>
        <Parameter name="vreset" dimension="voltage"/>
        <Parameter name="deltaV" dimension="voltage"/>
        <Parameter name="v0" dimension="voltage"/>

        <EventPort name="out" direction="out"/>
        <EventPort name="in" direction="in"/>

        <Exposure name="v" dimension="voltage"/>

        <Dynamics>
            <StateVariable name="v" exposure="v" dimension="voltage" />
            <StateVariable name="tin" dimension="time"/>

            <OnStart>
                <StateAssignment variable="v" value="v0"/>
            </OnStart>

            <Regime name="refr">
                <OnEntry>
```

(continues on next page)

(continued from previous page)

```
                    <StateAssignment variable="tin" value="t" />
                    <StateAssignment variable="v" value="vreset" />
                </OnEntry>
                <OnCondition test="t .gt. tin + refractoryPeriod">
                    <Transition regime="int" />
                </OnCondition>
            </Regime>

            <Regime name="int" initial="true">
                <TimeDerivative variable="v" value="(current + gleak * (vleak - v)) /␣
↪capacitance" />
                <OnCondition test="v .gt. threshold">
                    <EventOut port="out" />
                    <Transition regime="refr" />
                </OnCondition>
                <OnEvent port="in">
                    <StateAssignment variable="v" value="v + deltaV"/>
                </OnEvent>

            </Regime>
        </Dynamics>

    </ComponentType>


    <Component id="gen1" type="spikeGenerator" period="7ms"/>


    <Component id="multiregime" type="refractiaf" threshold="-50mV" v0="-80mV"
               refractoryPeriod="20ms" capacitance="1pF" vreset="-80mV" vleak="-90mV"
               gleak="5pS" current="0.00001nA" deltaV="5mV"/>


    <ComponentType name="Network">
        <Children name="populations" type="Population"/>
        <Children name="connectivities" type="EventConnectivity"/>
    </ComponentType>


    <ComponentType name="Population">
        <ComponentReference name="component" type="Component"/>
        <Parameter name="size" dimension="none"/>
        <Structure>
                <MultiInstantiate number="size" component="component"/>
        </Structure>
    </ComponentType>


    <ComponentType name="EventConnectivity">
        <Link name="source" type="Population"/>
        <Link name="target" type="Population"/>
        <Child name="Connections" type="ConnectionPattern"/>
    </ComponentType>


    <ComponentType name="ConnectionPattern"/>
```

(continues on next page)

(continued from previous page)

```
    <ComponentType name="AllAll" extends="ConnectionPattern">
        <Structure>
            <ForEach instances="../source" as="a">
                <ForEach instances="../target" as="b">
                    <EventConnection from="a" to="b"/>
                </ForEach>
            </ForEach>
        </Structure>
    </ComponentType>


    <Network id="net1">
        <Population id="p1" component="gen1" size="1"/>
        <Population id="p3" component="multiregime" size="2"/>

        <EventConnectivity id="p1-p3" source="p1" target="p3">
            <Connections type="AllAll"/>
        </EventConnectivity>
    </Network>


    <Include file="SingleSimulation.xml" />

    <Simulation id="sim1" length="80ms" step="0.05ms" target="net1">
        <Display id="d0" title="Example 8: Regimes in dynamics definitions" timeScale=
↪"1ms" xmin="-10" xmax="90" ymin="-90" ymax="20">
            <Line id="gen_vmr" quantity="p3[0]/v" scale="1mV" timeScale="1ms" color="
↪#00c000"/>
            <Line id="gen_sv" quantity="p1[0]/tsince" scale="1ms" timeScale="1ms"␣
↪color="#f00000"/>
        </Display>
    </Simulation>


</Lems>
```

CHAPTER

THIRTEEN

SCHEMA/SPECIFICATION

NeuroML v2.3 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.

13.1 NeuroML v2

The current stable version of NeuroML is v2.3, and the XSD Schema for this can be found here. The following figure, taken from Cannon et al. 2014 ([CGC+14]) shows some of the core elements defined in NeuroML version 2 (note: these key elements haven't changed since that publication).

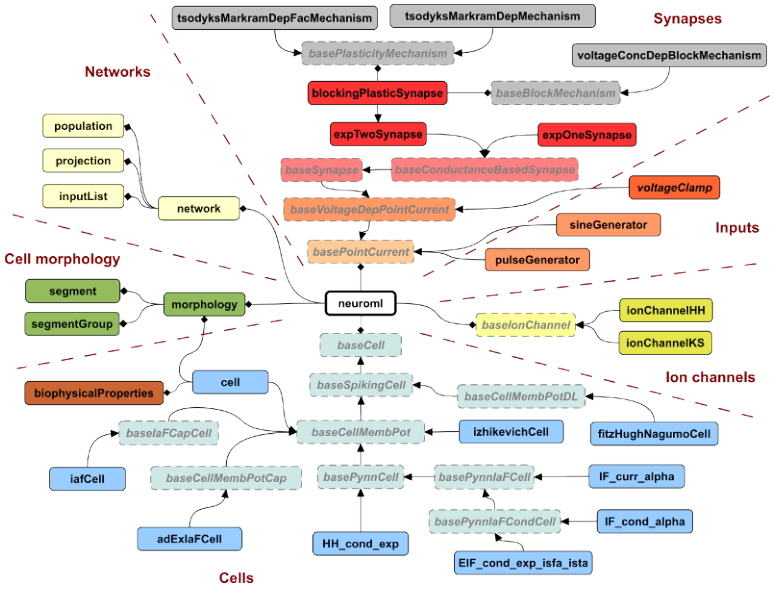


Fig. 13.1: Elements defined in the NeuroML schema, version 2.

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>*, **, *<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>*, *<izhikevich2007Cell>*, *<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).

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*.

13.1.1 NeuroMLDocument

The main NeuroML container class, and other associated types that do not fit into the other categories

---

Schema against which NeuroML based on these should be valid: NeuroML_v2.3.xsd. Please file any issues or questions at the issue tracker here.

NeuroMLDocument

Schema

```
  <xs:element name="neuroml" type="NeuroMLDocument">
    <xs:annotation>
      <xs:documentation>The root NeuroML element.</xs:documentation>
    </xs:annotation>
  </xs:element>

  <xs:complexType name="NeuroMLDocument">
    <xs:complexContent>
      <xs:extension base="Standalone">
        <xs:sequence>
          <xs:element name="include" type="IncludeType" minOccurs="0" maxOccurs=
↪"unbounded"/>
          <xs:element name="extracellularProperties" type="ExtracellularProperties"␣
↪minOccurs="0" maxOccurs="unbounded"/>
          <xs:element name="intracellularProperties" type="IntracellularProperties"␣
↪minOccurs="0" maxOccurs="unbounded"/>
          <xs:element name="morphology" type="Morphology" minOccurs="0" maxOccurs=
↪"unbounded"/>
          <xs:element name="ionChannel" type="IonChannel" minOccurs="0" maxOccurs=
↪"unbounded"/>
          <xs:element name="ionChannelHH" type="IonChannelHH" minOccurs="0" maxOccurs=
↪"unbounded"/>
          <xs:element name="ionChannelVShift" type="IonChannelVShift" minOccurs="0"␣
↪maxOccurs="unbounded"/>
          <xs:element name="ionChannelKS" type="IonChannelKS" minOccurs="0" maxOccurs=
↪"unbounded"/>
          <xs:group ref="ConcentrationModelTypes"/>
          <xs:group ref="SynapseTypes"/>
          <xs:element name="biophysicalProperties" type="BiophysicalProperties"␣
↪minOccurs="0" maxOccurs="unbounded"/>
          <xs:group ref="CellTypes"/>
          <xs:group ref="InputTypes"/>
          <xs:group ref="PyNNCellTypes"/>
          <xs:group ref="PyNNSynapseTypes"/>
          <xs:group ref="PyNNInputTypes"/>
          <xs:element name="network" type="Network" minOccurs="0" maxOccurs="unbounded
↪"/>
```

(continues on next page)

(continued from previous page)

```
        <xs:element name="ComponentType" type="ComponentType" minOccurs="0"␣
↪maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
 </xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import NeuroMLDocument
from neuroml.utils import component_factory

variable = component_factory(
    NeuroMLDocument,
    id: 'a NmlId (required)' = None
    metaid: 'a MetaId (optional)' = None
    notes: 'a string (optional)' = None
    properties: 'list of Property(s) (optional)' = None
    annotation: 'a Annotation (optional)' = None
    includes: 'list of IncludeType(s) (optional)' = None
    extracellular_properties: 'list of ExtracellularProperties(s) (optional)' = None
    intracellular_properties: 'list of IntracellularProperties(s) (optional)' = None
    morphology: 'list of Morphology(s) (optional)' = None
    ion_channel: 'list of IonChannel(s) (optional)' = None
    ion_channel_hhs: 'list of IonChannelHH(s) (optional)' = None
    ion_channel_v_shifts: 'list of IonChannelVShift(s) (optional)' = None
    ion_channel_kses: 'list of IonChannelKS(s) (optional)' = None
    decaying_pool_concentration_models: 'list of DecayingPoolConcentrationModel(s)␣
↪(optional)' = None
    fixed_factor_concentration_models: 'list of FixedFactorConcentrationModel(s)␣
↪(optional)' = None
    alpha_current_synapses: 'list of AlphaCurrentSynapse(s) (optional)' = None
    alpha_synapses: 'list of AlphaSynapse(s) (optional)' = None
    exp_one_synapses: 'list of ExpOneSynapse(s) (optional)' = None
    exp_two_synapses: 'list of ExpTwoSynapse(s) (optional)' = None
    exp_three_synapses: 'list of ExpThreeSynapse(s) (optional)' = None
    blocking_plastic_synapses: 'list of BlockingPlasticSynapse(s) (optional)' = None
    double_synapses: 'list of DoubleSynapse(s) (optional)' = None
    gap_junctions: 'list of GapJunction(s) (optional)' = None
    silent_synapses: 'list of SilentSynapse(s) (optional)' = None
    linear_graded_synapses: 'list of LinearGradedSynapse(s) (optional)' = None
    graded_synapses: 'list of GradedSynapse(s) (optional)' = None
    biophysical_properties: 'list of BiophysicalProperties(s) (optional)' = None
    cells: 'list of Cell(s) (optional)' = None
    cell2_ca_poolses: 'list of Cell2CaPools(s) (optional)' = None
    base_cells: 'list of BaseCell(s) (optional)' = None
    iaf_tau_cells: 'list of IafTauCell(s) (optional)' = None
    iaf_tau_ref_cells: 'list of IafTauRefCell(s) (optional)' = None
    iaf_cells: 'list of IafCell(s) (optional)' = None
    iaf_ref_cells: 'list of IafRefCell(s) (optional)' = None
    izhikevich_cells: 'list of IzhikevichCell(s) (optional)' = None
    izhikevich2007_cells: 'list of Izhikevich2007Cell(s) (optional)' = None
    ad_ex_ia_f_cells: 'list of AdExIaFCell(s) (optional)' = None
```

(continues on next page)

(continued from previous page)

```
    fitz_hugh_nagumo_cells: 'list of FitzHughNagumoCell(s) (optional)' = None
    fitz_hugh_nagumo1969_cells: 'list of FitzHughNagumo1969Cell(s) (optional)' = None
    pinsky_rinzel_ca3_cells: 'list of PinskyRinzelCA3Cell(s) (optional)' = None
    hindmarshRose1984Cell: 'list of HindmarshRose1984Cell(s) (optional)' = None
    pulse_generators: 'list of PulseGenerator(s) (optional)' = None
    pulse_generator_dls: 'list of PulseGeneratorDL(s) (optional)' = None
    sine_generators: 'list of SineGenerator(s) (optional)' = None
    sine_generator_dls: 'list of SineGeneratorDL(s) (optional)' = None
    ramp_generators: 'list of RampGenerator(s) (optional)' = None
    ramp_generator_dls: 'list of RampGeneratorDL(s) (optional)' = None
    compound_inputs: 'list of CompoundInput(s) (optional)' = None
    compound_input_dls: 'list of CompoundInputDL(s) (optional)' = None
    voltage_clamps: 'list of VoltageClamp(s) (optional)' = None
    voltage_clamp_triples: 'list of VoltageClampTriple(s) (optional)' = None
    spike_arrays: 'list of SpikeArray(s) (optional)' = None
    timed_synaptic_inputs: 'list of TimedSynapticInput(s) (optional)' = None
    spike_generators: 'list of SpikeGenerator(s) (optional)' = None
    spike_generator_randoms: 'list of SpikeGeneratorRandom(s) (optional)' = None
    spike_generator_poissons: 'list of SpikeGeneratorPoisson(s) (optional)' = None
    spike_generator_ref_poissons: 'list of SpikeGeneratorRefPoisson(s) (optional)' =␣
↪None
    poisson_firing_synapses: 'list of PoissonFiringSynapse(s) (optional)' = None
    transient_poisson_firing_synapses: 'list of TransientPoissonFiringSynapse(s)␣
↪(optional)' = None
    IF_curr_alpha: 'list of IF_curr_alpha(s) (optional)' = None
    IF_curr_exp: 'list of IF_curr_exp(s) (optional)' = None
    IF_cond_alpha: 'list of IF_cond_alpha(s) (optional)' = None
    IF_cond_exp: 'list of IF_cond_exp(s) (optional)' = None
    EIF_cond_exp_isfa_ista: 'list of EIF_cond_exp_isfa_ista(s) (optional)' = None
    EIF_cond_alpha_isfa_ista: 'list of EIF_cond_alpha_isfa_ista(s) (optional)' = None
    HH_cond_exp: 'list of HH_cond_exp(s) (optional)' = None
    exp_cond_synapses: 'list of ExpCondSynapse(s) (optional)' = None
    alpha_cond_synapses: 'list of AlphaCondSynapse(s) (optional)' = None
    exp_curr_synapses: 'list of ExpCurrSynapse(s) (optional)' = None
    alpha_curr_synapses: 'list of AlphaCurrSynapse(s) (optional)' = None
    SpikeSourcePoisson: 'list of SpikeSourcePoisson(s) (optional)' = None
    networks: 'list of Network(s) (optional)' = None
    ComponentType: 'list of ComponentType(s) (optional)' = None
```

Usage: XML

```
<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="HL23PYR">
    <include href="HL23PYR.cell.nml"/>
    <pulseGenerator id="pg_HL23PYR" delay="50ms" duration="200ms" amplitude="0.2nA">
        <notes>Simple pulse generator</notes>
    </pulseGenerator>
    <network id="HL23PYRNet" type="networkWithTemperature" temperature="34 degC">
        <population id="HL23PYR_pop" component="HL23PYR" type="populationList">
            <property tag="color" value="0.3220229197377431 0.19279726280626452 0.
↪37635392246534727"/>
            <property tag="region" value="L23"/>
            <instance id="0">
```

(continues on next page)

(continued from previous page)

```
                <location x="0.0" y="0.0" z="0.0"/>
            </instance>
        </population>
        <inputList id="stim_iclamp_HL23PYR" population="HL23PYR_pop" component="pg_
↪HL23PYR">
            <input id="0" target="../HL23PYR_pop/0" destination="synapses"/>
        </inputList>
    </network>
</neuroml>
```

IncludeType

Used to include other documents into each other.

Schema

```
  <xs:complexType name="IncludeType">
    <xs:attribute name="href" use="required" type="xs:anyURI"/>
  </xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IncludeType
from neuroml.utils import component_factory

variable = component_factory(
    IncludeType,
    href: 'a anyURI (required)' = None)
```

Usage: XML

```
<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="HL23PYR">
    <include href="A.cell.nml"/>
    ..
</neuroml>
```

13.1.2 NeuroMLCoreDimensions

Original ComponentType definitions: NeuroMLCoreDimensions.xml. Schema against which NeuroML based on these should be valid: NeuroML_v2.3.xsd. Generated on 14/08/24 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*

Schema

```
<xs:simpleType name="Nml2Quantity_capacitance">
  <xs:restriction base="xs:string">
    <xs:pattern value="-?([0-9]*(\.[0-9]+)?)([eE]-?[0-9]+)?[\s]*(F|uF|nF|pF)"/>
  </xs:restriction>
</xs:simpleType>
```

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*
- Defined unit: *pC_per_umol*

concentration

Dimensions $L^{-3}\ N^{1}$

Units

- Defined unit: *M*
- Defined unit: *mM*
- Defined unit: *mol_per_cm3*
- Defined unit: *mol_per_m3*

Schema

```
<xs:simpleType name="Nml2Quantity_concentration">
  <xs:restriction base="xs:string">
    <xs:pattern value="-?([0-9]*(\.[0-9]+)?)([eE]-?[0-9]+)?[\s]*(mol_per_m3|mol_per_
↪cm3|M|mM)"/>
  </xs:restriction>
</xs:simpleType>
```

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*

Schema

```
<xs:simpleType name="Nml2Quantity_conductance">
  <xs:restriction base="xs:string">
    <xs:pattern value="-?([0-9]*(\.[0-9]+)?)([eE]-?[0-9]+)?[\s]*(S|mS|uS|nS|pS)"/>
  </xs:restriction>
</xs:simpleType>
```

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*
- Defined unit: *uS_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*

Schema

```
<xs:simpleType name="Nml2Quantity_conductancePerVoltage">
  <xs:restriction base="xs:string">
    <xs:pattern value="-?([0-9]*(\.[0-9]+)?)([eE]-?[0-9]+)?[\s]*(S_per_V|nS_per_mV)"/>
  </xs:restriction>
</xs:simpleType>
```

current

Dimensions I^{1}

Units

- Defined unit: *A*
- Defined unit: *nA*
- Defined unit: *pA*
- Defined unit: *uA*

Schema

```
<xs:simpleType name="Nml2Quantity_current">
  <xs:restriction base="xs:string">
    <xs:pattern value="-?([0-9]*(\.[0-9]+)?)([eE]-?[0-9]+)?[\s]*(A|uA|nA|pA)"/>
  </xs:restriction>
</xs:simpleType>
```

currentDensity

Dimensions L^{-2} I^{1}

Units

- Defined unit: *A_per_m2*
- Defined unit: *mA_per_cm2*
- Defined unit: *uA_per_cm2*

idealGasConstantDims

Dimensions M^{1} L^{2} T^{-2} K^{-1} N^{-1}

Units

- Defined unit: *J_per_K_per_mol*
- Defined unit: *fJ_per_K_per_umol*

length

Dimensions L^{1}

Units

- Defined unit: *cm*
- Defined unit: *m*
- Defined unit: *um*

Schema

```
<xs:simpleType name="Nml2Quantity_length">
  <xs:restriction base="xs:string">
    <xs:pattern value="-?([0-9]*(\.[0-9]+)?)([eE]-?[0-9]+)?[\s]*(m|cm|um)"/>
  </xs:restriction>
</xs:simpleType>
```

per_time

Dimensions T^{-1}

Units

- Defined unit: *Hz*
- Defined unit: *per_hour*
- Defined unit: *per_min*
- Defined unit: *per_ms*
- Defined unit: *per_s*

Schema

```
<xs:simpleType name="Nml2Quantity_pertime">
  <xs:restriction base="xs:string">
    <xs:pattern value="-?([0-9]*(\.[0-9]+)?)([eE]-?[0-9]+)?[\s]*(per_s|per_ms|Hz)"/>
  </xs:restriction>
</xs:simpleType>
```

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*

Schema

```
<xs:simpleType name="Nml2Quantity_permeability">
  <xs:restriction base="xs:string">
    <xs:pattern value="-?([0-9]*(\.[0-9]+)?)([eE]-?[0-9]+)?[\s]*(m_per_s|um_per_ms|cm_
↪per_s|cm_per_ms)"/>
  </xs:restriction>
</xs:simpleType>
```

resistance

Dimensions M^{1} L^{2} T^{-3} I^{-2}

Units

- Defined unit: *Mohm*
- Defined unit: *kohm*
- Defined unit: *ohm*

Schema

```
<xs:simpleType name="Nml2Quantity_resistance">
  <xs:restriction base="xs:string">
    <xs:pattern value="-?([0-9]*(\.[0-9]+)?)([eE]-?[0-9]+)?[\s]*(ohm|kohm|Mohm)"/>
  </xs:restriction>
</xs:simpleType>
```

resistivity

Dimensions M^{2} L^{2} T^{-3} I^{-2}

Units

- Defined unit: *kohm_cm*
- Defined unit: *ohm_cm*
- Defined unit: *ohm_m*

Schema

```
<xs:complexType name="Resistivity">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="value" type="Nml2Quantity_resistivity" use="required"/>
      <xs:attribute name="segmentGroup" type="NmlId" use="optional" default="all"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

rho_factor

Dimensions L^{-1} T^{-1} I^{-1} N^{1}

Units

- Defined unit: *mol_per_cm_per_uA_per_ms*
- Defined unit: *mol_per_m_per_A_per_s*
- Defined unit: *umol_per_cm_per_nA_per_ms*

Schema

```
<xs:simpleType name="Nml2Quantity_rhoFactor">
  <xs:restriction base="xs:string">
    <xs:pattern value="-?([0-9]*(\.[0-9]+)?)([eE]-?[0-9]+)?[\s]*(mol_per_m_per_A_per_
↪s|mol_per_cm_per_uA_per_ms)"/>
  </xs:restriction>
</xs:simpleType>
```

specificCapacitance

Dimensions M^{-1} L^{-4} T^{4} I^{2}

Units

- Defined unit: *F_per_m2*
- Defined unit: *uF_per_cm2*

substance

Dimensions N^1

Units

- Defined unit: *mol*

temperature

Dimensions K^1

Units

- Defined unit: *K*
- Defined unit: *degC*

Schema

```
<xs:simpleType name="Nml2Quantity_temperature">
  <xs:restriction base="xs:string">
    <xs:pattern value="-?([0-9]*(\.[0-9]+)?)([eE]-?[0-9]+)?[\s]*(degC)"/>
  </xs:restriction>
</xs:simpleType>
```

time

Dimensions T^1

Units

- Defined unit: *hour*
- Defined unit: *min*
- Defined unit: *ms*
- Defined unit: *s*

Schema

```
<xs:simpleType name="Nml2Quantity_time">
  <xs:restriction base="xs:string">
    <xs:pattern value="-?([0-9]*(\.[0-9]+)?)([eE]-?[0-9]+)?[\s]*(s|ms)"/>
  </xs:restriction>
</xs:simpleType>
```

voltage

Dimensions M^{1} L^{2} T^{-3} I^{-1}

Units

- Defined unit: *V*
- Defined unit: *mV*

Schema

```
<xs:simpleType name="Nml2Quantity_voltage">
  <xs:restriction base="xs:string">
    <xs:pattern value="-?([0-9]*(\.[0-9]+)?)([eE]-?[0-9]+)?[\s]*(V|mV)"/>
  </xs:restriction>
</xs:simpleType>
```

volume

Dimensions L^{3}

Units

- Defined unit: *cm3*
- Defined unit: *litre*
- Defined unit: *m3*
- Defined unit: *um3*

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 C = 6.24e+18 *e*

C_per_mol

Summary

- Dimension: *charge_per_mole*
- Power of 10: 0

Conversions

- 1 C_per_mol = 1e-06 *nA_ms_per_amol*
- 1 C_per_mol = 1.00e+06 *pC_per_umol*

F

Summary

- Dimension: *capacitance*
- Power of 10: 0

Conversions

- 1 F = 1.00e+09 *nF*
- 1 F = 1.00e+12 *pF*
- 1 F = 1.00e+06 *uF*

F_per_m2

Summary

- Dimension: *specificCapacitance*
- Power of 10: 0

Conversions

- 1 F_per_m2 = 100 *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

Conversions

- 1 J_per_K_per_mol = 1.00e+09 *fJ_per_K_per_umol*

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*
- 1 S_per_cm2 = 1.00e+06 *uS_per_cm2*

S_per_m2

Summary

- Dimension: *conductanceDensity*
- Power of 10: 0

Conversions

- 1 S_per_m2 = 0.0001 *S_per_cm2*
- 1 S_per_m2 = 0.1 *mS_per_cm2*
- 1 S_per_m2 = 100 *uS_per_cm2*

V

Summary

- Dimension: *voltage*
- Power of 10: 0

Conversions

- 1 V = 1000 *mV*

cm

Summary

- Dimension: *length*
- Power of 10: -2

Conversions

- 1 cm = 0.01 *m*
- 1 cm = 10000 *um*

cm2

Summary

- Dimension: *area*
- Power of 10: -4

Conversions

- 1 cm2 = 0.0001 *m2*
- 1 cm2 = 1.00e+08 *um2*

cm3

Summary

- Dimension: *volume*
- Power of 10: -6

Conversions

- 1 cm3 = 0.001 *litre*
- 1 cm3 = 1e-06 *m3*
- 1 cm3 = 1.00e+12 *um3*

cm_per_ms

Summary

- Dimension: *permeability*
- Power of 10: 1

Conversions

- 1 cm_per_ms = 1000 *cm_per_s*
- 1 cm_per_ms = 10 *m_per_s*
- 1 cm_per_ms = 10000 *um_per_ms*

cm_per_s

Summary

- Dimension: *permeability*
- Power of 10: -2

Conversions

- 1 cm_per_s = 0.001 *cm_per_ms*
- 1 cm_per_s = 0.01 *m_per_s*
- 1 cm_per_s = 10 *um_per_ms*

degC

Summary

- Dimension: *temperature*
- Power of 10: 0
- Offset: 273.15

Conversions

- 1 degC = 274.15 *K*

e

Summary

- Dimension: *charge*
- Power of 10: 0
- Scale: 1.602176634e-19

Conversions

- 1 e = 1.6022e-19 *C*

fJ_per_K_per_umol

Summary

- Dimension: *idealGasConstantDims*
- Power of 10: -9

Conversions

- 1 fJ_per_K_per_umol = 1e-09 *J_per_K_per_mol*

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 *cm3*
- 1 litre = 0.001 *m3*
- 1 litre = 1.00e+15 *um3*

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 *cm2*
- 1 m2 = 1.00e+12 *um2*

m3

Summary

- Dimension: *volume*
- Power of 10: 0

Conversions

- 1 m3 = 1.00e+06 *cm3*
- 1 m3 = 1000 *litre*
- 1 m3 = 1.00e+18 *um3*

mA_per_cm2

Summary

- Dimension: *currentDensity*
- Power of 10: 1

Conversions

- 1 mA_per_cm2 = 10 *A_per_m2*
- 1 mA_per_cm2 = 1000 *uA_per_cm2*

mM

Summary

- Dimension: *concentration*
- Power of 10: 0

Conversions

- 1 mM = 0.001 *M*
- 1 mM = 1e-06 *mol_per_cm3*
- 1 mM = 1 *mol_per_m3*

mS

Summary

- Dimension: *conductance*
- Power of 10: -3

Conversions

- 1 mS = 0.001 *S*
- 1 mS = 1.00e+06 *nS*
- 1 mS = 1.00e+09 *pS*
- 1 mS = 1000 *uS*

mS_per_cm2

Summary

- Dimension: *conductanceDensity*
- Power of 10: 1

Conversions

- 1 mS_per_cm2 = 0.001 *S_per_cm2*
- 1 mS_per_cm2 = 10 *S_per_m2*
- 1 mS_per_cm2 = 1000 *uS_per_cm2*

mV

Summary

- Dimension: *voltage*
- Power of 10: -3

Conversions

- 1 mV = 0.001 *V*

m_per_s

Summary

- Dimension: *permeability*
- Power of 10: 0

Conversions

- 1 m_per_s = 0.1 *cm_per_ms*
- 1 m_per_s = 100 *cm_per_s*
- 1 m_per_s = 1000 *um_per_ms*

min

Summary

- Dimension: *time*
- Power of 10: 0
- Scale: 60.0

Conversions

- 1 min = 0.016667 *hour*
- 1 min = 6.00e+04 *ms*
- 1 min = 60 *s*

mol

Summary

- Dimension: *substance*
- Power of 10: 0

mol_per_cm3

Summary

- Dimension: *concentration*
- Power of 10: 6

Conversions

- 1 mol_per_cm3 = 1000 *M*
- 1 mol_per_cm3 = 1.00e+06 *mM*
- 1 mol_per_cm3 = 1.00e+06 *mol_per_m3*

mol_per_cm_per_uA_per_ms

Summary

- Dimension: *rho_factor*
- Power of 10: 11

Conversions

- 1 mol_per_cm_per_uA_per_ms = 1.00e+11 *mol_per_m_per_A_per_s*
- 1 mol_per_cm_per_uA_per_ms = 1000 *umol_per_cm_per_nA_per_ms*

mol_per_m3

Summary

- Dimension: *concentration*
- Power of 10: 0

Conversions

- 1 mol_per_m3 = 0.001 *M*
- 1 mol_per_m3 = 1 *mM*
- 1 mol_per_m3 = 1e-06 *mol_per_cm3*

mol_per_m_per_A_per_s

Summary

- Dimension: *rho_factor*
- Power of 10: 0

Conversions

- 1 mol_per_m_per_A_per_s = 1e-11 *mol_per_cm_per_uA_per_ms*
- 1 mol_per_m_per_A_per_s = 1e-08 *umol_per_cm_per_nA_per_ms*

ms

Summary

- Dimension: *time*
- Power of 10: -3

Conversions

- 1 ms = 2.7778e-07 *hour*
- 1 ms = 1.6667e-05 *min*
- 1 ms = 0.001 *s*

nA

Summary

- Dimension: *current*
- Power of 10: -9

Conversions

- 1 nA = 1e-09 *A*
- 1 nA = 1000 *pA*
- 1 nA = 0.001 *uA*

nA_ms_per_amol

Summary

- Dimension: *charge_per_mole*
- Power of 10: 6

Conversions

- 1 nA_ms_per_amol = 1.00e+06 *C_per_mol*
- 1 nA_ms_per_amol = 1.00e+12 *pC_per_umol*

nF

Summary

- Dimension: *capacitance*
- Power of 10: -9

Conversions

- 1 nF = 1e-09 *F*
- 1 nF = 1000 *pF*
- 1 nF = 0.001 *uF*

nS

Summary

- Dimension: *conductance*
- Power of 10: -9

Conversions

- 1 nS = 1e-09 *S*
- 1 nS = 1e-06 *mS*
- 1 nS = 1000 *pS*
- 1 nS = 0.001 *uS*

nS_per_mV

Summary

- Dimension: *conductance_per_voltage*
- Power of 10: -6

Conversions

- 1 nS_per_mV = 1e-06 *S_per_V*

ohm

Summary

- Dimension: *resistance*
- Power of 10: 0

Conversions

- 1 ohm = 1e-06 *Mohm*
- 1 ohm = 0.001 *kohm*

ohm_cm

Summary

- Dimension: *resistivity*
- Power of 10: -2

Conversions

- 1 ohm_cm = 0.001 *kohm_cm*
- 1 ohm_cm = 0.01 *ohm_m*

ohm_m

Summary

- Dimension: *resistivity*
- Power of 10: 0

Conversions

- 1 ohm_m = 0.1 *kohm_cm*
- 1 ohm_m = 100 *ohm_cm*

pA

Summary

- Dimension: *current*
- Power of 10: -12

Conversions

- 1 pA = 1e-12 *A*
- 1 pA = 0.001 *nA*
- 1 pA = 1e-06 *uA*

pC_per_umol

Summary

- Dimension: *charge_per_mole*
- Power of 10: -6

Conversions

- 1 pC_per_umol = 1e-06 *C_per_mol*
- 1 pC_per_umol = 1e-12 *nA_ms_per_amol*

pF

Summary

- Dimension: *capacitance*
- Power of 10: -12

Conversions

- 1 pF = 1e-12 *F*
- 1 pF = 0.001 *nF*
- 1 pF = 1e-06 *uF*

pS

Summary

- Dimension: *conductance*
- Power of 10: -12

Conversions

- 1 pS = 1e-12 *S*
- 1 pS = 1e-09 *mS*
- 1 pS = 0.001 *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: 0.01666666667

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 uA = 1e-06 *A*
- 1 uA = 1000 *nA*
- 1 uA = 1.00e+06 *pA*

uA_per_cm2

Summary

- Dimension: *currentDensity*
- Power of 10: -2

Conversions

- 1 uA_per_cm2 = 0.01 *A_per_m2*
- 1 uA_per_cm2 = 0.001 *mA_per_cm2*

uF

Summary

- Dimension: *capacitance*
- Power of 10: -6

Conversions

- 1 uF = 1e-06 *F*
- 1 uF = 1000 *nF*
- 1 uF = 1.00e+06 *pF*

uF_per_cm2

Summary

- Dimension: *specificCapacitance*
- Power of 10: -2

Conversions

- 1 uF_per_cm2 = 0.01 *F_per_m2*

uS

Summary

- Dimension: *conductance*
- Power of 10: -6

Conversions

- 1 uS = 1e-06 *S*
- 1 uS = 0.001 *mS*
- 1 uS = 1000 *nS*
- 1 uS = 1.00e+06 *pS*

uS_per_cm2

Summary

- Dimension: *conductanceDensity*
- Power of 10: -2

Conversions

- 1 uS_per_cm2 = 1e-06 *S_per_cm2*
- 1 uS_per_cm2 = 0.01 *S_per_m2*
- 1 uS_per_cm2 = 0.001 *mS_per_cm2*

um

Summary

- Dimension: *length*
- Power of 10: -6

Conversions

- 1 um = 0.0001 *cm*
- 1 um = 1e-06 *m*

um2

Summary

- Dimension: *area*
- Power of 10: -12

Conversions

- 1 um2 = 1e-08 *cm2*
- 1 um2 = 1e-12 *m2*

um3

Summary

- Dimension: *volume*
- Power of 10: -18

Conversions

- 1 um3 = 1e-12 *cm3*
- 1 um3 = 1e-15 *litre*
- 1 um3 = 1e-18 *m3*

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*

umol_per_cm_per_nA_per_ms

Summary

- Dimension: *rho_factor*
- Power of 10: 8

Conversions

- 1 umol_per_cm_per_nA_per_ms = 0.001 *mol_per_cm_per_uA_per_ms*
- 1 umol_per_cm_per_nA_per_ms = 1.00e+08 *mol_per_m_per_A_per_s*

13.1.3 NeuroMLCoreCompTypes

Original ComponentType definitions: NeuroMLCoreCompTypes.xml. Schema against which NeuroML based on these should be valid: NeuroML_v2.3.xsd. Generated on 14/08/24 from this commit. Please file any issues or questions at the issue tracker here.

---

notes

Human readable notes/description for a Component.

Schema

```
<xs:simpleType name="Notes">
  <xs:annotation>
    <xs:documentation>Textual human readable notes related to the element in question.
 ↪ It's useful to put these into
        the NeuroML files instead of XML comments, as the notes can be extracted and␣
↪repeated in the files to which the NeuroML is mapped.
           </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string"/>
</xs:simpleType>
```

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

rdf:RDF	*rdf_RDF*

Children list

property	*property*

Schema

```
<xs:complexType name="Annotation">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:sequence>
        <xs:any processContents="skip" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Annotation
from neuroml.utils import component_factory

variable = component_factory(
    Annotation,
    anytypeobjs_=None,
)
```

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

Schema

```
<xs:complexType name="Property">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="tag" type="xs:string" use="required"/>
      <xs:attribute name="value" type="xs:string" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Property
from neuroml.utils import component_factory

variable = component_factory(
    Property,
    tag: 'a string (required)' = None,
    value: 'a string (required)' = None,
)
```

Usage: XML

```
<property tag="numberInternalDivisions" value="9"/>
```

baseStandalone

Base type of any Component which can have *notes*, *annotation*, or a *property* list.

Child list

notes	*notes*
annotation	*annotation*

Children list

property	*property*

Schema

```
<xs:complexType name="Standalone">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:element name="notes" type="Notes" minOccurs="0"/>
        <xs:element name="property" type="Property" minOccurs="0" maxOccurs="unbounded
↪"/>
        <xs:element name="annotation" type="Annotation" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="metaid" type="MetaId" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

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:Descr	*rdf_Description*

rdf_Description

Structured block in an *annotation* based on RDF.

Text fields

rdf:about

Child list

bqbiol:en	*bqbiol_encodes*
bqbiol:ha	*bqbiol_hasPart*
bqbiol:ha	*bqbiol_hasProperty*
bqbiol:ha	*bqbiol_hasVersion*
bqbiol:is	*bqbiol_is*
bqbiol:isI	*bqbiol_isDescribedBy*
bqbiol:isE	*bqbiol_isEncodedBy*
bqbiol:isH	*bqbiol_isHomologTo*
bqbiol:isP	*bqbiol_isPartOf*
bqbiol:isP	*bqbiol_isPropertyOf*
bqbiol:isV	*bqbiol_isVersionOf*
bqbiol:oc	*bqbiol_occursIn*
bqbiol:ha	*bqbiol_hasTaxon*
bq-model:is	*bqmodel_is*
bq-model:isD	*bq-model_isDescribedBy*
bq-model:isD	*bq-model_isDerivedFrom*

baseBqbiol

Structured block in an *annotation* based on RDF.

Child list

rdf:Bag	*rdf_Bag*

bqbiol_encodes

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

bqbiol_hasPart

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

bqbiol_hasProperty

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

bqbiol_hasVersion

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

bqbiol_is

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

bqbiol_isDescribedBy

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

bqbiol_isEncodedBy

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

bqbiol_isHomologTo

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

bqbiol_isPartOf

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

bqbiol_isPropertyOf

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

bqbiol_isVersionOf

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

Text fields

xmlns:bqbio

bqbiol_occursIn

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

bqbiol_hasTaxon

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

bqmodel_is

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

bqmodel_isDescribedBy

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

Text fields

xmlns:bqmo

bqmodel_isDerivedFrom

extends baseBqbiol

See http://co.mbine.org/standards/qualifiers.

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 *point3DWithDiam* for details.	Dimensionless
x	x coordinate of the point. Note: no dimension used, see description of *point3DWithDiam* for details.	Dimensionless
y	y coordinate of the ppoint. Note: no dimension used, see description of *point3DWithDiam* for details.	Dimensionless
z	z coordinate of the ppoint. Note: no dimension used, see description of *point3DWithDiam* for details.	Dimensionless

Constants

MICRON = 1um		*length*

Derived parameters

radius	A dimensional quantity given by half the _diameter.	*length*

radius = MICRON * diameter / 2

xLength	A version of _x with dimension length.	*length*

xLength = MICRON * x

yLength	A version of _y with dimension length.	*length*

yLength = MICRON * y

zLength	A version of _z with dimension length.	*length*

zLength = MICRON * z

Schema

```
<xs:complexType name="Point3DWithDiam">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="x" type="xs:double" use="required"/>
      <xs:attribute name="y" type="xs:double" use="required"/>
      <xs:attribute name="z" type="xs:double" use="required"/>
      <xs:attribute name="diameter" type="DoubleGreaterThanZero" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Point3DWithDiam
from neuroml.utils import component_factory

variable = component_factory(
    Point3DWithDiam,
    x: 'a double (required)' = None,
```

(continues on next page)

(continued from previous page)

```
    y: 'a double (required)' = None,
    z: 'a double (required)' = None,
    diameter: 'a DoubleGreaterThanZero (required)' = None,
)
```

13.1.4 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 *segment*s) and *biophysicalProperties* separately.

Original ComponentType definitions: Cells.xml. Schema against which NeuroML based on these should be valid: NeuroML_v2.3.xsd. Generated on 14/08/24 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 *segment*s) which can be used in a *population*.

Schema

```
<xs:complexType name="BaseCell">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="neuroLexId" type="NeuroLexId" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BaseCell
from neuroml.utils import component_factory

variable = component_factory(
    BaseCell,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    extensiontype_=None,
)
```

baseSpikingCell

extends baseCell

Base type of any cell which can emit **spike** events.

Event Ports

spike	Spike event	Direction: out

baseCellMembPot

extends 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	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out

baseCellMembPotDL

extends 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 *(from* baseSpikingCell*)*	Direction: out

baseChannelPopulation

extends baseVoltageDepPointCurrent

Base type for any current produced by a population of channels, all of which are of type **ionChannel**.

Component References

ionChannel		*baseIonChannel*

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepPointCurrent*)*	*voltage*

channelPopulation

extends 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

Constants

vShift = 0mV	Set to a constant 0mV here to allow ion channels which use _vShift in their rate variable expressions to be used with *channelPopulation*, not just with *channelDensityVShift* (where _vShift would be explicitly set)	*voltage*

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.

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepPointCurrent*)*	*voltage*

Dynamics

Structure

CHILD INSTANCE: **ionChannel**

Derived Variables

channelg = ionChannel->g

geff = channelg * number

i = geff * (erev - v) (exposed as **i**)

Schema

```
<xs:complexType name="ChannelPopulation">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:element name="variableParameter" type="VariableParameter" minOccurs="0"␣
↪maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="ionChannel" type="NmlId" use="required"/>
      <xs:attribute name="number" type="NonNegativeInteger" use="required"/>
```

(continues on next page)

(continued from previous page)

```
      <xs:attribute name="erev" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="segmentGroup" type="NmlId" use="optional" default="all"/>
      <xs:attribute name="segment" type="NonNegativeInteger" use="optional"/>
      <xs:attribute name="ion" type="NmlId" use="required">
        <xs:annotation>
            </xs:annotation>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelPopulation
from neuroml.utils import component_factory

variable = component_factory(
    ChannelPopulation,
    id: 'a NmlId (required)' = None,
    ion_channel: 'a NmlId (required)' = None,
    number: 'a NonNegativeInteger (required)' = None,
    erev: 'a Nml2Quantity_voltage (required)' = None,
    segment_groups: 'a NmlId (optional)' = 'all',
    segments: 'a NonNegativeInteger (optional)' = None,
    ion: 'a NmlId (required)' = None,
    variable_parameters: 'list of VariableParameter(s) (optional)' = None,
)
```

Usage: XML

```
<channelPopulation id="naChansDend" ionChannel="NaConductance" segment="2" number=
↪"120000" erev="50mV" ion="na"/>
```

channelPopulationNernst

extends 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 *baseIonChannel*) to produce the total conductance	Dimensionless

Constants

R = 8.3144621 J_per_K_per_mol		*idealGasConstantDims*
zCa = 2		Dimensionless
F = 96485.3 C_per_mol		*charge_per_mole*
vShift = 0mV	Set to a constant 0mV here to allow ion channels which use _vShift in their rate variable expressions to be used with *channelPopulation*, not just with *channelDensityVShift* (where _vShift would be explicitly set)	*voltage*

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.

Exposures

erev	The reversal potential of the current produced, calculated from _caConcExt and _caConc	*voltage*
i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

caConc	The internal Ca2+ concentration, as calculated/exposed by the parent Component	*concentration*
caConcExt	The external Ca2+ concentration, as calculated/exposed by the parent Component	*concentration*
temperature	The temperature to use in the calculation of _erev. Note this is generally exposed by a *networkWithTemperature*.	*temperature*
v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepPointCurrent*)*	*voltage*

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

ionChannel	*baseIonChannel*

Exposures

iDensity	*currentDensity*

Requirements

v	*voltage*

baseChannelDensityCond

extends baseChannelDensity

Base type for distributed conductances on an area of a cell producing a (not necessarily ohmic) current.

Parameters

conductanceDensity	*conductanceDensity*

Exposures

gDensity		*conductanceDensity*
iDensity	*(from* baseChannelDensity*)*	*currentDensity*

Requirements

v	*(from* baseChannelDensity*)*	*voltage*

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
segmentGroup

Child list

inhomogeneousValue	*inhomogeneousValue*

Schema

```
<xs:complexType name="VariableParameter">
  <xs:sequence>
    <xs:element name="inhomogeneousValue" type="InhomogeneousValue" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="parameter" type="xs:string" use="required"/>
  <xs:attribute name="segmentGroup" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import VariableParameter
from neuroml.utils import component_factory

variable = component_factory(
    VariableParameter,
    parameter: 'a string (required)' = None,
    segment_groups: 'a string (required)' = None,
    inhomogeneous_value: 'a InhomogeneousValue (optional)' = None,
)
```

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

Schema

```
<xs:complexType name="InhomogeneousValue">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="inhomogeneousParameter" type="xs:string" use="required"/>
      <xs:attribute name="value" type="xs:string" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import InhomogeneousValue
from neuroml.utils import component_factory

variable = component_factory(
    InhomogeneousValue,
    inhomogeneous_parameters: 'a string (required)' = None,
    value: 'a string (required)' = None,
)
```

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	*voltage*

Constants

ZERO_CURR_DENS A_per_m2	=	0	*currentDensity*

Text fields

segmentGroup	
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	*variableParameter*

Exposures

iDensity	*(from* baseChannelDensity*)*	*currentDensity*

Requirements

v	*(from* baseChannelDensity*)*	*voltage*

Dynamics

Structure
CHILD INSTANCE: **ionChannel**

Derived Variables
iDensity = ZERO_CURR_DENS (exposed as **iDensity**)

Schema

```
<xs:complexType name="ChannelDensityNonUniform">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:element name="variableParameter" type="VariableParameter" minOccurs="0"␣
↪maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="ionChannel" type="NmlId" use="required"/>
      <xs:attribute name="erev" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="ion" type="NmlId" use="required">
        <xs:annotation>
            </xs:annotation>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelDensityNonUniform
from neuroml.utils import component_factory

variable = component_factory(
    ChannelDensityNonUniform,
    id: 'a NmlId (required)' = None,
    ion_channel: 'a NmlId (required)' = None,
    erev: 'a Nml2Quantity_voltage (required)' = None,
    ion: 'a NmlId (required)' = None,
    variable_parameters: 'list of VariableParameter(s) (optional)' = None,
)
```

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.

Constants

ZERO_CURR_DENS A_per_m2	=	0	*currentDensity*

Text fields

segmentGroup	
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	*variableParameter*

Exposures

iDensity	*(from* baseChannelDensity*)*	*currentDensity*

Requirements

v	*(from* baseChannelDensity*)*	*voltage*

Dynamics

Structure
CHILD INSTANCE: **ionChannel**

Derived Variables
iDensity = ZERO_CURR_DENS (exposed as **iDensity**)

Schema

```
<xs:complexType name="ChannelDensityNonUniformNernst">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:element name="variableParameter" type="VariableParameter" minOccurs="0"␣
↪maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="ionChannel" type="NmlId" use="required"/>
      <xs:attribute name="ion" type="NmlId" use="required">
        <xs:annotation>
            </xs:annotation>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelDensityNonUniformNernst
from neuroml.utils import component_factory

variable = component_factory(
    ChannelDensityNonUniformNernst,
    id: 'a NmlId (required)' = None,
    ion_channel: 'a NmlId (required)' = None,
    ion: 'a NmlId (required)' = None,
    variable_parameters: 'list of VariableParameter(s) (optional)' = None,
)
```

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.

Constants

ZERO_CURR_DENS A_per_m2	=	0	*currentDensity*

Text fields

segmentGroup	
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	*variableParameter*

Exposures

iDensity	*(from* baseChannelDensity*)*	*currentDensity*

Requirements

v	*(from* baseChannelDensity*)*	*voltage*

Dynamics

Structure
CHILD INSTANCE: **ionChannel**

Derived Variables
iDensity = ZERO_CURR_DENS (exposed as **iDensity**)

Schema

```
<xs:complexType name="ChannelDensityNonUniformGHK">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:element name="variableParameter" type="VariableParameter" minOccurs="0"␣
↪maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="ionChannel" type="NmlId" use="required"/>
      <xs:attribute name="ion" type="NmlId" use="required">
        <xs:annotation>
            </xs:annotation>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelDensityNonUniformGHK
from neuroml.utils import component_factory

variable = component_factory(
    ChannelDensityNonUniformGHK,
    id: 'a NmlId (required)' = None,
    ion_channel: 'a NmlId (required)' = None,
    ion: 'a NmlId (required)' = None,
    variable_parameters: 'list of VariableParameter(s) (optional)' = None,
)
```

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

condDensity	*(from* baseChannelDensityCond*)*	*conductanceDensity*
erev	The reversal potential of the current produced	*voltage*

Constants

vShift = 0mV	Set to a constant 0mV here to allow ion channels which use _vShift in their rate variable expressions to be used with *channelDensity*, not just with *channelDensityVShift* (where _vShift would be explicitly set)	*voltage*

Text fields

segmentGroup	Which *segmentGroup* the channelDensity is placed on. If this is missing, it implies it is placed on all _segment_s of the *cell*
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.

Exposures

gDensity	*(from* baseChannelDensityCond*)*	*conductanceDensity*
iDensity	*(from* baseChannelDensity*)*	*currentDensity*

Requirements

v	*(from* baseChannelDensity*)*	*voltage*

Dynamics

Structure

CHILD INSTANCE: **ionChannel**

Derived Variables

channelf = ionChannel->fopen

gDensity = condDensity * channelf (exposed as **gDensity**)

iDensity = gDensity * (erev - v) (exposed as **iDensity**)

Schema

```
<xs:complexType name="ChannelDensity">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:element name="variableParameter" type="VariableParameter" minOccurs="0"␣
↪maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="ionChannel" type="NmlId" use="required"/>
      <xs:attribute name="condDensity" type="Nml2Quantity_conductanceDensity" use=
↪"optional"/>
      <xs:attribute name="erev" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="segmentGroup" type="NmlId" use="optional" default="all"/>
      <xs:attribute name="segment" type="NonNegativeInteger" use="optional"/>
      <xs:attribute name="ion" type="NmlId" use="required">
        <xs:annotation>
            </xs:annotation>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelDensity
from neuroml.utils import component_factory

variable = component_factory(
    ChannelDensity,
    id: 'a NmlId (required)' = None,
    ion_channel: 'a NmlId (required)' = None,
    cond_density: 'a Nml2Quantity_conductanceDensity (optional)' = None,
    erev: 'a Nml2Quantity_voltage (required)' = None,
    segment_groups: 'a NmlId (optional)' = 'all',
    segments: 'a NonNegativeInteger (optional)' = None,
    ion: 'a NmlId (required)' = None,
    variable_parameters: 'list of VariableParameter(s) (optional)' = None,
    extensiontype_=None,
)
```

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

condDensity	*(from* baseChannelDensityCond*)*	*conductanceDensity*
erev	The reversal potential of the current produced *(from* channelDensity*)*	*voltage*
vShift		*voltage*

Text fields

segmentGroup	Which *segmentGroup* the channelDensity is placed on. If this is missing, it implies it is placed on all _segment_s of the *cell*
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.

Exposures

gDensity	*(from* baseChannelDensityCond*)*	*conductanceDensity*
iDensity	*(from* baseChannelDensity*)*	*currentDensity*

Requirements

v	*(from* baseChannelDensity*)*	*voltage*

Schema

```
<xs:complexType name="ChannelDensityVShift">
  <xs:complexContent>
    <xs:extension base="ChannelDensity">
      <xs:attribute name="vShift" type="Nml2Quantity_voltage" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelDensityVShift
from neuroml.utils import component_factory

variable = component_factory(
    ChannelDensityVShift,
    id: 'a NmlId (required)' = None,
    ion_channel: 'a NmlId (required)' = None,
    cond_density: 'a Nml2Quantity_conductanceDensity (optional)' = None,
    erev: 'a Nml2Quantity_voltage (required)' = None,
    segment_groups: 'a NmlId (optional)' = 'all',
    segments: 'a NonNegativeInteger (optional)' = None,
    ion: 'a NmlId (required)' = None,
    variable_parameters: 'list of VariableParameter(s) (optional)' = None,
    v_shift: 'a Nml2Quantity_voltage (required)' = None,
)
```

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

condDensity	*(from* baseChannelDensityCond*)*	*conductanceDensity*

Constants

R = 8.3144621 J_per_K_per_mol	*idealGasConstantDims*
zCa = 2	Dimensionless
F = 96485.3 C_per_mol	*charge_per_mole*

Text fields

segmentGroup	Which *segmentGroup* the channelDensityNernst is placed on. If this is missing, it implies it is placed on all _segment_s of the *cell*
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.

Exposures

erev	The reversal potential of the current produced, calculated from caConcExt and caConc	*voltage*
gDensity	*(from* baseChannelDensityCond*)*	*conductanceDensity*
iDensity	*(from* baseChannelDensity*)*	*currentDensity*

Requirements

caConc		*concentration*
caConcExt		*concentration*
temperature		*temperature*
v	*(from* baseChannelDensity*)*	*voltage*

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**)

Schema

```
<xs:complexType name="ChannelDensityNernst">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:element name="variableParameter" type="VariableParameter" minOccurs="0"␣
↪maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="ionChannel" type="NmlId" use="required"/>
      <xs:attribute name="condDensity" type="Nml2Quantity_conductanceDensity" use=
↪"optional"/>
      <xs:attribute name="segmentGroup" type="NmlId" use="optional" default="all"/>
      <xs:attribute name="segment" type="NmlId" use="optional"/>
      <xs:attribute name="ion" type="NmlId" use="required">
        <xs:annotation>
            </xs:annotation>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelDensityNernst
from neuroml.utils import component_factory

variable = component_factory(
    ChannelDensityNernst,
    id: 'a NmlId (required)' = None,
    ion_channel: 'a NmlId (required)' = None,
    cond_density: 'a Nml2Quantity_conductanceDensity (optional)' = None,
    segment_groups: 'a NmlId (optional)' = 'all',
    segments: 'a NonNegativeInteger (optional)' = None,
    ion: 'a NmlId (required)' = None,
    variable_parameters: 'list of VariableParameter(s) (optional)' = None,
    extensiontype_=None,
)
```

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

condDensity	*(from* baseChannelDensityCond*)*	*conductanceDensity*

Constants

R = 8.3144621 J_per_K_per_mol	*idealGasConstantDims*
zCa = 2	Dimensionless
F = 96485.3 C_per_mol	*charge_per_mole*

Text fields

segmentGroup	Which *segmentGroup* the channelDensityNernstCa2 is placed on. If this is missing, it implies it is placed on all _segment_s of the *cell*
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.

Exposures

erev	The reversal potential of the current produced	*voltage*
gDensity	*(from* baseChannelDensityCond*)*	*conductanceDensity*
iDensity	*(from* baseChannelDensity*)*	*currentDensity*

Requirements

caConc2		*concentration*
caConcExt2		*concentration*
temperature		*temperature*
v	*(from* baseChannelDensity*)*	*voltage*

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**)

Schema

```
<xs:complexType name="ChannelDensityNernstCa2">
  <xs:complexContent>
    <xs:extension base="ChannelDensityNernst">
      </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelDensityNernstCa2
from neuroml.utils import component_factory

variable = component_factory(
    ChannelDensityNernstCa2,
    id: 'a NmlId (required)' = None,
    ion_channel: 'a NmlId (required)' = None,
    cond_density: 'a Nml2Quantity_conductanceDensity (optional)' = None,
    segment_groups: 'a NmlId (optional)' = 'all',
    segments: 'a NonNegativeInteger (optional)' = None,
    ion: 'a NmlId (required)' = None,
    variable_parameters: 'list of VariableParameter(s) (optional)' = None,
)
```

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	*permeability*

Constants

R = 8.3144621 J_per_K_per_mol	*idealGasConstantDims*
zCa = 2	Dimensionless
F = 96485.3 C_per_mol	*charge_per_mole*

Text fields

segmentGroup	Which *segmentGroup* the channelDensityGHK is placed on. If this is missing, it implies it is placed on all _segment_s of the *cell*
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.

Exposures

iDensity	*(from* baseChannelDensity*)*	*currentDensity*

Requirements

caConc		*concentration*
caConcExt		*concentration*
temperature		*temperature*
v	*(from* baseChannelDensity*)*	*voltage*

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**)

Schema

```
<xs:complexType name="ChannelDensityGHK">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:attribute name="ionChannel" type="NmlId" use="required"/>
      <xs:attribute name="permeability" type="Nml2Quantity_permeability" use="required
↪"/>
      <xs:attribute name="segmentGroup" type="NmlId" use="optional" default="all"/>
      <xs:attribute name="segment" type="NmlId" use="optional"/>
      <xs:attribute name="ion" type="NmlId" use="required">
        <xs:annotation>
            </xs:annotation>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelDensityGHK
from neuroml.utils import component_factory

variable = component_factory(
    ChannelDensityGHK,
    id: 'a NmlId (required)' = None,
    ion_channel: 'a NmlId (required)' = None,
    permeability: 'a Nml2Quantity_permeability (required)' = None,
    segment_groups: 'a NmlId (optional)' = 'all',
    segments: 'a NonNegativeInteger (optional)' = None,
    ion: 'a NmlId (required)' = None,
)
```

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

condDensity	*(from* baseChannelDensityCond*)*	*conductanceDensity*

Constants

VOLT_SCALE = 1 mV	*voltage*
CONC_SCALE = 1 mM	*concentration*
TEMP_SCALE = 1 K	*temperature*

Text fields

segmentGroup	Which *segmentGroup* the channelDensityGHK2 is placed on. If this is missing, it implies it is placed on all _segment_s of the *cell*
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.

Exposures

gDensity	*(from* baseChannelDensityCond*)*	*conductanceDensity*
iDensity	*(from* baseChannelDensity*)*	*currentDensity*

Requirements

caConc		*concentration*
caConcExt		*concentration*
temperature		*temperature*
v	*(from* baseChannelDensity*)*	*voltage*

Dynamics

Structure

CHILD INSTANCE: **ionChannel**

Derived Variables

V = v / VOLT_SCALE

ca_conc_i = caConc / CONC_SCALE

ca_conc_ext = caConcExt / CONC_SCALE

T = temperature / TEMP_SCALE

channelf = ionChannel->fopen

gDensity = condDensity * channelf (exposed as **gDensity**)

tmp = (25 * T) / (293.15 * 2)

Conditional Derived Variables

IF V/tmp = 0. THEN

pOpen = tmp * 1e-3 * (1 - ((ca_conc_i/ca_conc_ext) * exp(V/tmp))) * (1 - (V/tmp)/2)

IF V/tmp != 0. THEN

pOpen = tmp * 1e-3 * (1 - ((ca_conc_i/ca_conc_ext) * exp(V/tmp))) * ((V/tmp) / (exp(V/tmp) - 1))

IF ca_conc_ext > 0 THEN

iDensity = gDensity * pOpen (exposed as **iDensity**)

IF ca_conc_ext <= 0 THEN

iDensity = 0 (exposed as **iDensity**)

Schema

```
<xs:complexType name="ChannelDensityGHK2">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:attribute name="ionChannel" type="NmlId" use="required"/>
      <xs:attribute name="condDensity" type="Nml2Quantity_conductanceDensity" use=
↪"optional"/>
      <xs:attribute name="segmentGroup" type="NmlId" use="optional" default="all"/>
      <xs:attribute name="segment" type="NmlId" use="optional"/>
      <xs:attribute name="ion" type="NmlId" use="required">
        <xs:annotation>
            </xs:annotation>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ChannelDensityGHK2
from neuroml.utils import component_factory

variable = component_factory(
    ChannelDensityGHK2,
    id: 'a NmlId (required)' = None,
    ion_channel: 'a NmlId (required)' = None,
    cond_density: 'a Nml2Quantity_conductanceDensity (optional)' = None,
    segment_groups: 'a NmlId (optional)' = 'all',
    segments: 'a NonNegativeInteger (optional)' = None,
    ion: 'a NmlId (required)' = None,
)
```

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 *(from* baseCellMembPotCap*)*	*capacitance*
thresh	The voltage threshold above which the cell is considered to be _spiking	*voltage*
v0	The initial membrane potential of the cell	*voltage*

Children list

populations	*baseChannelPopulation*

Exposures

iMemb	Total current crossing the cell membrane *(from* baseCellMembPotCap*)*	*current*
iSyn	Total current due to synaptic inputs *(from* baseCellMembPotCap*)*	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out

Attachments

synapses	*basePointCurrent*

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!!!

Parameters

C	Total capacitance of the cell membrane *(from* baseCellMembPotCap*)*	*capacitance*
thresh	The voltage threshold above which the cell is considered to be _spiking	*voltage*
v0	The initial membrane potential of the cell	*voltage*

Children list

populations		*baseChannelPopulation*
concentrationModels		*concentrationModel*

Exposures

caConc		*concentration*
iCa		*current*
iMemb	Total current crossing the cell membrane *(from* baseCellMembPotCap*)*	*current*
iSyn	Total current due to synaptic inputs *(from* baseCellMembPotCap*)*	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out

Attachments

synapses	*basePointCurrent*

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 *point3DWithDiam* for details. *(from* point3DWithDiam*)*	Dimensionless
x	x coordinate of the point. Note: no dimension used, see description of *point3DWithDiam* for details. *(from* point3DWithDiam*)*	Dimensionless
y	y coordinate of the ppoint. Note: no dimension used, see description of *point3DWithDiam* for details. *(from* point3DWithDiam*)*	Dimensionless
z	z coordinate of the ppoint. Note: no dimension used, see description of *point3DWithDiam* for details. *(from* point3DWithDiam*)*	Dimensionless

Derived parameters

radius	A dimensional quantity given by half the _diameter. *(from* point3DWithDiam*)*	*length*

radius = MICRON * diameter / 2

xLength	A version of _x with dimension length. *(from* point3DWithDiam*)*	*length*

xLength = MICRON * x

yLength	A version of _y with dimension length. *(from* point3DWithDiam*)*	*length*

yLength = MICRON * y

zLength	A version of _z with dimension length. *(from* point3DWithDiam*)*	*length*

zLength = MICRON * z

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 *point3DWithDiam* for details. *(from* point3DWithDiam*)*	Dimensionless
x	x coordinate of the point. Note: no dimension used, see description of *point3DWithDiam* for details. *(from* point3DWithDiam*)*	Dimensionless
y	y coordinate of the ppoint. Note: no dimension used, see description of *point3DWithDiam* for details. *(from* point3DWithDiam*)*	Dimensionless
z	z coordinate of the ppoint. Note: no dimension used, see description of *point3DWithDiam* for details. *(from* point3DWithDiam*)*	Dimensionless

Derived parameters

radius	A dimensional quantity given by half the _diameter. *(from* point3DWithDiam*)*	*length*

radius = MICRON * diameter / 2

xLength	A version of _x with dimension length. *(from* point3DWithDiam*)*	*length*

xLength = MICRON * x

yLength	A version of _y with dimension length. *(from* point3DWithDiam*)*	*length*

yLength = MICRON * y

zLength	A version of _z with dimension length. *(from* point3DWithDiam*)*	*length*

zLength = MICRON * z

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
fractionAlong	The fraction along the the parent segment at which this segment is attached. For usage see *proximal*

Schema

```
<xs:complexType name="SegmentParent">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="segment" type="NonNegativeInteger" use="required"/>
      <xs:attribute name="fractionAlong" type="ZeroToOne" use="optional" default="1"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

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, x 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.

Constants

LEN = 1m	*length*

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	*parent*
distal	*distal*
proximal	*proximal*

Exposures

length	*length*
radDist	*length*
surfaceArea	*area*

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**)

Schema

```
<xs:complexType name="Segment">
  <xs:complexContent>
    <xs:extension base="BaseNonNegativeIntegerId">
      <xs:sequence>
        <xs:element name="parent" type="SegmentParent" minOccurs="0"/>
        <xs:element name="proximal" type="Point3DWithDiam" minOccurs="0"/>
        <xs:element name="distal" type="Point3DWithDiam" minOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="optional"/>
      <xs:attribute name="neuroLexId" type="NeuroLexId" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Segment
from neuroml.utils import component_factory

variable = component_factory(
    Segment,
    id: 'a NonNegativeInteger (required)' = None,
    name: 'a string (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    parent: 'a SegmentParent (optional)' = None,
    proximal: 'a Point3DWithDiam (optional)' = None,
    distal: 'a Point3DWithDiam (required)' = None,
)
```

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 *segment*s 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 *segment*s 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

neuroLexId	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	*notes*
annotation	*annotation*

Children list

property	*property*
members	*member*
paths	*path*
subTrees	*subTree*
includes	*include*
inhomogeneousParameter	*inhomogeneousParameter*

Schema

```
<xs:complexType name="SegmentGroup">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:element name="notes" type="Notes" minOccurs="0"/>
        <xs:element name="property" type="Property" minOccurs="0" maxOccurs="unbounded
↪"/>
        <xs:element name="annotation" type="Annotation" minOccurs="0"/>
        <xs:element name="member" type="Member" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="include" type="Include" minOccurs="0" maxOccurs="unbounded"/
↪>
        <xs:element name="path" type="Path" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="subTree" type="SubTree" minOccurs="0" maxOccurs="unbounded"/
↪>
        <xs:element name="inhomogeneousParameter" type="InhomogeneousParameter"␣
↪minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="neuroLexId" type="NeuroLexId" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SegmentGroup
from neuroml.utils import component_factory

variable = component_factory(
    SegmentGroup,
    id: 'a NonNegativeInteger (required)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    notes: 'a string (optional)' = None,
```

(continues on next page)

(continued from previous page)

```
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    members: 'list of Member(s) (optional)' = None,
    includes: 'list of Include(s) (optional)' = None,
    paths: 'list of Path(s) (optional)' = None,
    sub_trees: 'list of SubTree(s) (optional)' = None,
    inhomogeneous_parameters: 'list of InhomogeneousParameter(s) (optional)' = None,
)
```

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

Schema

```
<xs:complexType name="Member">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="segment" type="NonNegativeInteger" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Member
from neuroml.utils import component_factory

variable = component_factory(
    Member,
    segments: 'a NonNegativeInteger (required)' = None,
)
```

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

Schema

```
<xs:complexType name="SegmentEndPoint">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="segment" type="NonNegativeInteger" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

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

Schema

```
<xs:complexType name="SegmentEndPoint">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="segment" type="NonNegativeInteger" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: XML

```
<to segment="2"/>
```

include

Include all members of another *segmentGroup* in this group.

Text fields

href
segment-Group

Schema

```
<xs:complexType name="Include">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="segmentGroup" type="NmlId" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Include
from neuroml.utils import component_factory

variable = component_factory(
    Include,
    segment_groups: 'a NmlId (required)' = None,
)
```

Usage: XML

```
<include href="NML2_SingleCompHHCell.nml"/>
```

```
<include href="NML2_SimpleIonChannel.nml"/>
```

```
<include href="NML2_SimpleIonChannel.nml"/>
```

path

Include all the *segment*s between those specified by *from* and *to*, inclusive.

Child list

from	*from*
to	*to*

Schema

```
<xs:complexType name="Path">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:sequence>
        <xs:element name="from" type="SegmentEndPoint" minOccurs="0"/>
        <xs:element name="to" type="SegmentEndPoint" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Path
from neuroml.utils import component_factory

variable = component_factory(
    Path,
    from_: 'a SegmentEndPoint (optional)' = None,
    to: 'a SegmentEndPoint (optional)' = None,
)
```

Usage: XML

```
<path>
    <from segment="1"/>
    <to segment="2"/>
</path>
```

subTree

Include all the *segment*s distal to that specified by *from* in the *segmentGroup*.

Child list

from	*from*

Schema

```
<xs:complexType name="SubTree">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:choice>
        <xs:element name="from" type="SegmentEndPoint" minOccurs="0"/>
        <xs:element name="to" type="SegmentEndPoint" minOccurs="0"/>
      </xs:choice>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SubTree
from neuroml.utils import component_factory

variable = component_factory(
    SubTree,
    from_: 'a SegmentEndPoint (optional)' = None,
    to: 'a SegmentEndPoint (optional)' = None,
)
```

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	*proximalDetails*
distal	*distalDetails*

Schema

```
<xs:complexType name="InhomogeneousParameter">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:element name="proximal" type="ProximalDetails" minOccurs="0"/>
        <xs:element name="distal" type="DistalDetails" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="variable" type="xs:string" use="required"/>
      <xs:attribute name="metric" type="Metric" use="required"/>
    </xs:extension>
```

(continues on next page)

(continued from previous page)

```
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import InhomogeneousParameter
from neuroml.utils import component_factory

variable = component_factory(
    InhomogeneousParameter,
    id: 'a NmlId (required)' = None,
    variable: 'a string (required)' = None,
    metric: 'a Metric (required)' = None,
    proximal: 'a ProximalDetails (optional)' = None,
    distal: 'a DistalDetails (optional)' = None,
)
```

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"/>
```

proximalDetails

What to do at the proximal point when creating an inhomogeneous parameter.

Text fields

transla- tionStart

Schema

```
<xs:complexType name="ProximalDetails">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="translationStart" type="xs:double" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ProximalDetails
from neuroml.utils import component_factory

variable = component_factory(
    ProximalDetails,
    translation_start: 'a double (required)' = None,
)
```

distalDetails

What to do at the distal point when creating an inhomogeneous parameter.

Text fields

normaliza- tionEnd

Schema

```
<xs:complexType name="DistalDetails">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="normalizationEnd" type="xs:double" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import DistalDetails
from neuroml.utils import component_factory

variable = component_factory(
    DistalDetails,
    normalization_end: 'a double (required)' = None,
)
```

morphology

The collection of *segment*s which specify the 3D structure of the cell, along with a number of *segmentGroup*s.

Children list

segments	*segment*
segmentGroups	*segmentGroup*

Schema

```
<xs:complexType name="Morphology">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:sequence>
        <xs:element name="segment" type="Segment" maxOccurs="unbounded"/>
        <xs:element name="segmentGroup" type="SegmentGroup" minOccurs="0" maxOccurs=
↪"unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Morphology
from neuroml.utils import component_factory

variable = component_factory(
    Morphology,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
```

(continues on next page)

(continued from previous page)

```
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    segments: 'list of Segment(s) (required)' = None,
    segment_groups: 'list of SegmentGroup(s) (optional)' = None,
)
```

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"/>
```

(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	*specificCapacitance*

Dynamics

Derived Variables

specCap = value (exposed as **specCap**)

Schema

```
<xs:complexType name="SpecificCapacitance">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="value" type="Nml2Quantity_specificCapacitance" use="required
↪"/>
      <xs:attribute name="segmentGroup" type="NmlId" use="optional" default="all"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SpecificCapacitance
from neuroml.utils import component_factory

variable = component_factory(
    SpecificCapacitance,
    value: 'a Nml2Quantity_specificCapacitance (required)' = None,
    segment_groups: 'a NmlId (optional)' = 'all',
)
```

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	*voltage*

Schema

```
<xs:complexType name="InitMembPotential">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="value" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="segmentGroup" type="NmlId" use="optional" default="all"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import InitMembPotential
from neuroml.utils import component_factory

variable = component_factory(
    InitMembPotential,
    value: 'a Nml2Quantity_voltage (required)' = None,
    segment_groups: 'a NmlId (optional)' = 'all',
)
```

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	*voltage*

Schema

```
<xs:complexType name="SpikeThresh">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="value" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="segmentGroup" type="NmlId" use="optional" default="all"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SpikeThresh
from neuroml.utils import component_factory

variable = component_factory(
    SpikeThresh,
    value: 'a Nml2Quantity_voltage (required)' = None,
    segment_groups: 'a NmlId (optional)' = 'all',
)
```

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

initMembPotential	*initMembPotential*
spikeThre	*spikeThresh*

Children list

specificCapacitances	*specificCapacitance*
populations	*baseChannelPopulation*
channelDensities	*baseChannelDensity*

Exposures

iCa	*current*
totChanCurrent	*current*
totSpecCap	*specificCapacitance*

Requirements

surfaceArea	*area*

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**)

Schema

```
<xs:complexType name="MembraneProperties">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:sequence>
        <xs:element name="channelPopulation" type="ChannelPopulation" minOccurs="0"␣
↪maxOccurs="unbounded"/>
        <xs:element name="channelDensity" type="ChannelDensity" minOccurs="0"␣
↪maxOccurs="unbounded"/>
        <xs:element name="channelDensityVShift" type="ChannelDensityVShift" minOccurs=
↪"0" maxOccurs="unbounded"/>
        <xs:element name="channelDensityNernst" type="ChannelDensityNernst" minOccurs=
↪"0" maxOccurs="unbounded"/>
        <xs:element name="channelDensityGHK" type="ChannelDensityGHK" minOccurs="0"␣
↪maxOccurs="unbounded"/>
        <xs:element name="channelDensityGHK2" type="ChannelDensityGHK2" minOccurs="0"␣
↪maxOccurs="unbounded"/>
        <xs:element name="channelDensityNonUniform" type="ChannelDensityNonUniform"␣
↪minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="channelDensityNonUniformNernst" type=
↪"ChannelDensityNonUniformNernst" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="channelDensityNonUniformGHK" type=
↪"ChannelDensityNonUniformGHK" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="spikeThresh" type="SpikeThresh" minOccurs="0" maxOccurs=
↪"unbounded"/>
        <xs:element name="specificCapacitance" type="SpecificCapacitance" minOccurs="0
↪" maxOccurs="unbounded"/>
        <xs:element name="initMembPotential" type="InitMembPotential" minOccurs="0"␣
↪maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import MembraneProperties
from neuroml.utils import component_factory

variable = component_factory(
    MembraneProperties,
    channel_populations: 'list of ChannelPopulation(s) (optional)' = None,
    channel_densities: 'list of ChannelDensity(s) (optional)' = None,
    channel_density_v_shifts: 'list of ChannelDensityVShift(s) (optional)' = None,
    channel_density_nernsts: 'list of ChannelDensityNernst(s) (optional)' = None,
    channel_density_ghks: 'list of ChannelDensityGHK(s) (optional)' = None,
    channel_density_ghk2s: 'list of ChannelDensityGHK2(s) (optional)' = None,
    channel_density_non_uniforms: 'list of ChannelDensityNonUniform(s) (optional)' =␣
↪None,
    channel_density_non_uniform_nernsts: 'list of ChannelDensityNonUniformNernst(s)␣
↪(optional)' = None,
    channel_density_non_uniform_ghks: 'list of ChannelDensityNonUniformGHK(s)␣
↪(optional)' = None,
```

(continues on next page)

(continued from previous page)

```
    spike_threshes: 'list of SpikeThresh(s) (required)' = None,
    specific_capacitances: 'list of SpecificCapacitance(s) (required)' = None,
    init_memb_potentials: 'list of InitMembPotential(s) (required)' = None,
    extensiontype_=None,
)
```

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

initMembPotential	*initMembPotential*
spikeThre	*spikeThresh*

Children list

specificCapacitances	*specificCapacitance*
populations	*baseChannelPopulation*
channelDensities	*baseChannelDensity*

Exposures

iCa	(*from* membraneProperties)	*current*
iCa2		*current*
totChanCurrent	(*from* membraneProperties)	*current*
totSpecCap	(*from* membraneProperties)	*specificCapacitance*

Requirements

surfaceArea		*area*
surfaceArea	(*from* membraneProperties)	*area*

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**)

Schema

```
<xs:complexType name="MembraneProperties2CaPools">
  <xs:complexContent>
    <xs:extension base="MembraneProperties">
      <xs:sequence>
        <xs:element name="channelDensityNernstCa2" type="ChannelDensityNernstCa2"␣
↪minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import MembraneProperties2CaPools
from neuroml.utils import component_factory

variable = component_factory(
    MembraneProperties2CaPools,
    channel_populations: 'list of ChannelPopulation(s) (optional)' = None,
    channel_densities: 'list of ChannelDensity(s) (optional)' = None,
    channel_density_v_shifts: 'list of ChannelDensityVShift(s) (optional)' = None,
    channel_density_nernsts: 'list of ChannelDensityNernst(s) (optional)' = None,
    channel_density_ghks: 'list of ChannelDensityGHK(s) (optional)' = None,
    channel_density_ghk2s: 'list of ChannelDensityGHK2(s) (optional)' = None,
    channel_density_non_uniforms: 'list of ChannelDensityNonUniform(s) (optional)' =␣
↪None,
    channel_density_non_uniform_nernsts: 'list of ChannelDensityNonUniformNernst(s)␣
↪(optional)' = None,
    channel_density_non_uniform_ghks: 'list of ChannelDensityNonUniformGHK(s)␣
↪(optional)' = None,
```

(continues on next page)

(continued from previous page)

```
    spike_threshes: 'list of SpikeThresh(s) (required)' = None,
    specific_capacitances: 'list of SpecificCapacitance(s) (required)' = None,
    init_memb_potentials: 'list of InitMembPotential(s) (required)' = None,
    channel_density_nernst_ca2s: 'list of ChannelDensityNernstCa2(s) (optional)' =␣
 ↪None,
)
```

biophysicalProperties

The biophysical properties of the *cell*, including the *membraneProperties* and the *intracellularProperties*.

Child list

membraneProperties	*membraneProperties*
intracellularProperties	*intracellularProperties*

Exposures

totSpecCap	*specificCapacitance*

Dynamics

Derived Variables

totSpecCap = membraneProperties->totSpecCap (exposed as **totSpecCap**)

Schema

```
<xs:complexType name="BiophysicalProperties">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:sequence>
        <xs:element name="membraneProperties" type="MembraneProperties"/>
        <xs:element name="intracellularProperties" type="IntracellularProperties"␣
 ↪minOccurs="0"/>
        <xs:element name="extracellularProperties" type="ExtracellularProperties"␣
 ↪minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
```

(continues on next page)

(continued from previous page)

```
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BiophysicalProperties
from neuroml.utils import component_factory

variable = component_factory(
    BiophysicalProperties,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    membrane_properties: 'a MembraneProperties (required)' = None,
    intracellular_properties: 'a IntracellularProperties (optional)' = None,
    extracellular_properties: 'a ExtracellularProperties (optional)' = None,
)
```

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"/>
    </intracellularProperties>
</biophysicalProperties>
```

```
<biophysicalProperties id="PyrCellChanDist">
    <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>
    <intracellularProperties>
        <resistivity value="0.1 kohm_cm"/>
        <!-- REMOVED UNTIL WE CHECK HOW THE USAGE OF LEMS IMPACTS THIS...
            <biochemistry reactionScheme="InternalCaDynamics"/>  Ref to earlier␣
```

(continues on next page)

(continued from previous page)

```
↪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

membraneProperties2CaPo	*membraneProperties2CaPools*
intracellularProperties2CaPo	*intracellularProperties2CaPools*

Exposures

totSpecCap	*specificCapacitance*

Dynamics

Derived Variables

totSpecCap = membraneProperties2CaPools->totSpecCap (exposed as **totSpecCap**)

Schema

```
<xs:complexType name="BiophysicalProperties2CaPools">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:sequence>
        <xs:element name="membraneProperties2CaPools" type="MembraneProperties2CaPools
↪"/>
        <xs:element name="intracellularProperties2CaPools" type=
↪"IntracellularProperties2CaPools" minOccurs="0"/>
        <xs:element name="extracellularProperties" type="ExtracellularProperties"␣
↪minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BiophysicalProperties2CaPools
from neuroml.utils import component_factory

variable = component_factory(
    BiophysicalProperties2CaPools,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    membrane_properties2_ca_pools: 'a MembraneProperties2CaPools (required)' = None,
    intracellular_properties2_ca_pools: 'a IntracellularProperties2CaPools (optional)
↪' = None,
    extracellular_properties: 'a ExtracellularProperties (optional)' = None,
)
```

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	*resistivity*
speciesList	*species*

Exposures

caConc	*concentration*
caConcExt	*concentration*

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**)

Schema

```
<xs:complexType name="IntracellularProperties">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:sequence>
        <xs:element name="species" type="Species" minOccurs="0" maxOccurs="unbounded"/
↪>
        <xs:element name="resistivity" type="Resistivity" minOccurs="0" maxOccurs=
↪"unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IntracellularProperties
from neuroml.utils import component_factory

variable = component_factory(
    IntracellularProperties,
    species: 'list of Species(s) (optional)' = None,
    resistivities: 'list of Resistivity(s) (optional)' = None,
    extensiontype_=None,
)
```

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

speciesList		*species*
resistivity		*resistivity*

Exposures

caConc	(*from* intracellularProperties)	*concentration*
caConc2		*concentration*
caConcExt	(*from* intracellularProperties)	*concentration*
caConcExt2		*concentration*

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**)

Schema

```
<xs:complexType name="IntracellularProperties2CaPools">
  <xs:complexContent>
    <xs:extension base="IntracellularProperties">
      </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IntracellularProperties2CaPools
from neuroml.utils import component_factory

variable = component_factory(
    IntracellularProperties2CaPools,
    species: 'list of Species(s) (optional)' = None,
    resistivities: 'list of Resistivity(s) (optional)' = None,
)
```

resistivity

The resistivity, or specific axial resistance, of the cytoplasm.

Parameters

value	*resistivity*

Text fields

segment-Group

Schema

```
<xs:complexType name="Resistivity">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="value" type="Nml2Quantity_resistivity" use="required"/>
      <xs:attribute name="segmentGroup" type="NmlId" use="optional" default="all"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Resistivity
from neuroml.utils import component_factory

variable = component_factory(
    Resistivity,
    value: 'a Nml2Quantity_resistivity (required)' = None,
    segment_groups: 'a NmlId (optional)' = 'all',
)
```

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	*concentration*
extConcentration	*concentration*

Requirements

initialConcentration	*concentration*
initialExtConcentration	*concentration*
surfaceArea	*area*

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		*time*
restingConc		*concentration*
shellThickness		*length*

Constants

Faraday = 96485.3C_per_mol	*charge_per_mole*
AREA_SCALE = 1m2	*area*
LENGTH_SCALE = 1m	*length*

Text fields

ion

Exposures

concentration	*(from* concentrationModel*)*	*concentration*
extConcentration	*(from* concentrationModel*)*	*concentration*

Requirements

iCa		*current*
initialConcentration	*(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 **concentration** /dt = iCa / (2 * Faraday * shellVolume) - ((concentration - restingConc) / decayConstant)

Schema

```
<xs:complexType name="DecayingPoolConcentrationModel">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="ion" type="NmlId" use="required">
        <xs:annotation>
            </xs:annotation>
      </xs:attribute>
      <xs:attribute name="restingConc" type="Nml2Quantity_concentration" use="required
↪"/>
      <xs:attribute name="decayConstant" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="shellThickness" type="Nml2Quantity_length" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import DecayingPoolConcentrationModel
from neuroml.utils import component_factory

variable = component_factory(
    DecayingPoolConcentrationModel,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
```

(continues on next page)

(continued from previous page)

```
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    ion: 'a NmlId (required)' = None,
    resting_conc: 'a Nml2Quantity_concentration (required)' = None,
    decay_constant: 'a Nml2Quantity_time (required)' = None,
    shell_thickness: 'a Nml2Quantity_length (required)' = None,
    extensiontype_=None,
)
```

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

decayConstant	*time*
restingConc	*concentration*
rho	*rho_factor*

Text fields

ion

Exposures

concentration	(*from* concentrationModel)	*concentration*
extConcentration	(*from* concentrationModel)	*concentration*

Requirements

iCa		*current*
initialConcentration	*(from* concentrationModel*)*	*concentration*
initialExtConcentration	*(from* concentrationModel*)*	*concentration*
surfaceArea		*area*
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

Time Derivatives

d **concentration** /dt = (iCa/surfaceArea) * rho - ((concentration - restingConc) / decayConstant)

Schema

```
<xs:complexType name="FixedFactorConcentrationModel">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="ion" type="NmlId" use="required">
        <xs:annotation>
            </xs:annotation>
      </xs:attribute>
      <xs:attribute name="restingConc" type="Nml2Quantity_concentration" use="required
↪"/>
      <xs:attribute name="decayConstant" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="rho" type="Nml2Quantity_rhoFactor" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import FixedFactorConcentrationModel
from neuroml.utils import component_factory

variable = component_factory(
    FixedFactorConcentrationModel,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    ion: 'a NmlId (required)' = None,
    resting_conc: 'a Nml2Quantity_concentration (required)' = None,
    decay_constant: 'a Nml2Quantity_time (required)' = None,
    rho: 'a Nml2Quantity_rhoFactor (required)' = None,
)
```

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	*per_time*
phi	*rho_factor*
resting-Conc	*concentration*

Text fields

species

Exposures

concentration	(*from* concentrationModel)	*concentration*
extConcentration	(*from* concentrationModel)	*concentration*

Requirements

iCa		*current*
initialConcentration	(*from* concentrationModel)	*concentration*
initialExtConcentration	(*from* concentrationModel)	*concentration*
surfaceArea		*area*
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

Time Derivatives

d **concentration** /dt = (iCa/surfaceArea) * 1e-9 * phi - ((concentration - restingConc) * beta)

species

Description of a chemical species identified by **ion,** which has internal, **concentration,** and external, **extConcentration** values for its concentration.

Parameters

initialConcentration	*concentration*
initialExtConcentration	*concentration*

Text fields

ion
segmentGroup

Component References

concentrationModel	*concentrationModel*

Exposures

concentration	*concentration*
extConcentration	*concentration*

Dynamics

Structure
CHILD INSTANCE: **concentrationModel**

Derived Variables
concentration = concentrationModel->concentration (exposed as **concentration**)

extConcentration = concentrationModel->extConcentration (exposed as **extConcentration**)

Schema

```
<xs:complexType name="Species">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:attribute name="concentrationModel" type="NmlId" use="required"/>
      <xs:attribute name="ion" type="NmlId" use="optional">
        <xs:annotation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="initialConcentration" type="Nml2Quantity_concentration" use=
↪"required"/>
      <xs:attribute name="initialExtConcentration" type="Nml2Quantity_concentration"␣
↪use="required"/>
      <xs:attribute name="segmentGroup" type="NmlId" use="optional" default="all"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Species
from neuroml.utils import component_factory

variable = component_factory(
    Species,
    id: 'a NmlId (required)' = None,
    concentration_model: 'a NmlId (required)' = None,
    ion: 'a NmlId (optional)' = None,
    initial_concentration: 'a Nml2Quantity_concentration (required)' = None,
    initial_ext_concentration: 'a Nml2Quantity_concentration (required)' = None,
    segment_groups: 'a NmlId (optional)' = 'all',
)
```

cell

extends baseCellMembPot

Cell with *segment*s 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

neuroLexId	

Child list

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

Exposures

caConc		*concentration*
caConcExt		*concentration*
iCa		*current*
iChannels		*current*
iSyn		*current*
spiking		Dimensionless
surfaceArea		*area*
totSpecCap		*specificCapacitance*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out

Attachments

synapses	*basePointCurrent*

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

Schema

```
<xs:complexType name="Cell">
  <xs:complexContent>
    <xs:extension base="BaseCell">
      <xs:sequence>
        <xs:element name="morphology" type="Morphology" minOccurs="0"/>
        <xs:element name="biophysicalProperties" type="BiophysicalProperties"␣
↪minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="morphology" type="NmlId" use="optional">
        <xs:annotation>
            </xs:annotation>
      </xs:attribute>
      <xs:attribute name="biophysicalProperties" type="NmlId" use="optional">
        <xs:annotation>
            </xs:annotation>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Cell
from neuroml.utils import component_factory

variable = component_factory(
    Cell,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    morphology_attr: 'a NmlId (optional)' = None,
    biophysical_properties_attr: 'a NmlId (optional)' = None,
    morphology: 'a Morphology (optional)' = None,
    biophysical_properties: 'a BiophysicalProperties (optional)' = None,
    extensiontype_=None,
)
```

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>
```

(continues on next page)

(continued from previous page)

```
                        <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>
        <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">
            <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 *segment*s 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

neuroLexId	

Child list

biophysicalProperties2CaPc	*biophysicalProperties2CaPools*

Exposures

caConc	*(from* cell*)*	*concentration*
caConc2		*concentration*
caConcExt	*(from* cell*)*	*concentration*
caConcExt2		*concentration*
iCa	*(from* cell*)*	*current*
iCa2		*current*
iChannels	*(from* cell*)*	*current*
iSyn	*(from* cell*)*	*current*
spiking	*(from* cell*)*	Dimensionless
surfaceArea	*(from* cell*)*	*area*
totSpecCap	*(from* cell*)*	*specificCapacitance*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out

Attachments

synapses	*basePointCurrent*

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

Schema

```
<xs:complexType name="Cell2CaPools">
  <xs:complexContent>
    <xs:extension base="Cell">
      <xs:sequence>
        <xs:element name="biophysicalProperties2CaPools" type=
↪"BiophysicalProperties2CaPools" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Cell2CaPools
from neuroml.utils import component_factory

variable = component_factory(
    Cell2CaPools,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    morphology_attr: 'a NmlId (optional)' = None,
    biophysical_properties_attr: 'a NmlId (optional)' = None,
    morphology: 'a Morphology (optional)' = None,
    biophysical_properties: 'a BiophysicalProperties (optional)' = None,
    biophysical_properties2_ca_pools: 'a BiophysicalProperties2CaPools (optional)' =␣
↪None,
)
```

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	*capacitance*

Exposures

iMemb	Total current crossing the cell membrane	*current*
iSyn	Total current due to synaptic inputs	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out

Schema

```
<xs:complexType name="BaseCellMembPotCap">
  <xs:complexContent>
    <xs:extension base="BaseCell">
      <xs:attribute name="C" type="Nml2Quantity_capacitance" use="required">
        <xs:annotation>
          <xs:appinfo>
            <jxb:property name="Cap"/>
          </xs:appinfo>
        </xs:annotation>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BaseCellMembPotCap
from neuroml.utils import component_factory

variable = component_factory(
    BaseCellMembPotCap,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    C: 'a Nml2Quantity_capacitance (required)' = None,
    extensiontype_=None,
)
```

baseIaf

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	*voltage*
thresh	The membrane potential at which to emit a spiking event and reset voltage	*voltage*

Exposures

v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	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		*voltage*
reset	The value the membrane potential is reset to on spiking *(from* baseIaf*)*	*voltage*
tau		*time*
thresh	The membrane potential at which to emit a spiking event and reset voltage *(from* baseIaf*)*	*voltage*

Exposures

v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out

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**

Time Derivatives
d **v** /dt = (leakReversal - v) / tau

Schema

```
<xs:complexType name="IafTauCell">
  <xs:complexContent>
    <xs:extension base="BaseCell">
      <xs:attribute name="leakReversal" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="thresh" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="reset" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="tau" type="Nml2Quantity_time" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IafTauCell
from neuroml.utils import component_factory

variable = component_factory(
    IafTauCell,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
```

(continues on next page)

(continued from previous page)

```
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    leak_reversal: 'a Nml2Quantity_voltage (required)' = None,
    thresh: 'a Nml2Quantity_voltage (required)' = None,
    reset: 'a Nml2Quantity_voltage (required)' = None,
    tau: 'a Nml2Quantity_time (required)' = None,
    extensiontype_=None,
)
```

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	*(from* iafTauCell*)*	*voltage*
refract		*time*
reset	The value the membrane potential is reset to on spiking *(from* baseIaf*)*	*voltage*
tau	*(from* iafTauCell*)*	*time*
thresh	The membrane potential at which to emit a spiking event and reset voltage *(from* baseIaf*)*	*voltage*

Exposures

v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	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

Schema

```
<xs:complexType name="IafTauRefCell">
  <xs:complexContent>
    <xs:extension base="IafTauCell">
      <xs:attribute name="refract" type="Nml2Quantity_time" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IafTauRefCell
from neuroml.utils import component_factory

variable = component_factory(
    IafTauRefCell,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
```

(continues on next page)

(continued from previous page)

```
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    leak_reversal: 'a Nml2Quantity_voltage (required)' = None,
    thresh: 'a Nml2Quantity_voltage (required)' = None,
    reset: 'a Nml2Quantity_voltage (required)' = None,
    tau: 'a Nml2Quantity_time (required)' = None,
    refract: 'a Nml2Quantity_time (required)' = None,
)
```

Usage: XML

```
<iafTauRefCell id="iafTauRef" leakReversal="-50mV" thresh="-55mV" reset="-70mV" tau=
 ↪"30ms" refract="5ms"/>
```

baseIafCapCell

extends 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 *(from* baseCellMembPotCap*)*	*capacitance*
reset		*voltage*
thresh		*voltage*

Exposures

iMemb	Total current crossing the cell membrane *(from* baseCellMembPotCap*)*	*current*
iSyn	Total current due to synaptic inputs *(from* baseCellMembPotCap*)*	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out

iafCell

extends baseIafCapCell

Integrate and fire cell with capacitance **C, leakConductance** and **leakReversal**.

Parameters

C	Total capacitance of the cell membrane *(from* baseCellMembPotCap*)*	*capacitance*
leakConductance		*conductance*
leakReversal		*voltage*
reset	*(from* baseIafCapCell*)*	*voltage*
thresh	*(from* baseIafCapCell*)*	*voltage*

Exposures

iMemb	Total current crossing the cell membrane *(from* baseCellMembPotCap*)*	*current*
iSyn	Total current due to synaptic inputs *(from* baseCellMembPotCap*)*	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out

Attachments

synapses	*basePointCurrent*

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

Schema

```
<xs:complexType name="IafCell">
  <xs:complexContent>
    <xs:extension base="BaseCell">
      <xs:attribute name="leakReversal" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="thresh" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="reset" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="C" type="Nml2Quantity_capacitance" use="required"/>
      <xs:attribute name="leakConductance" type="Nml2Quantity_conductance" use=
↪"required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IafCell
from neuroml.utils import component_factory

variable = component_factory(
    IafCell,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    leak_reversal: 'a Nml2Quantity_voltage (required)' = None,
    thresh: 'a Nml2Quantity_voltage (required)' = None,
    reset: 'a Nml2Quantity_voltage (required)' = None,
    C: 'a Nml2Quantity_capacitance (required)' = None,
    leak_conductance: 'a Nml2Quantity_conductance (required)' = None,
    extensiontype_=None,
)
```

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"/>
```

iafRefCell

extends *iafCell*

Integrate and fire cell with capacitance **C, leakConductance, leakReversal** and refractory period **refract**.

Parameters

C	Total capacitance of the cell membrane *(from* baseCellMembPotCap*)*	*capacitance*
leakConductance	*(from* iafCell*)*	*conductance*
leakReversal	*(from* iafCell*)*	*voltage*
refract		*time*
reset	*(from* baseIafCapCell*)*	*voltage*
thresh	*(from* baseIafCapCell*)*	*voltage*

Exposures

iMemb	Total current crossing the cell membrane *(from* baseCellMembPotCap*)*	*current*
iSyn	Total current due to synaptic inputs *(from* baseCellMembPotCap*)*	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out

Attachments

synapses	*basePointCurrent*

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

Schema

```
<xs:complexType name="IafRefCell">
  <xs:complexContent>
    <xs:extension base="IafCell">
      <xs:attribute name="refract" type="Nml2Quantity_time" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IafRefCell
from neuroml.utils import component_factory

variable = component_factory(
    IafRefCell,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    leak_reversal: 'a Nml2Quantity_voltage (required)' = None,
    thresh: 'a Nml2Quantity_voltage (required)' = None,
    reset: 'a Nml2Quantity_voltage (required)' = None,
    C: 'a Nml2Quantity_capacitance (required)' = None,
    leak_conductance: 'a Nml2Quantity_conductance (required)' = None,
    refract: 'a Nml2Quantity_time (required)' = None,
)
```

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	*voltage*
v0	Initial membrane potential	*voltage*

Constants

MSEC = 1ms	*time*
MVOLT = 1mV	*voltage*

Exposures

U	Membrane recovery variable	Dimensionless
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out

Attachments

synapses	*basePointCurrentDL*

Dynamics

State Variables

v: *voltage* (exposed as **v**)

U: Dimensionless (exposed as **U**)

On Start

v = v0

U = v0 * b / MVOLT

On Conditions

IF v > thresh THEN

v = c * MVOLT

U = U + d

EVENT OUT on port: **spike**

Derived Variables

ISyn = synapses[*]->I(reduce method: add)

Time Derivatives

d **v** /dt = (0.04 * v^2 / MVOLT + 5 * v + (140.0 - U + ISyn) * MVOLT)/MSEC

d **U** /dt = a * (b * v / MVOLT - U) / MSEC

Schema

```
<xs:complexType name="IzhikevichCell">
  <xs:complexContent>
    <xs:extension base="BaseCell">
      <xs:attribute name="v0" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="thresh" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="a" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="b" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="c" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="d" type="Nml2Quantity_none" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IzhikevichCell
from neuroml.utils import component_factory

variable = component_factory(
    IzhikevichCell,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    v0: 'a Nml2Quantity_voltage (required)' = None,
    thresh: 'a Nml2Quantity_voltage (required)' = None,
    a: 'a Nml2Quantity_none (required)' = None,
    b: 'a Nml2Quantity_none (required)' = None,
    c: 'a Nml2Quantity_none (required)' = None,
    d: 'a Nml2Quantity_none (required)' = None,
)
```

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 *(from* baseCellMembPotCap*)*	*capacitance*
a	Time scale of recovery variable u	*per_time*
b	Sensitivity of recovery variable u to subthreshold fluctuations of membrane potential v	*conductance*
c	After-spike reset value of v	*voltage*
d	After-spike increase to u	*current*
k		*conductance_per_voltage*
v0	Initial membrane potential	*voltage*
vpeak	Peak action potential value	*voltage*
vr	Resting membrane potential	*voltage*
vt	Spike threshold	*voltage*

Exposures

iMemb	Total current crossing the cell membrane *(from* baseCellMembPotCap*)*	*current*
iSyn	Total current due to synaptic inputs *(from* baseCellMembPotCap*)*	*current*
u	Membrane recovery variable	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out

Attachments

synapses	*basePointCurrent*

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 = iMemb / C

d **u** /dt = a * (b * (v-vr) - u)

Schema

```
<xs:complexType name="Izhikevich2007Cell">
  <xs:complexContent>
    <xs:extension base="BaseCellMembPotCap">
      <xs:attribute name="v0" type="Nml2Quantity_voltage" use="required"/>
      <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>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Izhikevich2007Cell
from neuroml.utils import component_factory

variable = component_factory(
    Izhikevich2007Cell,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    C: 'a Nml2Quantity_capacitance (required)' = None,
    v0: 'a Nml2Quantity_voltage (required)' = None,
    k: 'a Nml2Quantity_conductancePerVoltage (required)' = None,
    vr: 'a Nml2Quantity_voltage (required)' = None,
    vt: 'a Nml2Quantity_voltage (required)' = None,
    vpeak: 'a Nml2Quantity_voltage (required)' = None,
    a: 'a Nml2Quantity_pertime (required)' = None,
    b: 'a Nml2Quantity_conductance (required)' = None,
    c: 'a Nml2Quantity_voltage (required)' = None,
    d: 'a Nml2Quantity_current (required)' = None,
)
```

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 *(from* baseCellMembPotCap*)*	*capacitance*
EL	Leak reversal potential	*voltage*
VT	Spike threshold	*voltage*
a	Sub-threshold adaptation variable	*conductance*
b	Spike-triggered adaptation variable	*current*
delT	Slope factor	*voltage*
gL	Leak conductance	*conductance*
refract	Refractory period	*time*
reset	Reset potential	*voltage*
tauw	Adaptation time constant	*time*
thresh	Spike detection threshold	*voltage*

Exposures

iMemb	Total current crossing the cell membrane *(from* baseCellMembPotCap*)*	*current*
iSyn	Total current due to synaptic inputs *(from* baseCellMembPotCap*)*	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*
w	Adaptation current	*current*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out

Attachments

synapses	*basePointCurrent*

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) / tauw

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

d **w** /dt = (a * (v - EL) - w) / tauw

Schema

```
<xs:complexType name="AdExIaFCell">
  <xs:complexContent>
    <xs:extension base="BaseCellMembPotCap">
      <xs:attribute name="gL" type="Nml2Quantity_conductance" use="required"/>
      <xs:attribute name="EL" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="reset" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="VT" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="thresh" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="delT" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="tauw" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="refract" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="a" type="Nml2Quantity_conductance" use="required"/>
      <xs:attribute name="b" type="Nml2Quantity_current" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import AdExIaFCell
from neuroml.utils import component_factory

variable = component_factory(
    AdExIaFCell,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    C: 'a Nml2Quantity_capacitance (required)' = None,
    g_l: 'a Nml2Quantity_conductance (required)' = None,
    EL: 'a Nml2Quantity_voltage (required)' = None,
    reset: 'a Nml2Quantity_voltage (required)' = None,
    VT: 'a Nml2Quantity_voltage (required)' = None,
    thresh: 'a Nml2Quantity_voltage (required)' = None,
    del_t: 'a Nml2Quantity_voltage (required)' = None,
    tauw: 'a Nml2Quantity_time (required)' = None,
```

(continues on next page)

(continued from previous page)

```
    refract: 'a Nml2Quantity_time (required)' = None,
    a: 'a Nml2Quantity_conductance (required)' = None,
    b: 'a Nml2Quantity_current (required)' = None,
)
```

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 *(from* baseCellMembPotDL*)*	Dimensionless
W		Dimensionless

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out

Dynamics

State Variables

V: Dimensionless (exposed as **V**)

W: Dimensionless (exposed as **W**)

Time Derivatives

d **V** /dt = ((V - ((V^3) / 3)) - W + I) / SEC

d **W** /dt = (0.08 * (V + 0.7 - 0.8 * W)) / SEC

Schema

```
<xs:complexType name="FitzHughNagumoCell">
  <xs:complexContent>
    <xs:extension base="BaseCell">
      <xs:attribute name="I" type="Nml2Quantity_none" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import FitzHughNagumoCell
from neuroml.utils import component_factory

variable = component_factory(
    FitzHughNagumoCell,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    I: 'a Nml2Quantity_none (required)' = None,
)
```

Usage: XML

```
<fitzHughNagumoCell id="fn1" I="0.8"/>
```

pinskyRinzelCA3Cell

extends baseCellMembPot

Reduced CA3 cell model from Pinsky, P.F., Rinzel, J. Intrinsic and network rhythmogenesis in a reduced traub model for CA3 neurons. J Comput Neurosci 1, 39-60 (1994). See https://github.com/OpenSourceBrain/PinskyRinzelModel.

Parameters

alphac	Dimensionless
betac	Dimensionless
cm	*specificCapacitance*
eCa	*voltage*
eK	*voltage*
eL	*voltage*
eNa	*voltage*
gAmpa	*conductanceDensity*
gCa	*conductanceDensity*
gKC	*conductanceDensity*
gKahp	*conductanceDensity*
gKdr	*conductanceDensity*
gLd	*conductanceDensity*
gLs	*conductanceDensity*
gNa	*conductanceDensity*
gNmda	*conductanceDensity*
gc	*conductanceDensity*
iDend	*currentDensity*
iSoma	*currentDensity*
pp	Dimensionless
qd0	Dimensionless

Constants

MSEC = 1 ms	*time*
MVOLT = 1 mV	*voltage*
UAMP_PER_CM2 = 1 uA_per_cm2	*currentDensity*
Smax = 125.0	Dimensionless
Vsyn = 60.0 mV	*voltage*
betaqd = 0.001	Dimensionless

Exposures

Cad		Dimensionless
ICad		*currentDensity*
Si		Dimensionless
Vd	Dendritic membrane potential	*voltage*
Vs	Somatic membrane potential	*voltage*
Wi		Dimensionless
cd		Dimensionless
hs		Dimensionless
ns		Dimensionless
qd		Dimensionless
sd		Dimensionless
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	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 = gCa*sd*sd*(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 = 0.25*exp(-1.0-0.025*Vs/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 = gAmpa*Wi*(Vd-Vsyn)

Inmda = gNmda*Sisat*(Vd-Vsyn)/(1.0+0.28*exp(-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

OTHERWISE

chid = Cad/250

IF Vd < -10*MVOLT THEN

alphacd_Vd = exp((Vd/MVOLT+50.0)/11-(Vd/MVOLT+53.5)/27)/18.975

OTHERWISE

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)

OTHERWISE

betacd_Vd = 0

IF Si > Smax THEN

Sisat = Smax

OTHERWISE

Sisat = Si

Time Derivatives

d **Vs** /dt = (-gLs*(Vs-eL)-gNa*(Minfs_Vs^2)*hs*(Vs-eNa)-gKdr*ns*(Vs-eK)+(gc/pp)*(Vd-Vs)+iSoma/pp) / cm

d **Vd** /dt = (iDend/(1.0-pp)-Isyn/(1.0-pp)-gLd*(Vd-eL)-ICad-gKahp*qd*(Vd-eK)-gKC*cd*chid*(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

Schema

```
<xs:complexType name="PinskyRinzelCA3Cell">
  <xs:complexContent>
    <xs:extension base="BaseCell">
      <xs:attribute name="iSoma" type="Nml2Quantity_currentDensity" use="required"/>
      <xs:attribute name="iDend" type="Nml2Quantity_currentDensity" use="required"/>
      <xs:attribute name="gc" type="Nml2Quantity_conductanceDensity" use="required"/>
      <xs:attribute name="gLs" type="Nml2Quantity_conductanceDensity" use="required"/>
      <xs:attribute name="gLd" type="Nml2Quantity_conductanceDensity" use="required"/>
      <xs:attribute name="gNa" type="Nml2Quantity_conductanceDensity" use="required"/>
      <xs:attribute name="gKdr" type="Nml2Quantity_conductanceDensity" use="required"/
↪>
      <xs:attribute name="gCa" type="Nml2Quantity_conductanceDensity" use="required"/>
      <xs:attribute name="gKahp" type="Nml2Quantity_conductanceDensity" use="required
↪"/>
      <xs:attribute name="gKC" type="Nml2Quantity_conductanceDensity" use="required"/>
      <xs:attribute name="gNmda" type="Nml2Quantity_conductanceDensity" use="required
↪"/>
      <xs:attribute name="gAmpa" type="Nml2Quantity_conductanceDensity" use="required
↪"/>
      <xs:attribute name="eNa" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="eCa" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="eK" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="eL" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="qd0" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="pp" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="alphac" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="betac" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="cm" type="Nml2Quantity_specificCapacitance" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import PinskyRinzelCA3Cell
from neuroml.utils import component_factory

variable = component_factory(
    PinskyRinzelCA3Cell,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    i_soma: 'a Nml2Quantity_currentDensity (required)' = None,
    i_dend: 'a Nml2Quantity_currentDensity (required)' = None,
    gc: 'a Nml2Quantity_conductanceDensity (required)' = None,
    g_ls: 'a Nml2Quantity_conductanceDensity (required)' = None,
    g_ld: 'a Nml2Quantity_conductanceDensity (required)' = None,
    g_na: 'a Nml2Quantity_conductanceDensity (required)' = None,
    g_kdr: 'a Nml2Quantity_conductanceDensity (required)' = None,
    g_ca: 'a Nml2Quantity_conductanceDensity (required)' = None,
    g_kahp: 'a Nml2Quantity_conductanceDensity (required)' = None,
    g_kc: 'a Nml2Quantity_conductanceDensity (required)' = None,
    g_nmda: 'a Nml2Quantity_conductanceDensity (required)' = None,
    g_ampa: 'a Nml2Quantity_conductanceDensity (required)' = None,
    e_na: 'a Nml2Quantity_voltage (required)' = None,
    e_ca: 'a Nml2Quantity_voltage (required)' = None,
    e_k: 'a Nml2Quantity_voltage (required)' = None,
    e_l: 'a Nml2Quantity_voltage (required)' = None,
    qd0: 'a Nml2Quantity_none (required)' = None,
    pp: 'a Nml2Quantity_none (required)' = None,
    alphac: 'a Nml2Quantity_none (required)' = None,
    betac: 'a Nml2Quantity_none (required)' = None,
    cm: 'a Nml2Quantity_specificCapacitance (required)' = None,
)
```

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" gNmda="0 mS_per_cm2" gAmpa="0 mS_per_cm2"/>
```

hindmarshRose1984Cell

extends baseCellMembPotCap

The Hindmarsh Rose model is a simplified point cell model which captures complex firing patterns of single neurons, such as periodic and chaotic bursting. It has a fast spiking subsystem, which is a generalization of the FitzHugh-Nagumo system, coupled to a slower subsystem which allows the model to fire bursts. The dynamical variables x, y, z correspond to the membrane potential, a recovery variable, and a slower adaptation current, respectively. See Hindmarsh J. L., and Rose R. M. (1984) A model of neuronal bursting using three coupled first order differential equations. Proc. R. Soc. London, Ser. B 221:87–102.

Parameters

C	Total capacitance of the cell membrane *(from* baseCellMembPotCap*)*	*capacitance*
a	cubic term in x nullcline	Dimensionless
b	quadratic term in x nullcline	Dimensionless
c	constant term in y nullcline	Dimensionless
d	quadratic term in y nullcline	Dimensionless
r	timescale separation between slow and fast subsystem (r greater than 0; r much less than 1)	Dimensionless
s	related to adaptation	Dimensionless
v_scaling	scaling of x for physiological membrane potential	*voltage*
x0		Dimensionless
x1	related to the system's resting potential	Dimensionless
y0		Dimensionless
z0		Dimensionless

Constants

MSEC = 1ms	*time*

Exposures

chi		Dimensionless
iMemb	Total current crossing the cell membrane *(from* baseCellMembPotCap*)*	*current*
iSyn	Total current due to synaptic inputs *(from* baseCellMembPotCap*)*	*current*
phi		Dimensionless
rho		Dimensionless
spiking		Dimensionless
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*
x		Dimensionless
y		Dimensionless
z		Dimensionless

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out

Attachments

synapses	*basePointCurrent*

Dynamics

State Variables

v: *voltage* (exposed as **v**)

y: Dimensionless (exposed as **y**)

z: Dimensionless (exposed as **z**)

spiking: Dimensionless (exposed as **spiking**)

On Start

v = x0 * v_scaling

y = y0

z = z0

On Conditions

IF v > 0 AND spiking < 0.5 THEN

spiking = 1

EVENT OUT on port: **spike**

IF v < 0 THEN

spiking = 0

Derived Variables

iSyn = synapses[*]->i(reduce method: add) (exposed as **iSyn**)

x = v / v_scaling (exposed as **x**)

phi = y - a * x^3 + b * x^2 (exposed as **phi**)

chi = c - d * x^2 - y (exposed as **chi**)

rho = s * (x - x1) - z (exposed as **rho**)

iMemb = (C * (v_scaling * (phi - z) / MSEC)) + iSyn (exposed as **iMemb**)

Time Derivatives

d **v** /dt = iMemb/C

d **y** /dt = chi / MSEC

d **z** /dt = r * rho / MSEC

Schema

```
<xs:complexType name="HindmarshRose1984Cell">
  <xs:complexContent>
    <xs:extension base="BaseCellMembPotCap">
      <xs:attribute name="a" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="b" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="c" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="d" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="s" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="x1" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="r" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="x0" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="y0" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="z0" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="v_scaling" type="Nml2Quantity_voltage" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import HindmarshRose1984Cell
from neuroml.utils import component_factory

variable = component_factory(
    HindmarshRose1984Cell,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    C: 'a Nml2Quantity_capacitance (required)' = None,
    a: 'a Nml2Quantity_none (required)' = None,
    b: 'a Nml2Quantity_none (required)' = None,
    c: 'a Nml2Quantity_none (required)' = None,
    d: 'a Nml2Quantity_none (required)' = None,
    s: 'a Nml2Quantity_none (required)' = None,
    x1: 'a Nml2Quantity_none (required)' = None,
    r: 'a Nml2Quantity_none (required)' = None,
    x0: 'a Nml2Quantity_none (required)' = None,
    y0: 'a Nml2Quantity_none (required)' = None,
    z0: 'a Nml2Quantity_none (required)' = None,
    v_scaling: 'a Nml2Quantity_voltage (required)' = None,
)
```

13.1.5 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_v2.3.xsd. Generated on 14/08/24 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	*per_time*

Requirements

v	*voltage*

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*)*	*per_time*

Requirements

caConc		*concentration*
v	*(from* baseVoltageDepRate*)*	*voltage*

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

midpoint	*voltage*
rate	*per_time*
scale	*voltage*

Exposures

r	*(from* baseVoltageDepRate*)*	*per_time*

Requirements

v	*(from* baseVoltageDepRate*)*	*voltage*

Schema

```
<xs:complexType name="HHRate">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="type" type="NmlId" use="required"/>
      <xs:attribute name="rate" type="Nml2Quantity_pertime" use="optional"/>
      <xs:attribute name="midpoint" type="Nml2Quantity_voltage" use="optional"/>
      <xs:attribute name="scale" type="Nml2Quantity_voltage" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

HHExpRate

extends baseHHRate

Exponential form for rate equation (Q: Should these be renamed hhExpRate, etc?).

Parameters

midpoint	*(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 = rate * exp((v - midpoint)/scale) (exposed as **r**)

Schema

```
<xs:complexType name="HHRate">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="type" type="NmlId" use="required"/>
      <xs:attribute name="rate" type="Nml2Quantity_pertime" use="optional"/>
      <xs:attribute name="midpoint" type="Nml2Quantity_voltage" use="optional"/>
      <xs:attribute name="scale" type="Nml2Quantity_voltage" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

HHSigmoidRate

extends baseHHRate

Sigmoidal form for rate equation.

Parameters

midpoint	(*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 = rate / (1 + exp(0 - (v - midpoint)/scale)) (exposed as **r**)

Schema

```
<xs:complexType name="HHRate">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="type" type="NmlId" use="required"/>
      <xs:attribute name="rate" type="Nml2Quantity_pertime" use="optional"/>
      <xs:attribute name="midpoint" type="Nml2Quantity_voltage" use="optional"/>
      <xs:attribute name="scale" type="Nml2Quantity_voltage" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

HHExpLinearRate

extends baseHHRate

Exponential linear form for rate equation. Linear for large positive **v,** exponentially decays for large negative **v.**.

Parameters

midpoint	*(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 - midpoint) / scale

Conditional Derived Variables

IF x != 0 THEN

r = rate * x / (1 - exp(0 - x)) (exposed as **r**)

IF x = 0 THEN

r = rate (exposed as **r**)

Schema

```
<xs:complexType name="HHRate">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="type" type="NmlId" use="required"/>
      <xs:attribute name="rate" type="Nml2Quantity_pertime" use="optional"/>
      <xs:attribute name="midpoint" type="Nml2Quantity_voltage" use="optional"/>
      <xs:attribute name="scale" type="Nml2Quantity_voltage" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

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

midpoint	*voltage*
rate	Dimensionless
scale	*voltage*

Exposures

x	*(from* baseVoltageDepVariable*)*	Dimensionless

Requirements

v	*(from* baseVoltageDepVariable*)*	*voltage*

Schema

```
<xs:complexType name="HHVariable">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="type" type="NmlId" use="required"/>
      <xs:attribute name="rate" type="xs:float" use="optional"/>
      <xs:attribute name="midpoint" type="Nml2Quantity_voltage" use="optional"/>
      <xs:attribute name="scale" type="Nml2Quantity_voltage" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

HHExpVariable

extends baseHHVariable

Exponential form for variable equation.

Parameters

midpoint	*(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 = rate * exp((v - midpoint)/scale) (exposed as **x**)

Schema

```
<xs:complexType name="HHVariable">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="type" type="NmlId" use="required"/>
      <xs:attribute name="rate" type="xs:float" use="optional"/>
      <xs:attribute name="midpoint" type="Nml2Quantity_voltage" use="optional"/>
      <xs:attribute name="scale" type="Nml2Quantity_voltage" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

HHSigmoidVariable

extends baseHHVariable

Sigmoidal form for variable equation.

Parameters

midpoint	*(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 = rate / (1 + exp(0 - (v - midpoint)/scale)) (exposed as **x**)

Schema

```
<xs:complexType name="HHVariable">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="type" type="NmlId" use="required"/>
      <xs:attribute name="rate" type="xs:float" use="optional"/>
      <xs:attribute name="midpoint" type="Nml2Quantity_voltage" use="optional"/>
      <xs:attribute name="scale" type="Nml2Quantity_voltage" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

HHExpLinearVariable

extends baseHHVariable

Exponential linear form for variable equation. Linear for large positive **v**, exponentially decays for large negative **v**..

Parameters

midpoint	*(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 - midpoint) / scale

x = rate * a / (1 - exp(0 - a)) (exposed as **x**)

Schema

```
<xs:complexType name="HHVariable">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="type" type="NmlId" use="required"/>
      <xs:attribute name="rate" type="xs:float" use="optional"/>
      <xs:attribute name="midpoint" type="Nml2Quantity_voltage" use="optional"/>
      <xs:attribute name="scale" type="Nml2Quantity_voltage" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

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	*time*

Exposures

t	*(from* baseVoltageDepTime*)*	*time*

Requirements

v	*(from* baseVoltageDepTime*)*	*voltage*

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

temperature	*temperature*

Schema

```
<xs:complexType name="Q10Settings">
  <xs:attribute name="type" type="NmlId" use="required"/>
  <xs:attribute name="fixedQ10" type="Nml2Quantity_none" use="optional"/>
  <xs:attribute name="q10Factor" type="Nml2Quantity_none" use="optional"/>
  <xs:attribute name="experimentalTemp" type="Nml2Quantity_temperature" use="optional
↪"/>
</xs:complexType>
```

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*)*	*temperature*

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

experimentalTemp	*temperature*
q10Factor	Dimensionless

Constants

TENDEGREES = 10K	*temperature*

Exposures

q10	*(from* baseQ10Settings*)*	Dimensionless

Requirements

temperature	*(from* baseQ10Settings*)*	*temperature*

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	*temperature*

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	*temperature*
q10Factor	Dimensionless

Constants

TENDEGREES = 10K	*temperature*

Exposures

factor	*(from* baseConductanceScaling*)*	Dimensionless

Requirements

temperature	*(from* baseConductanceScaling*)*	*temperature*

Dynamics

Derived Variables

factor = q10Factor^((temperature - experimentalTemp)/TENDEGREES) (exposed as **factor**)

Schema

```
<xs:complexType name="Q10ConductanceScaling">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="q10Factor" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="experimentalTemp" type="Nml2Quantity_temperature" use=
↪"required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Q10ConductanceScaling
from neuroml.utils import component_factory

variable = component_factory(
    Q10ConductanceScaling,
    q10_factor: 'a Nml2Quantity_none (required)' = None,
    experimental_temp: 'a Nml2Quantity_temperature (required)' = None,
)
```

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		*concentration*
temperature	*(from* baseConductanceScaling*)*	*temperature*

baseGate

Base ComponentType for a voltage and/or concentration dependent gate.

Parameters

instances		Dimensionless

Child list

notes	*notes*

Exposures

fcond	Dimensionless
q	Dimensionless

Schema

```
<xs:complexType name="GateHHUndetermined">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:element name="notes" type="Notes" minOccurs="0"/>
        <xs:element name="q10Settings" type="Q10Settings" minOccurs="0"/>
        <xs:element name="forwardRate" type="HHRate" minOccurs="0"/>
        <xs:element name="reverseRate" type="HHRate" minOccurs="0"/>
        <xs:element name="timeCourse" type="HHTime" minOccurs="0"/>
        <xs:element name="steadyState" type="HHVariable" minOccurs="0"/>
        <xs:element name="subGate" type="GateFractionalSubgate" minOccurs="0"␣
↪maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="instances" type="PositiveInteger" use="required"/>
      <xs:attribute name="type" type="gateTypes" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

gate

extends baseGate

Conveniently named baseGate.

Parameters

instances	*(from baseGate)*	Dimensionless

Exposures

fcond	*(from* baseGate*)*	Dimensionless
q	*(from* baseGate*)*	Dimensionless

Schema

```
<xs:complexType name="GateHHUndetermined">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:element name="notes" type="Notes" minOccurs="0"/>
        <xs:element name="q10Settings" type="Q10Settings" minOccurs="0"/>
        <xs:element name="forwardRate" type="HHRate" minOccurs="0"/>
        <xs:element name="reverseRate" type="HHRate" minOccurs="0"/>
        <xs:element name="timeCourse" type="HHTime" minOccurs="0"/>
        <xs:element name="steadyState" type="HHVariable" minOccurs="0"/>
        <xs:element name="subGate" type="GateFractionalSubgate" minOccurs="0"␣
↪maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="instances" type="PositiveInteger" use="required"/>
      <xs:attribute name="type" type="gateTypes" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GateHHUndetermined
from neuroml.utils import component_factory

variable = component_factory(
    GateHHUndetermined,
    id: 'a NmlId (required)' = None,
    instances: 'a PositiveInteger (required)' = None,
    type: 'a gateTypes (required)' = None,
    notes: 'a string (optional)' = None,
    q10_settings: 'a Q10Settings (optional)' = None,
    forward_rate: 'a HHRate (optional)' = None,
    reverse_rate: 'a HHRate (optional)' = None,
    time_course: 'a HHTime (optional)' = None,
    steady_state: 'a HHVariable (optional)' = None,
    sub_gates: 'list of GateFractionalSubgate(s) (optional)' = None,
)
```

gateHHrates

extends *gate*

Gate which follows the general Hodgkin Huxley formalism.

Parameters

instances	*(from* baseGate*)*	Dimensionless

Child list

forwardRate	*baseVoltageDepRate*
reverseRate	*baseVoltageDepRate*

Children list

q10Settin	*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**)

tau = 1/((alpha+beta) * rateScale) (exposed as **tau**)

Time Derivatives

d **q** /dt = (inf - q) / tau

Schema

```
<xs:complexType name="GateHHRates">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:all>
        <xs:element name="notes" type="Notes" minOccurs="0"/>
        <xs:element name="q10Settings" type="Q10Settings" minOccurs="0"/>
        <xs:element name="forwardRate" type="HHRate" minOccurs="1"/>
        <xs:element name="reverseRate" type="HHRate" minOccurs="1"/>
      </xs:all>
      <xs:attribute name="instances" type="PositiveInteger" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GateHHRates
from neuroml.utils import component_factory

variable = component_factory(
    GateHHRates,
    id: 'a NmlId (required)' = None,
    instances: 'a PositiveInteger (required)' = None,
    notes: 'a string (optional)' = None,
    q10_settings: 'a Q10Settings (optional)' = None,
    forward_rate: 'a HHRate (required)' = None,
    reverse_rate: 'a HHRate (required)' = None,
)
```

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"/>
```

(continues on next page)

(continued from previous page)

```
    <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>
```

gateHHtauInf

extends *gate*

Gate which follows the general Hodgkin Huxley formalism.

Parameters

instances	*(from* baseGate*)*	Dimensionless

Child list

timeCourse	*baseVoltageDepTime*
steadyState	*baseVoltageDepVariable*

Children list

q10Settin	*baseQ10Settings*

Exposures

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**)

fcond = q^instances (exposed as **fcond**)

inf = steadyState->x (exposed as **inf**)

tauUnscaled = timeCourse->t

tau = tauUnscaled / rateScale (exposed as **tau**)

Time Derivatives
d **q** /dt = (inf - q) / tau

Schema

```
<xs:complexType name="GateHHTauInf">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:all>
        <xs:element name="notes" type="Notes" minOccurs="0"/>
        <xs:element name="q10Settings" type="Q10Settings" minOccurs="0"/>
        <xs:element name="timeCourse" type="HHTime" minOccurs="1"/>
        <xs:element name="steadyState" type="HHVariable" minOccurs="1"/>
      </xs:all>
      <xs:attribute name="instances" type="PositiveInteger" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GateHHTauInf
from neuroml.utils import component_factory

variable = component_factory(
    GateHHTauInf,
    id: 'a NmlId (required)' = None,
    instances: 'a PositiveInteger (required)' = None,
    notes: 'a string (optional)' = None,
    q10_settings: 'a Q10Settings (optional)' = None,
    time_course: 'a HHTime (required)' = None,
    steady_state: 'a HHVariable (required)' = None,
)
```

gateHHInstantaneous

extends *gate*

Gate which follows the general Hodgkin Huxley formalism but is instantaneous, so tau = 0 and gate follows exactly inf value.

Parameters

instances	*(from* baseGate*)*	Dimensionless

Constants

SEC = 1 s	*time*

Child list

steadyState	*baseVoltageDepVariable*

Exposures

fcond	*(from* baseGate*)*	Dimensionless
inf		Dimensionless
q	*(from* baseGate*)*	Dimensionless
tau		*time*

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**)

Schema

```
<xs:complexType name="GateHHInstantaneous">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:all>
        <xs:element name="notes" type="Notes" minOccurs="0"/>
        <xs:element name="steadyState" type="HHVariable" minOccurs="1"/>
      </xs:all>
      <xs:attribute name="instances" type="PositiveInteger" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GateHHInstantaneous
from neuroml.utils import component_factory

variable = component_factory(
    GateHHInstantaneous,
    id: 'a NmlId (required)' = None,
    instances: 'a PositiveInteger (required)' = None,
    notes: 'a string (optional)' = None,
    steady_state: 'a HHVariable (required)' = None,
)
```

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

q10Settin	*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

Schema

```
<xs:complexType name="GateHHRatesTau">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:all>
        <xs:element name="notes" type="Notes" minOccurs="0"/>
        <xs:element name="q10Settings" type="Q10Settings" minOccurs="0"/>
        <xs:element name="forwardRate" type="HHRate" minOccurs="1"/>
        <xs:element name="reverseRate" type="HHRate" minOccurs="1"/>
        <xs:element name="timeCourse" type="HHTime" minOccurs="1"/>
      </xs:all>
      <xs:attribute name="instances" type="PositiveInteger" use="required"/>
```

(continues on next page)

(continued from previous page)

```
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GateHHRatesTau
from neuroml.utils import component_factory

variable = component_factory(
    GateHHRatesTau,
    id: 'a NmlId (required)' = None,
    instances: 'a PositiveInteger (required)' = None,
    notes: 'a string (optional)' = None,
    q10_settings: 'a Q10Settings (optional)' = None,
    forward_rate: 'a HHRate (required)' = None,
    reverse_rate: 'a HHRate (required)' = None,
    time_course: 'a HHTime (required)' = None,
)
```

gateHHratesInf

extends *gate*

Gate which follows the general Hodgkin Huxley formalism.

Parameters

instances	*(from* baseGate*)*	Dimensionless

Child list

forwardRate	*baseVoltageDepRate*
reverseRate	*baseVoltageDepRate*
steadyState	*baseVoltageDepVariable*

Children list

q10Settin	*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 = steadyState->x (exposed as **inf**)

tau = 1/((alpha+beta) * rateScale) (exposed as **tau**)

Time Derivatives
d **q** /dt = (inf - q) / tau

Schema

```
<xs:complexType name="GateHHRatesInf">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:all>
        <xs:element name="notes" type="Notes" minOccurs="0"/>
        <xs:element name="q10Settings" type="Q10Settings" minOccurs="0"/>
        <xs:element name="forwardRate" type="HHRate" minOccurs="1"/>
        <xs:element name="reverseRate" type="HHRate" minOccurs="1"/>
        <xs:element name="steadyState" type="HHVariable" minOccurs="1"/>
      </xs:all>
      <xs:attribute name="instances" type="PositiveInteger" use="required"/>
    </xs:extension>
```

(continues on next page)

(continued from previous page)

```
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GateHHRatesInf
from neuroml.utils import component_factory

variable = component_factory(
    GateHHRatesInf,
    id: 'a NmlId (required)' = None,
    instances: 'a PositiveInteger (required)' = None,
    notes: 'a string (optional)' = None,
    q10_settings: 'a Q10Settings (optional)' = None,
    forward_rate: 'a HHRate (required)' = None,
    reverse_rate: 'a HHRate (required)' = None,
    steady_state: 'a HHVariable (required)' = None,
)
```

gateHHratesTauInf

extends *gate*

Gate which follows the general Hodgkin Huxley formalism.

Parameters

instances	*(from* baseGate*)*	Dimensionless

Child list

forwardRate	*baseVoltageDepRate*
reverseRate	*baseVoltageDepRate*
timeCourse	*baseVoltageDepTime*
steadyState	*baseVoltageDepVariable*

Children list

q10Settin	*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**)

inf = steadyState->x (exposed as **inf**)

tauUnscaled = timeCourse->t

tau = tauUnscaled / rateScale (exposed as **tau**)

fcond = q^instances (exposed as **fcond**)

Time Derivatives
d **q** /dt = (inf - q) / tau

Schema

```
<xs:complexType name="GateHHRatesTauInf">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:all>
        <xs:element name="notes" type="Notes" minOccurs="0"/>
        <xs:element name="q10Settings" type="Q10Settings" minOccurs="0"/>
        <xs:element name="forwardRate" type="HHRate" minOccurs="1"/>
        <xs:element name="reverseRate" type="HHRate" minOccurs="1"/>
        <xs:element name="timeCourse" type="HHTime" minOccurs="1"/>
        <xs:element name="steadyState" type="HHVariable" minOccurs="1"/>
      </xs:all>
```

(continues on next page)

(continued from previous page)

```
      <xs:attribute name="instances" type="PositiveInteger" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GateHHRatesTauInf
from neuroml.utils import component_factory

variable = component_factory(
    GateHHRatesTauInf,
    id: 'a NmlId (required)' = None,
    instances: 'a PositiveInteger (required)' = None,
    notes: 'a string (optional)' = None,
    q10_settings: 'a Q10Settings (optional)' = None,
    forward_rate: 'a HHRate (required)' = None,
    reverse_rate: 'a HHRate (required)' = None,
    time_course: 'a HHTime (required)' = None,
    steady_state: 'a HHVariable (required)' = None,
)
```

gateFractional

extends *gate*

Gate composed of subgates contributing with fractional conductance.

Parameters

instances	*(from* baseGate*)*	Dimensionless

Children list

q10Settin	*baseQ10Settings*
subGate	*subGate*

Exposures

fcond	*(from* baseGate*)*	Dimensionless
q	*(from* baseGate*)*	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**)

Schema

```
<xs:complexType name="GateFractional">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:element name="notes" type="Notes" minOccurs="0"/>
        <xs:element name="q10Settings" type="Q10Settings" minOccurs="0"/>
        <xs:element name="subGate" type="GateFractionalSubgate" minOccurs="1"␣
↪maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="instances" type="PositiveInteger" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GateFractional
from neuroml.utils import component_factory

variable = component_factory(
    GateFractional,
    id: 'a NmlId (required)' = None,
    instances: 'a PositiveInteger (required)' = None,
    notes: 'a string (optional)' = None,
    q10_settings: 'a Q10Settings (optional)' = None,
    sub_gates: 'list of GateFractionalSubgate(s) (required)' = None,
)
```

subGate

Gate composed of subgates contributing with fractional conductance.

Parameters

fractionalConductance	Dimensionless

Child list

notes	*notes*
timeCourse	*baseVoltageDepTime*
steadyState	*baseVoltageDepVariable*

Exposures

inf	Dimensionless
q	Dimensionless
qfrac	Dimensionless
tau	*time*

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

Schema

```
<xs:complexType name="GateFractionalSubgate">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:all>
        <xs:element name="notes" type="Notes" minOccurs="0"/>
        <xs:element name="q10Settings" type="Q10Settings" minOccurs="0"/>
        <xs:element name="steadyState" type="HHVariable" minOccurs="1"/>
        <xs:element name="timeCourse" type="HHTime" minOccurs="1"/>
      </xs:all>
      <xs:attribute name="fractionalConductance" type="Nml2Quantity_none" use=
↪"required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GateFractionalSubgate
from neuroml.utils import component_factory

variable = component_factory(
    GateFractionalSubgate,
    id: 'a NmlId (required)' = None,
    fractional_conductance: 'a Nml2Quantity_none (required)' = None,
    notes: 'a string (optional)' = None,
    q10_settings: 'a Q10Settings (optional)' = None,
    steady_state: 'a HHVariable (required)' = None,
    time_course: 'a HHTime (required)' = None,
)
```

baseIonChannel

Base for all ion channel ComponentTypes.

Parameters

conductance	*conductance*

Text fields

neuroLexId

Child list

notes	*notes*
annotation	*annotation*

Exposures

fopen	Dimensionless
g	*conductance*

Requirements

v	*voltage*

ionChannelPassive

extends *ionChannel*

Simple passive ion channel where the constant conductance through the channel is equal to **conductance**.

Parameters

conductance	*(from* baseIonChannel*)*	*conductance*

Text fields

species

Exposures

fopen	*(from* baseIonChannel*)*	Dimensionless
g	*(from* baseIonChannel*)*	*conductance*

Requirements

v	*(from* baseIonChannel*)*	*voltage*

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*)*	*conductance*

Text fields

species

Children list

conductanceScaling	*baseConductanceScaling*
gates	*gate*

Exposures

fopen	*(from* baseIonChannel*)*	Dimensionless
g	*(from* baseIonChannel*)*	*conductance*

Requirements

v	*(from* baseIonChannel*)*	*voltage*

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**)

Schema

```
<xs:complexType name="IonChannelHH">
  <xs:complexContent>
    <xs:extension base="IonChannel"/>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IonChannelHH
from neuroml.utils import component_factory

variable = component_factory(
    IonChannelHH,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    q10_conductance_scalings: 'list of Q10ConductanceScaling(s) (optional)' = None,
    species: 'a NmlId (optional)' = None,
    type: 'a channelTypes (optional)' = None,
    conductance: 'a Nml2Quantity_conductance (optional)' = None,
    gates: 'list of GateHHUndetermined(s) (optional)' = None,
    gate_hh_rates: 'list of GateHHRates(s) (optional)' = None,
    gate_h_hrates_taus: 'list of GateHHRatesTau(s) (optional)' = None,
```

(continues on next page)

(continued from previous page)

```
    gate_hh_tau_infs: 'list of GateHHTauInf(s) (optional)' = None,
    gate_h_hrates_infs: 'list of GateHHRatesInf(s) (optional)' = None,
    gate_h_hrates_tau_infs: 'list of GateHHRatesTauInf(s) (optional)' = None,
    gate_hh_instantaneouses: 'list of GateHHInstantaneous(s) (optional)' = None,
    gate_fractionals: 'list of GateFractional(s) (optional)' = None,
)
```

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
↪"/>
        <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* baseIonChannel*)*	*conductance*

Exposures

fopen	*(from* baseIonChannel*)*	Dimensionless
g	*(from* baseIonChannel*)*	*conductance*

Requirements

v	*(from* baseIonChannel*)*	*voltage*

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**)

Schema

```
<xs:complexType name="IonChannel">
  <xs:complexContent>
    <xs:extension base="IonChannelScalable">
      <xs:choice>
        <xs:element name="gate" type="GateHHUndetermined" minOccurs="0" maxOccurs=
↪"unbounded"/>
        <xs:element name="gateHHrates" type="GateHHRates" minOccurs="0" maxOccurs=
↪"unbounded"/>
        <xs:element name="gateHHratesTau" type="GateHHRatesTau" minOccurs="0"␣
↪maxOccurs="unbounded"/>
        <xs:element name="gateHHtauInf" type="GateHHTauInf" minOccurs="0" maxOccurs=
↪"unbounded"/>
        <xs:element name="gateHHratesInf" type="GateHHRatesInf" minOccurs="0"␣
↪maxOccurs="unbounded"/>
        <xs:element name="gateHHratesTauInf" type="GateHHRatesTauInf" minOccurs="0"␣
↪maxOccurs="unbounded"/>
        <xs:element name="gateHHInstantaneous" type="GateHHInstantaneous" minOccurs="0
↪" maxOccurs="unbounded"/>
        <xs:element name="gateFractional" type="GateFractional" minOccurs="0"␣
↪maxOccurs="unbounded"/>
      </xs:choice>
      <xs:attribute name="species" type="NmlId" use="optional"/>
      <xs:attribute name="type" type="channelTypes" use="optional"/>
      <xs:attribute name="conductance" type="Nml2Quantity_conductance" use="optional"/
↪>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IonChannel
from neuroml.utils import component_factory

variable = component_factory(
    IonChannel,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    q10_conductance_scalings: 'list of Q10ConductanceScaling(s) (optional)' = None,
    species: 'a NmlId (optional)' = None,
    type: 'a channelTypes (optional)' = None,
    conductance: 'a Nml2Quantity_conductance (optional)' = None,
    gates: 'list of GateHHUndetermined(s) (optional)' = None,
    gate_hh_rates: 'list of GateHHRates(s) (optional)' = None,
    gate_h_hrates_taus: 'list of GateHHRatesTau(s) (optional)' = None,
    gate_hh_tau_infs: 'list of GateHHTauInf(s) (optional)' = None,
    gate_h_hrates_infs: 'list of GateHHRatesInf(s) (optional)' = None,
    gate_h_hrates_tau_infs: 'list of GateHHRatesTauInf(s) (optional)' = None,
    gate_hh_instantaneouses: 'list of GateHHInstantaneous(s) (optional)' = None,
    gate_fractionals: 'list of GateFractional(s) (optional)' = None,
    extensiontype_=None,
)
```

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* baseIonChannel*)*	*conductance*
vShift		*voltage*

Text fields

species

Exposures

fopen	*(from* baseIonChannel*)*	Dimensionless
g	*(from* baseIonChannel*)*	*conductance*

Requirements

v	*(from* baseIonChannel*)*	*voltage*

Schema

```
<xs:complexType name="IonChannelVShift">
  <xs:complexContent>
    <xs:extension base="IonChannel">
      <xs:attribute name="vShift" type="Nml2Quantity_voltage" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IonChannelVShift
from neuroml.utils import component_factory

variable = component_factory(
    IonChannelVShift,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    q10_conductance_scalings: 'list of Q10ConductanceScaling(s) (optional)' = None,
    species: 'a NmlId (optional)' = None,
    type: 'a channelTypes (optional)' = None,
    conductance: 'a Nml2Quantity_conductance (optional)' = None,
    gates: 'list of GateHHUndetermined(s) (optional)' = None,
    gate_hh_rates: 'list of GateHHRates(s) (optional)' = None,
    gate_h_hrates_taus: 'list of GateHHRatesTau(s) (optional)' = None,
    gate_hh_tau_infs: 'list of GateHHTauInf(s) (optional)' = None,
    gate_h_hrates_infs: 'list of GateHHRatesInf(s) (optional)' = None,
```

(continues on next page)

(continued from previous page)

```
    gate_h_hrates_tau_infs: 'list of GateHHRatesTauInf(s) (optional)' = None,
    gate_hh_instantaneouses: 'list of GateHHInstantaneous(s) (optional)' = None,
    gate_fractionals: 'list of GateFractional(s) (optional)' = None,
    v_shift: 'a Nml2Quantity_voltage (required)' = None,
)
```

KSState

One of the states in which a *gateKS* can be. The rates of transitions between these states are given by *KSTransition*s.

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* KSState*)*	Dimensionless

Exposures

occupancy	*(from* KSState*)*	Dimensionless
q	*(from* KSState*)*	Dimensionless

Schema

```
<xs:complexType name="ClosedState">
  <xs:complexContent>
    <xs:extension base="Base">
      </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ClosedState
from neuroml.utils import component_factory

variable = component_factory(
    ClosedState,
    id: 'a NmlId (required)' = None,
)
```

openState

extends *KSState*

A *KSState* with **relativeConductance** of 1.

Parameters

relativeConductance	*(from* KSState*)*	Dimensionless

Exposures

occupancy	*(from* KSState*)*	Dimensionless
q	*(from* KSState*)*	Dimensionless

Schema

```
<xs:complexType name="OpenState">
  <xs:complexContent>
    <xs:extension base="Base">
      </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import OpenState
from neuroml.utils import component_factory

variable = component_factory(
    OpenState,
    id: 'a NmlId (required)' = None,
)
```

ionChannelKS

extends baseIonChannel

A kinetic scheme based ion channel with multiple *gateKS*s, each of which consists of multiple *KSState*s and *KSTransition*s giving the rates of transition between them.

Parameters

conductance	*(from* baseIonChannel*)*	*conductance*

Text fields

species

Children list

conductanceScaling	*baseConductanceScaling*
gates	*gateKS*

Exposures

fopen	*(from* baseIonChannel*)*	Dimensionless
g	*(from* baseIonChannel*)*	*conductance*

Requirements

v	*(from* baseIonChannel*)*	*voltage*

Dynamics

Derived Variables

fopen = gates[*]->fcond(reduce method: multiply) (exposed as **fopen**)

g = fopen * conductance (exposed as **g**)

Schema

```
<xs:complexType name="IonChannelKS">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:sequence>
        <xs:element name="gateKS" type="GateKS" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="species" type="NmlId" use="optional"/>
      <xs:attribute name="conductance" type="Nml2Quantity_conductance" use="optional"/
```

(continues on next page)

(continued from previous page)

```
↪>
      <xs:attribute name="neuroLexId" type="NeuroLexId" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IonChannelKS
from neuroml.utils import component_factory

variable = component_factory(
    IonChannelKS,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    species: 'a NmlId (optional)' = None,
    conductance: 'a Nml2Quantity_conductance (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    gate_kses: 'list of GateKS(s) (optional)' = None,
)
```

KSTransition

Specified the forward and reverse rates of transition between two *KSState*s in a *gateKS*.

Exposures

rf	*per_time*
rr	*per_time*

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*.

Constants

SEC = 1s	*time*

Child list

rate	*baseHHRate*

Exposures

rf	(*from* KSTransition)	*per_time*
rr	(*from* KSTransition)	*per_time*

Dynamics

Derived Variables

rf0 = rate->r

rf = rf0 (exposed as **rf**)

rr = 0/SEC (exposed as **rr**)

Schema

```
<xs:complexType name="ForwardTransition">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:any processContents="skip" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="from" type="NmlId" use="required"/>
      <xs:attribute name="to" type="NmlId" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ForwardTransition
from neuroml.utils import component_factory

variable = component_factory(
    ForwardTransition,
    id: 'a NmlId (required)' = None,
    from_: 'a NmlId (required)' = None,
```

(continues on next page)

(continued from previous page)

```
    to: 'a NmlId (required)' = None,
    anytypeobjs_=None,
)
```

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*.

Constants

SEC = 1s	*time*

Child list

rate	*baseHHRate*

Exposures

rf	(*from* KSTransition)	*per_time*
rr	(*from* KSTransition)	*per_time*

Dynamics

Derived Variables

rr0 = rate->r

rr = rr0 (exposed as **rr**)

rf = 0/SEC (exposed as **rf**)

Schema

```
<xs:complexType name="ReverseTransition">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:any processContents="skip" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="from" type="NmlId" use="required"/>
      <xs:attribute name="to" type="NmlId" use="required"/>
    </xs:extension>
```

(continues on next page)

(continued from previous page)

```
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ReverseTransition
from neuroml.utils import component_factory

variable = component_factory(
    ReverseTransition,
    id: 'a NmlId (required)' = None,
    from_: 'a NmlId (required)' = None,
    to: 'a NmlId (required)' = None,
    anytypeobjs_=None,
)
```

vHalfTransition

extends *KSTransition*

Transition which specifies both the forward and reverse rates of transition.

Parameters

gamma	Dimensionless
tau	*time*
tauMin	*time*
vHalf	*voltage*
z	Dimensionless

Constants

kte = 25.3mV	*voltage*

Exposures

rf	(*from* KSTransition)	*per_time*
rr	(*from* KSTransition)	*per_time*

Requirements

v	*voltage*

Dynamics

Derived Variables

rf0 = exp(z * gamma * (v - vHalf) / kte) / tau

rr0 = exp(-z * (1 - gamma) * (v - vHalf) / kte) / tau

rf = 1 / (1/rf0 + tauMin) (exposed as **rf**)

rr = 1 / (1/rr0 + tauMin) (exposed as **rr**)

tauInfTransition

extends *KSTransition*

KS Transition specified in terms of time constant schema:tau and steady state schema:inf.

Child list

timeCourse	*baseVoltageDepTime*
steadyState	*baseVoltageDepVariable*

Exposures

rf	*(from* KSTransition*)*	*per_time*
rr	*(from* KSTransition*)*	*per_time*

Dynamics

Derived Variables

tau = timeCourse->t

inf = steadyState->x

rf = inf/tau (exposed as **rf**)

rr = (1-inf)/tau (exposed as **rr**)

Schema

```
<xs:complexType name="TauInfTransition">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:all>
        <xs:element name="steadyState" type="HHVariable"/>
        <xs:element name="timeCourse" type="HHTime"/>
      </xs:all>
      <xs:attribute name="from" type="NmlId" use="required"/>
      <xs:attribute name="to" type="NmlId" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import TauInfTransition
from neuroml.utils import component_factory

variable = component_factory(
    TauInfTransition,
    id: 'a NmlId (required)' = None,
    from_: 'a NmlId (required)' = None,
    to: 'a NmlId (required)' = None,
    steady_state: 'a HHVariable (required)' = None,
    time_course: 'a HHTime (required)' = None,
)
```

gateKS

extends baseGate

A gate which consists of multiple *KSState*s and *KSTransition*s giving the rates of transition between them.

Parameters

instances	*(from* baseGate*)*	Dimensionless

Children list

states	*KSState*
transitions	*KSTransition*
q10Settin	*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**)

Schema

```
<xs:complexType name="GateKS">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:element name="notes" type="Notes" minOccurs="0"/>
        <xs:element name="q10Settings" type="Q10Settings" minOccurs="0"/>
        <xs:element name="closedState" type="ClosedState" minOccurs="1" maxOccurs=
↪"unbounded"/>
        <xs:element name="openState" type="OpenState" minOccurs="1" maxOccurs=
↪"unbounded"/>
        <xs:choice minOccurs="1" maxOccurs="unbounded">
          <xs:group ref="ForwardReverseTransition"/>
          <xs:element name="tauInfTransition" type="TauInfTransition"/>
        </xs:choice>
      </xs:sequence>
      <xs:attribute name="instances" type="PositiveInteger" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GateKS
from neuroml.utils import component_factory

variable = component_factory(
    GateKS,
    id: 'a NmlId (required)' = None,
    instances: 'a PositiveInteger (required)' = None,
    notes: 'a string (optional)' = None,
    q10_settings: 'a Q10Settings (optional)' = None,
    closed_states: 'list of ClosedState(s) (required)' = None,
    open_states: 'list of OpenState(s) (required)' = None,
    forward_transition: 'list of ForwardTransition(s) (required)' = None,
    reverse_transition: 'list of ReverseTransition(s) (required)' = None,
    tau_inf_transition: 'list of TauInfTransition(s) (required)' = None,
)
```

13.1.6 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_v2.3.xsd. Generated on 14/08/24 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 *(from current basePointCurrent)*

Event Ports

in	Direction: in

Schema

```
<xs:complexType name="BaseSynapse">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="neuroLexId" type="NeuroLexId" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BaseSynapse
from neuroml.utils import component_factory

variable = component_factory(
    BaseSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    extensiontype_=None,
)
```

baseVoltageDepSynapse

extends baseSynapse

Base type for synapses with a dependence on membrane potential.

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from current basePointCurrent)*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed	*voltage*

Event Ports

in	*(from* baseSynapse*)*	Direction: in

Schema

```
<xs:complexType name="BaseVoltageDepSynapse">
  <xs:complexContent>
    <xs:extension base="BaseSynapse">

            </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BaseVoltageDepSynapse
from neuroml.utils import component_factory

variable = component_factory(
    BaseVoltageDepSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    extensiontype_=None,
)
```

baseSynapseDL

extends baseVoltageDepPointCurrentDL

Base type for all synapses, i.e. ComponentTypes which produce a dimensionless current and change Dynamics in response to an incoming event.

Bioportal entry for Computational Neuroscience Ontology related to baseSynapseDL.

Exposures

I	The total (time varying) current produced by this ComponentType *(from* basePointCurrentDL*)*	Dimensionless

Requirements

V	The current may vary with the dimensionless voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepPointCurrentDL*)*	Dimensionless

baseCurrentBasedSynapse

extends baseSynapse

Synapse model which produces a synaptic current.

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Event Ports

in	*(from* baseSynapse*)*	Direction: in

Schema

```
<xs:complexType name="BaseCurrentBasedSynapse">
  <xs:complexContent>
    <xs:extension base="BaseSynapse">

            </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BaseCurrentBasedSynapse
from neuroml.utils import component_factory

variable = component_factory(
    BaseCurrentBasedSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    extensiontype_=None,
)
```

alphaCurrentSynapse

extends baseCurrentBasedSynapse

Alpha current synapse: rise time and decay time are both **tau.**.

Parameters

ibase	Baseline current increase after receiving a spike	*current*
tau	Time course for rise and decay	*time*

Properties

weight (default: 1)	Dimensionless

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Event Ports

in	*(from* baseSynapse*)*	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

Schema

```
<xs:complexType name="AlphaCurrentSynapse">
  <xs:complexContent>
    <xs:extension base="BaseCurrentBasedSynapse">
      <xs:attribute name="tau" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="ibase" type="Nml2Quantity_current" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import AlphaCurrentSynapse
from neuroml.utils import component_factory

variable = component_factory(
    AlphaCurrentSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
```

(continues on next page)

(continued from previous page)

```
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    tau: 'a Nml2Quantity_time (required)' = None,
    ibase: 'a Nml2Quantity_current (required)' = None,
)
```

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	*voltage*
gbase	Baseline conductance, generally the maximum conductance following a single spike	*conductance*

Exposures

g	Time varying conductance through the synapse	*conductance*
i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepSynapse*)*	*voltage*

Event Ports

in	*(from* baseSynapse*)*	Direction: in

Schema

```
<xs:complexType name="BaseConductanceBasedSynapse">
  <xs:complexContent>
    <xs:extension base="BaseVoltageDepSynapse">
      <xs:attribute name="gbase" type="Nml2Quantity_conductance" use="required"/>
      <xs:attribute name="erev" type="Nml2Quantity_voltage" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BaseConductanceBasedSynapse
from neuroml.utils import component_factory

variable = component_factory(
    BaseConductanceBasedSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    gbase: 'a Nml2Quantity_conductance (required)' = None,
    erev: 'a Nml2Quantity_voltage (required)' = None,
    extensiontype_=None,
)
```

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!!

Bioportal entry for Computational Neuroscience Ontology related to baseConductanceBasedSynapseTwo.

Parameters

erev	Reversal potential of the synapse	*voltage*
gbase1	Baseline conductance 1	*conductance*
gbase2	Baseline conductance 2	*conductance*

Exposures

g	Time varying conductance through the synapse	*conductance*
i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepSynapse*)*	*voltage*

Event Ports

in	*(from* baseSynapse*)*	Direction: in

Schema

```
<xs:complexType name="BaseConductanceBasedSynapseTwo">
  <xs:complexContent>
    <xs:extension base="BaseVoltageDepSynapse">
      <xs:attribute name="gbase1" type="Nml2Quantity_conductance" use="required"/>
      <xs:attribute name="gbase2" type="Nml2Quantity_conductance" use="required"/>
      <xs:attribute name="erev" type="Nml2Quantity_voltage" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BaseConductanceBasedSynapseTwo
from neuroml.utils import component_factory

variable = component_factory(
    BaseConductanceBasedSynapseTwo,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    gbase1: 'a Nml2Quantity_conductance (required)' = None,
    gbase2: 'a Nml2Quantity_conductance (required)' = None,
    erev: 'a Nml2Quantity_voltage (required)' = None,
    extensiontype_=None,
)
```

expOneSynapse

extends 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 *(from* baseConductanceBasedSynapse*)*	*voltage*
gbase	Baseline conductance, generally the maximum conductance following a single spike *(from* baseConductanceBasedSynapse*)*	*conductance*
tauDecay	Time course of decay	*time*

Properties

weight (default: 1)	Dimensionless

Exposures

g	Time varying conductance through the synapse *(from* baseConductanceBasedSynapse*)*	*conductance*
i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepSynapse*)*	*voltage*

Event Ports

in	*(from* baseSynapse*)*	Direction: in

Dynamics

State Variables
g: *conductance* (exposed as **g**)

On Start
g = 0

On Events
EVENT IN on port: **in**

g = g + (weight * gbase)

Derived Variables
i = g * (erev - v) (exposed as **i**)

Time Derivatives
d **g** /dt = -g / tauDecay

Schema

```
<xs:complexType name="ExpOneSynapse">
  <xs:complexContent>
    <xs:extension base="BaseConductanceBasedSynapse">
      <xs:attribute name="tauDecay" type="Nml2Quantity_time" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ExpOneSynapse
from neuroml.utils import component_factory

variable = component_factory(
    ExpOneSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    gbase: 'a Nml2Quantity_conductance (required)' = None,
    erev: 'a Nml2Quantity_voltage (required)' = None,
    tau_decay: 'a Nml2Quantity_time (required)' = None,
)
```

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 *(from* baseConductanceBasedSynapse*)*	*voltage*
gbase	Baseline conductance, generally the maximum conductance following a single spike *(from* baseConductanceBasedSynapse*)*	*conductance*
tau	Time course of rise/decay	*time*

Properties

weight (default: 1)	Dimensionless

Exposures

g	Time varying conductance through the synapse *(from* baseConductanceBasedSynapse*)*	*conductance*
i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepSynapse*)*	*voltage*

Event Ports

in	*(from* 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

Schema

```
<xs:complexType name="AlphaSynapse">
  <xs:complexContent>
    <xs:extension base="BaseConductanceBasedSynapse">
      <xs:attribute name="tau" type="Nml2Quantity_time" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import AlphaSynapse
from neuroml.utils import component_factory

variable = component_factory(
    AlphaSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
```

(continues on next page)

(continued from previous page)

```
    gbase: 'a Nml2Quantity_conductance (required)' = None,
    erev: 'a Nml2Quantity_voltage (required)' = None,
    tau: 'a Nml2Quantity_time (required)' = None,
)
```

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

erev	Reversal potential of the synapse *(from* baseConductanceBasedSynapse*)*	*voltage*
gbase	Baseline conductance, generally the maximum conductance following a single spike *(from* baseConductanceBasedSynapse*)*	*conductance*
tauDecay		*time*
tauRise		*time*

Derived parameters

peakTime		*time*

peakTime = log(tauDecay / tauRise) * (tauRise * tauDecay)/(tauDecay - tauRise)

waveformFactor		Dimensionless

waveformFactor = 1 / (-exp(-peakTime / tauRise) + exp(-peakTime / tauDecay))

Properties

weight (default: 1)	Dimensionless

Exposures

g	Time varying conductance through the synapse *(from* baseConductanceBasedSynapse*)*	*conductance*
i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepSynapse*)*	*voltage*

Event Ports

in	*(from* baseSynapse*)*	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 **A** /dt = -A / tauRise

d **B** /dt = -B / tauDecay

Schema

```
<xs:complexType name="ExpTwoSynapse">
  <xs:complexContent>
    <xs:extension base="BaseConductanceBasedSynapse">
      <xs:attribute name="tauDecay" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="tauRise" type="Nml2Quantity_time" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ExpTwoSynapse
from neuroml.utils import component_factory

variable = component_factory(
    ExpTwoSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    gbase: 'a Nml2Quantity_conductance (required)' = None,
    erev: 'a Nml2Quantity_voltage (required)' = None,
    tau_decay: 'a Nml2Quantity_time (required)' = None,
    tau_rise: 'a Nml2Quantity_time (required)' = None,
    extensiontype_=None,
)
```

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 *(from* baseConductanceBasedSynapseTwo*)*	*voltage*
gbase1	Baseline conductance 1 *(from* baseConductanceBasedSynapseTwo*)*	*conductance*
gbase2	Baseline conductance 2 *(from* baseConductanceBasedSynapseTwo*)*	*conductance*
tauDecay1		*time*
tauDecay2		*time*
tauRise		*time*

Derived parameters

peakTime1	*time*

peakTime1 = log(tauDecay1 / tauRise) * (tauRise * tauDecay1)/(tauDecay1 - tauRise)

peakTime2	*time*

peakTime2 = log(tauDecay2 / tauRise) * (tauRise * tauDecay2)/(tauDecay2 - tauRise)

waveformFactor1	Dimensionless

waveformFactor1 = 1 / (-exp(-peakTime1 / tauRise) + exp(-peakTime1 / tauDecay1))

waveformFactor2	Dimensionless

waveformFactor2 = 1 / (-exp(-peakTime2 / tauRise) + exp(-peakTime2 / tauDecay2))

Properties

weight (default: 1)	Dimensionless

Exposures

g	Time varying conductance through the synapse *(from* baseConductanceBasedSynapseTwo*)*	*conductance*
i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepSynapse*)*	*voltage*

Event Ports

in	*(from* baseSynapse*)*	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 + (gbase1*weight * waveformFactor1 + gbase2*weight*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 **A** /dt = -A / tauRise

d **B** /dt = -B / tauDecay1

d **C** /dt = -C / tauDecay2

Schema

```
<xs:complexType name="ExpThreeSynapse">
  <xs:complexContent>
    <xs:extension base="BaseConductanceBasedSynapseTwo">
      <xs:attribute name="tauDecay1" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="tauDecay2" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="tauRise" type="Nml2Quantity_time" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ExpThreeSynapse
from neuroml.utils import component_factory

variable = component_factory(
    ExpThreeSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    gbase1: 'a Nml2Quantity_conductance (required)' = None,
    gbase2: 'a Nml2Quantity_conductance (required)' = None,
    erev: 'a Nml2Quantity_voltage (required)' = None,
    tau_decay1: 'a Nml2Quantity_time (required)' = None,
    tau_decay2: 'a Nml2Quantity_time (required)' = None,
    tau_rise: 'a Nml2Quantity_time (required)' = None,
)
```

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

blockFactor	Dimensionless

voltageConcDepBlockMechanism

extends baseBlockMechanism

Synaptic blocking mechanism which varys with membrane potential across the synapse, e.g. in NMDA receptor mediated synapses.

Parameters

blockConcentration	*concentration*
scalingConc	*concentration*
scalingVolt	*voltage*

Text fields

species

Exposures

blockFactor	*(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	*time*

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**

R = R * (1 - U)

Derived Variables

U = initReleaseProb

plasticityFactor = R * U (exposed as **plasticityFactor**)

Time Derivatives

d **R** /dt = (1 - R) / tauRec

tsodyksMarkramDepFacMechanism

extends basePlasticityMechanism

Full Tsodyks-Markram STP model with both depression and facilitation, as in Tsodyks, Pawelzik and Markram 1998.

Parameters

initReleaseProb	Dimensionless
tauFac	*time*
tauRec	*time*

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 **R** /dt = (1 - R) / tauRec

d **U** /dt = (initReleaseProb - U) / tauFac

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*)*	*voltage*
gbase	Baseline conductance, generally the maximum conductance following a single spike *(from* baseConductanceBasedSynapse*)*	*conductance*
tauDecay	*(from* expTwoSynapse*)*	*time*
tauRise	*(from* expTwoSynapse*)*	*time*

Derived parameters

peakTime	*(from* expTwoSynapse*)*	*time*

peakTime = log(tauDecay / tauRise) * (tauRise * tauDecay)/(tauDecay - tauRise)

waveformFactor	*(from* expTwoSynapse*)*	Dimensionless

waveformFactor = 1 / (-exp(-peakTime / tauRise) + exp(-peakTime / tauDecay))

Children list

plasticityMechanisms	*basePlasticityMechanism*
blockMechanisms	*baseBlockMechanism*

Properties

weight (default: 1)	Dimensionless

Exposures

g	Time varying conductance through the synapse *(from* baseConductanceBasedSynapse*)*	*conductance*
i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepSynapse*)*	*voltage*

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 / tauRise

d **B** /dt = -B / tauDecay

Schema

```
<xs:complexType name="BlockingPlasticSynapse">
  <xs:complexContent>
    <xs:extension base="ExpTwoSynapse">
      <xs:sequence>
        <xs:element name="plasticityMechanism" type="PlasticityMechanism" minOccurs="0
↪"/>
        <xs:element name="blockMechanism" type="BlockMechanism" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BlockingPlasticSynapse
from neuroml.utils import component_factory

variable = component_factory(
    BlockingPlasticSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    gbase: 'a Nml2Quantity_conductance (required)' = None,
    erev: 'a Nml2Quantity_voltage (required)' = None,
    tau_decay: 'a Nml2Quantity_time (required)' = None,
    tau_rise: 'a Nml2Quantity_time (required)' = None,
    plasticity_mechanism: 'a PlasticityMechanism (optional)' = None,
    block_mechanism: 'a BlockMechanism (optional)' = None,
)
```

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

synapse1Pat	
synapse2Pat	

Component References

synapse1	*baseSynapse*
synapse2	*baseSynapse*

Properties

weight (default: 1)	Dimensionless

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepSynapse*)*	*voltage*

Event Ports

in	*(from* baseSynapse*)*	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**)

Schema

```
<xs:complexType name="DoubleSynapse">
  <xs:complexContent>
    <xs:extension base="BaseVoltageDepSynapse">
      <xs:attribute name="synapse1" type="NmlId" use="required"/>
      <xs:attribute name="synapse2" type="NmlId" use="required"/>
      <xs:attribute name="synapse1Path" type="xs:string" use="required"/>
      <xs:attribute name="synapse2Path" type="xs:string" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import DoubleSynapse
from neuroml.utils import component_factory

variable = component_factory(
    DoubleSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    synapse1: 'a NmlId (required)' = None,
    synapse2: 'a NmlId (required)' = None,
    synapse1_path: 'a string (required)' = None,
    synapse2_path: 'a string (required)' = None,
)
```

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 *(from* baseConductanceBasedSynapse*)*	*voltage*
gbase	Baseline conductance, generally the maximum conductance following a single spike *(from* baseConductanceBasedSynapse*)*	*conductance*
tauDecay	*(from* expTwoSynapse*)*	*time*
tauRise	*(from* expTwoSynapse*)*	*time*

Constants

tsinceRate = 1	Dimensionless
longTime = 1000s	*time*

Derived parameters

peakTime	*(from* expTwoSynapse*)*	*time*

peakTime = log(tauDecay / tauRise) * (tauRise * tauDecay)/(tauDecay - tauRise)

waveformFactor	*(from* expTwoSynapse*)*	Dimensionless

waveformFactor = 1 / (-exp(-peakTime / tauRise) + exp(-peakTime / tauDecay))

Exposures

M		Dimensionless
P		Dimensionless
g	Time varying conductance through the synapse *(from* baseConductanceBasedSynapse*)*	*conductance*
i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*
tsince		*time*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepSynapse*)*	*voltage*

Event Ports

in	*(from* baseSynapse*)*	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 **tsince** /dt = tsinceRate

gapJunction

extends baseSynapse

Gap junction/single electrical connection.

Parameters

conductance	*conductance*

Properties

weight (default: 1)	Dimensionless

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	*voltage*

Event Ports

in	*(from* baseSynapse*)*	Direction: in

Dynamics

Derived Variables

vpeer = peer->v

i = weight * conductance * (vpeer - v) (exposed as **i**)

Schema

```
<xs:complexType name="GapJunction">
  <xs:complexContent>
    <xs:extension base="BaseSynapse">
      <xs:attribute name="conductance" type="Nml2Quantity_conductance" use="required"/
↪>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GapJunction
from neuroml.utils import component_factory

variable = component_factory(
    GapJunction,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    conductance: 'a Nml2Quantity_conductance (required)' = None,
)
```

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 *(from current* basePointCurrent*)*

Event Ports

in	*(from* baseSynapse*)*	Direction: in

silentSynapse

extends baseGradedSynapse

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

Constants

AMP = 1A	*current*

Properties

weight (default: 1)	Dimensionless

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	*voltage*

Event Ports

in	*(from* baseSynapse*)*	Direction: in

Dynamics

Derived Variables

vpeer = peer->v

i = 0 * AMP (exposed as **i**)

Schema

```
<xs:complexType name="SilentSynapse">
  <xs:complexContent>
    <xs:extension base="BaseSynapse">

            </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SilentSynapse
from neuroml.utils import component_factory

variable = component_factory(
    SilentSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
)
```

Usage: XML

```
<silentSynapse id="silent1"/>
```

```
<silentSynapse id="silent2"/>
```

```
<silentSynapse id="silent1"/>
```

linearGradedSynapse

extends baseGradedSynapse

Behaves just like a one way gap junction.

Parameters

conductance	*conductance*

Properties

weight (default: 1)	Dimensionless

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	*voltage*

Event Ports

in	*(from* baseSynapse*)*	Direction: in

Dynamics

Derived Variables

vpeer = peer->v

i = weight * conductance * (vpeer - v) (exposed as **i**)

Schema

```
<xs:complexType name="LinearGradedSynapse">
  <xs:complexContent>
    <xs:extension base="BaseSynapse">
      <xs:attribute name="conductance" type="Nml2Quantity_conductance" use="required"/
↪>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import LinearGradedSynapse
from neuroml.utils import component_factory

variable = component_factory(
    LinearGradedSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    conductance: 'a Nml2Quantity_conductance (required)' = None,
)
```

Usage: XML

```
<linearGradedSynapse 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	*voltage*
conductance		*conductance*
delta	Slope of the activation curve	*voltage*
erev	The reversal potential of the synapse	*voltage*
k	Rate constant for transmitter-receptor dissociation rate	*per_time*

Properties

weight (default: 1)	Dimensionless

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*
inf		Dimensionless
tau		*time*

Requirements

v	*voltage*

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((Vth - vpeer)/delta)) (exposed as **inf**)

tau = (1-inf)/k (exposed as **tau**)

i = weight * conductance * s * (erev-v) (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

Schema

```
<xs:complexType name="GradedSynapse">
  <xs:complexContent>
    <xs:extension base="BaseSynapse">
      <xs:attribute name="conductance" type="Nml2Quantity_conductance" use="required"/
 ↪>
      <xs:attribute name="delta" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="Vth" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="k" type="Nml2Quantity_pertime" use="required"/>
      <xs:attribute name="erev" type="Nml2Quantity_voltage" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import GradedSynapse
from neuroml.utils import component_factory

variable = component_factory(
    GradedSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    conductance: 'a Nml2Quantity_conductance (required)' = None,
    delta: 'a Nml2Quantity_voltage (required)' = None,
    Vth: 'a Nml2Quantity_voltage (required)' = None,
    k: 'a Nml2Quantity_pertime (required)' = None,
    erev: 'a Nml2Quantity_voltage (required)' = None,
)
```

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"/>
```

13.1.7 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_v2.3.xsd. Generated on 14/08/24 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	*current*

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 *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed	*voltage*

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 *(from* basePointCurrent*)*	*current*
tsince	Time since the last spike was emitted	*time*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepPointCurrent*)*	*voltage*

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 *(from* basePointCurrentDL*)*	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	*time*

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.	*time*

Constants

SMALL_TIME = 1e-9ms	A useful constant for use as a non zero time increment	*time*

Exposures

tnext	When the next spike should ideally be emitted (dt permitting)	*time*
tsince	Time since the last spike was emitted *(from* baseSpikeSource*)*	*time*

Event Ports

spike	Port on which spikes are emitted *(from* baseSpikeSource*)*	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 **tsince** /dt = 1

d **tnext** /dt = 0

Schema

```
<xs:complexType name="SpikeGenerator">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="period" type="Nml2Quantity_time" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SpikeGenerator
from neuroml.utils import component_factory

variable = component_factory(
    SpikeGenerator,
    id: 'a NonNegativeInteger (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    period: 'a Nml2Quantity_time (required)' = None,
)
```

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	*time*
minISI	Minimum interspike interval	*time*

Constants

MSEC = 1ms	Required for converting time values to/from dimensionless quantities	*time*

Exposures

isi	The interval until the next spike	*time*
tnext	When the next spike should ideally be emitted (dt permitting)	*time*
tsince	Time since the last spike was emitted *(from* baseSpikeSource*)*	*time*

Event Ports

spike	Port on which spikes are emitted *(from* baseSpikeSource*)*	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

Schema

```
<xs:complexType name="SpikeGeneratorRandom">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="maxISI" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="minISI" type="Nml2Quantity_time" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SpikeGeneratorRandom
from neuroml.utils import component_factory

variable = component_factory(
    SpikeGeneratorRandom,
    id: 'a NonNegativeInteger (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    max_isi: 'a Nml2Quantity_time (required)' = None,
    min_isi: 'a Nml2Quantity_time (required)' = None,
)
```

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

averageRate	The average rate at which spikes are emitted	*per_time*

Constants

SMALL_TIME = 1e-9ms	*time*

Exposures

isi	The interval until the next spike	*time*
tnextIdeal	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	*time*
tnextUsed	This is the next spike time for practical purposes, ensuring that it's later than the current time	*time*
tsince	Time since the last spike was emitted *(from* baseSpikeSource*)*	*time*

Event Ports

spike	Port on which spikes are emitted *(from* baseSpikeSource*)*	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

Schema

```
<xs:complexType name="SpikeGeneratorPoisson">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="averageRate" type="Nml2Quantity_pertime" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SpikeGeneratorPoisson
from neuroml.utils import component_factory

variable = component_factory(
    SpikeGeneratorPoisson,
    id: 'a NonNegativeInteger (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    average_rate: 'a Nml2Quantity_pertime (required)' = None,
    extensiontype_=None,
)
```

Usage: XML

```
<spikeGeneratorPoisson id="spikeGenPoisson" averageRate="50 Hz"/>
```

spikeGeneratorRefPoisson

extends *spikeGeneratorPoisson*

Generator of spikes whose ISI distribution is the maximum entropy distribution over [**minimumISI,** +infinity) with mean: 1 / **averageRate**.

Parameters

averageRate	The average rate at which spikes are emitted *(from* spikeGeneratorPoisson*)*	*per_time*
minimumISI	The minimum interspike interval	*time*

Derived parameters

averageIsi	The average interspike interval	*time*

averageIsi = 1 / averageRate

Exposures

isi	The interval until the next spike *(from* spikeGeneratorPoisson*)*	*time*
tnextIdeal	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 *(from* spikeGeneratorPoisson*)*	*time*
tnextUsed	This is the next spike time for practical purposes, ensuring that it's later than the current time *(from* spikeGeneratorPoisson*)*	*time*
tsince	Time since the last spike was emitted *(from* baseSpikeSource*)*	*time*

Event Ports

spike	Port on which spikes are emitted *(from* baseSpikeSource*)*	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

Schema

```
<xs:complexType name="SpikeGeneratorRefPoisson">
  <xs:complexContent>
    <xs:extension base="SpikeGeneratorPoisson">
      <xs:attribute name="minimumISI" type="Nml2Quantity_time" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SpikeGeneratorRefPoisson
from neuroml.utils import component_factory

variable = component_factory(
    SpikeGeneratorRefPoisson,
    id: 'a NonNegativeInteger (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    average_rate: 'a Nml2Quantity_pertime (required)' = None,
    minimum_isi: 'a Nml2Quantity_time (required)' = None,
)
```

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

averageRate	The average rate at which spikes are emitted	*per_time*

Constants

SMALL_TIME = 1e-9ms	*time*

Derived parameters

averageIsi	The average interspike interval	*time*

averageIsi = 1 / averageRate

Paths

spikeTarget	The target of the spikes, i.e. the synapse

Component References

synapse	*baseSynapse*

Properties

weight (default: 1)	Dimensionless

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*
isi	The interval until the next spike	*time*
tnextIdeal		*time*
tnextUsed		*time*
tsince	Time since the last spike was emitted *(from* baseVoltageDepPointCurrentSpiking*)*	*time*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepPointCurrent*)*	*voltage*

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 *(from* baseVoltageDepPointCurrentSpiking*)*	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 **tsince** /dt = 1

d **tnextUsed** /dt = 0

d **tnextIdeal** /dt = 0

Schema

```
<xs:complexType name="PoissonFiringSynapse">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="averageRate" type="Nml2Quantity_pertime" use="required"/>
      <xs:attribute name="synapse" type="xs:string" use="required"/>
      <xs:attribute name="spikeTarget" type="xs:string" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import PoissonFiringSynapse
from neuroml.utils import component_factory

variable = component_factory(
    PoissonFiringSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    average_rate: 'a Nml2Quantity_pertime (required)' = None,
    synapse: 'a string (required)' = None,
    spike_target: 'a string (required)' = None,
)
```

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

averageRate	*per_time*
delay	*time*
duration	*time*

Constants

SMALL_TIME = 1e-9ms	*time*
LONG_TIME = 1e9hour	*time*

Derived parameters

averageIsi	*time*

averageIsi = 1 / averageRate

Paths

spikeTarget

Component References

synapse	*baseSynapse*

Properties

weight (default: 1)	Dimensionless

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*
isi		*time*
tnextIdeal		*time*
tnextUsed		*time*
tsince	Time since the last spike was emitted *(from* baseVoltageDepPointCurrentSpiking*)*	*time*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepPointCurrent*)*	*voltage*

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 *(from* baseVoltageDepPointCurrentSpiking*)*	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 **tsince** /dt = 1

d **tnextUsed** /dt = 0

d **tnextIdeal** /dt = 0

Schema

```
<xs:complexType name="TransientPoissonFiringSynapse">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="averageRate" type="Nml2Quantity_pertime" use="required"/>
      <xs:attribute name="delay" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="duration" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="synapse" type="xs:string" use="required"/>
      <xs:attribute name="spikeTarget" type="xs:string" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import TransientPoissonFiringSynapse
from neuroml.utils import component_factory

variable = component_factory(
    TransientPoissonFiringSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
```

(continues on next page)

(continued from previous page)

```
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    average_rate: 'a Nml2Quantity_pertime (required)' = None,
    delay: 'a Nml2Quantity_time (required)' = None,
    duration: 'a Nml2Quantity_time (required)' = None,
    synapse: 'a string (required)' = None,
    spike_target: 'a string (required)' = None,
)
```

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

spikeTarget

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 *(from* basePointCurrent*)*	*current*
tsince	Time since the last spike was emitted *(from* baseVoltageDepPointCurrentSpiking*)*	*time*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepPointCurrent*)*	*voltage*

Event Ports

in	This will receive events from the children	Direction: in
spike	Port on which spikes are emitted *(from* baseVoltageDepPointCurrentSpiking*)*	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

Schema

```
<xs:complexType name="TimedSynapticInput">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:sequence>
        <xs:element name="spike" type="Spike" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="synapse" type="NmlId" use="required"/>
      <xs:attribute name="spikeTarget" type="xs:string" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import TimedSynapticInput
from neuroml.utils import component_factory

variable = component_factory(
    TimedSynapticInput,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    synapse: 'a NmlId (required)' = None,
    spike_target: 'a string (required)' = None,
    spikes: 'list of Spike(s) (optional)' = None,
)
```

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	*spike*

Exposures

tsince	Time since the last spike was emitted *(from* baseSpikeSource*)*	*time*

Event Ports

in	This will receive events from the children	Direction: in
spike	Port on which spikes are emitted *(from* baseSpikeSource*)*	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

Schema

```
<xs:complexType name="SpikeArray">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:sequence>
        <xs:element name="spike" type="Spike" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SpikeArray
from neuroml.utils import component_factory

variable = component_factory(
    SpikeArray,
    id: 'a NonNegativeInteger (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    spikes: 'list of Spike(s) (optional)' = None,
)
```

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>
```

spike

extends baseSpikeSource

Emits a single spike at the specified **time**.

Parameters

time	Time at which to emit one spike event	*time*

Exposures

spiked	0 signals not yet spiked, 1 signals has spiked	Dimensionless
tsince	Time since the last spike was emitted *(from* baseSpikeSource*)*	*time*

Event Ports

spike	Port on which spikes are emitted *(from* baseSpikeSource*)*	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

Schema

```
<xs:complexType name="Spike">
  <xs:complexContent>
    <xs:extension base="BaseNonNegativeIntegerId">
      <xs:attribute name="time" type="Nml2Quantity_time" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Spike
from neuroml.utils import component_factory

variable = component_factory(
    Spike,
    id: 'a NonNegativeInteger (required)' = None,
    time: 'a Nml2Quantity_time (required)' = None,
)
```

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	*current*
delay	Delay before change in current. Current is zero prior to this.	*time*
duration	Duration for holding current at amplitude. Current is zero after delay + duration.	*time*

Properties

weight (default: 1)	Dimensionless

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

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 AND t < duration + delay THEN

i = weight * amplitude

IF t >= duration + delay THEN

i = 0

Schema

```
<xs:complexType name="PulseGenerator">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="delay" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="duration" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="amplitude" type="Nml2Quantity_current" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import PulseGenerator
from neuroml.utils import component_factory

variable = component_factory(
    PulseGenerator,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    delay: 'a Nml2Quantity_time (required)' = None,
    duration: 'a Nml2Quantity_time (required)' = None,
    amplitude: 'a Nml2Quantity_current (required)' = None,
)
```

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 *(from* 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**)

Schema

```
<xs:complexType name="CompoundInput">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:sequence>
        <xs:element name="pulseGenerator" type="PulseGenerator" minOccurs="0"␣
↪maxOccurs="unbounded"/>
        <xs:element name="sineGenerator" type="SineGenerator" minOccurs="0" maxOccurs=
↪"unbounded"/>
        <xs:element name="rampGenerator" type="RampGenerator" minOccurs="0" maxOccurs=
↪"unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import CompoundInput
from neuroml.utils import component_factory

variable = component_factory(
    CompoundInput,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    pulse_generators: 'list of PulseGenerator(s) (optional)' = None,
    sine_generators: 'list of SineGenerator(s) (optional)' = None,
    ramp_generators: 'list of RampGenerator(s) (optional)' = None,
)
```

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>
```

compoundInputDL

extends 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

currents	*basePointCurrentDL*

Properties

weight (default: 1)	Dimensionless

Exposures

I	The total (time varying) current produced by this ComponentType *(from* basePointCurrentDL*)*	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**)

Schema

```
<xs:complexType name="CompoundInputDL">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:sequence>
        <xs:element name="pulseGeneratorDL" type="PulseGeneratorDL" minOccurs="0"␣
↪maxOccurs="unbounded"/>
        <xs:element name="sineGeneratorDL" type="SineGeneratorDL" minOccurs="0"␣
```

(continues on next page)

(continued from previous page)

```
↪maxOccurs="unbounded"/>
       <xs:element name="rampGeneratorDL" type="RampGeneratorDL" minOccurs="0"␣
↪maxOccurs="unbounded"/>
     </xs:sequence>
   </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import CompoundInputDL
from neuroml.utils import component_factory

variable = component_factory(
    CompoundInputDL,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    pulse_generator_dls: 'list of PulseGeneratorDL(s) (optional)' = None,
    sine_generator_dls: 'list of SineGeneratorDL(s) (optional)' = None,
    ramp_generator_dls: 'list of RampGeneratorDL(s) (optional)' = None,
)
```

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

amplitude	Amplitude of current pulse	Dimensionless
delay	Delay before change in current. Current is zero prior to this.	*time*
duration	Duration for holding current at amplitude. Current is zero after delay + duration.	*time*

Properties

weight (default: 1)	Dimensionless

Exposures

I	The total (time varying) current produced by this ComponentType *(from* basePointCurrentDL*)*	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 < delay THEN

I = 0

IF t >= delay AND t < duration + delay THEN

I = weight * amplitude

IF t >= duration + delay THEN

I = 0

Schema

```
<xs:complexType name="PulseGeneratorDL">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="delay" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="duration" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="amplitude" type="Nml2Quantity_none" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import PulseGeneratorDL
from neuroml.utils import component_factory

variable = component_factory(
    PulseGeneratorDL,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    delay: 'a Nml2Quantity_time (required)' = None,
    duration: 'a Nml2Quantity_time (required)' = None,
    amplitude: 'a Nml2Quantity_current (required)' = None,
)
```

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

amplitude	Maximum amplitude of current	*current*
delay	Delay before change in current. Current is zero prior to this.	*time*
duration	Duration for holding current at amplitude. Current is zero after delay + duration.	*time*
period	Time period of oscillation	*time*
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 *(from current basePointCurrent)*

Event Ports

in	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 AND t < duration+delay THEN

i = weight * amplitude * sin(phase + (2 * 3.14159265 * (t-delay)/period))

IF t >= duration+delay THEN

i = 0

Schema

```
<xs:complexType name="SineGenerator">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="delay" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="phase" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="duration" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="amplitude" type="Nml2Quantity_current" use="required"/>
      <xs:attribute name="period" type="Nml2Quantity_time" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SineGenerator
from neuroml.utils import component_factory

variable = component_factory(
    SineGenerator,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    delay: 'a Nml2Quantity_time (required)' = None,
    phase: 'a Nml2Quantity_none (required)' = None,
    duration: 'a Nml2Quantity_time (required)' = None,
    amplitude: 'a Nml2Quantity_current (required)' = None,
    period: 'a Nml2Quantity_time (required)' = None,
)
```

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.	*time*
duration	Duration for holding current at amplitude. Current is zero after delay + duration.	*time*
period	Time period of oscillation	*time*
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 *(from* basePointCurrentDL*)*	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

Schema

```
<xs:complexType name="SineGeneratorDL">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="delay" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="phase" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="duration" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="amplitude" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="period" type="Nml2Quantity_time" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SineGeneratorDL
from neuroml.utils import component_factory

variable = component_factory(
    SineGeneratorDL,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    delay: 'a Nml2Quantity_time (required)' = None,
    phase: 'a Nml2Quantity_none (required)' = None,
    duration: 'a Nml2Quantity_time (required)' = None,
    amplitude: 'a Nml2Quantity_current (required)' = None,
    period: 'a Nml2Quantity_time (required)' = None,
)
```

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	*current*
delay	Delay before change in current. Current is baselineAmplitude prior to this.	*time*
duration	Duration for holding current at amplitude. Current is baselineAmplitude after delay + duration.	*time*
finishAmplitude	Amplitude of linearly varying current at time delay + duration	*current*
startAmplitude	Amplitude of linearly varying current at time delay	*current*

Properties

weight (default: 1)	Dimensionless

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from basePointCurrent)*	*current*

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

Schema

```
<xs:complexType name="RampGenerator">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="delay" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="duration" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="startAmplitude" type="Nml2Quantity_current" use="required"/>
      <xs:attribute name="finishAmplitude" type="Nml2Quantity_current" use="required"/
↪>
      <xs:attribute name="baselineAmplitude" type="Nml2Quantity_current" use="required
↪"/>
    </xs:extension>
```

(continues on next page)

(continued from previous page)

```
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import RampGenerator
from neuroml.utils import component_factory

variable = component_factory(
    RampGenerator,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    delay: 'a Nml2Quantity_time (required)' = None,
    duration: 'a Nml2Quantity_time (required)' = None,
    start_amplitude: 'a Nml2Quantity_current (required)' = None,
    finish_amplitude: 'a Nml2Quantity_current (required)' = None,
    baseline_amplitude: 'a Nml2Quantity_current (required)' = None,
)
```

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.	*time*
duration	Duration for holding current at amplitude. Current is baselineAmplitude after delay + duration.	*time*
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 *(from* basePointCurrentDL*)*	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

Schema

```
<xs:complexType name="RampGeneratorDL">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="delay" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="duration" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="startAmplitude" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="finishAmplitude" type="Nml2Quantity_none" use="required"/>
      <xs:attribute name="baselineAmplitude" type="Nml2Quantity_none" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import RampGeneratorDL
from neuroml.utils import component_factory

variable = component_factory(
    RampGeneratorDL,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    delay: 'a Nml2Quantity_time (required)' = None,
    duration: 'a Nml2Quantity_time (required)' = None,
    start_amplitude: 'a Nml2Quantity_current (required)' = None,
    finish_amplitude: 'a Nml2Quantity_current (required)' = None,
    baseline_amplitude: 'a Nml2Quantity_current (required)' = None,
)
```

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

delay	Delay before change in current. Current is zero prior to this.	*time*
duration	Duration for attempting to keep parent at targetVoltage. Current is zero after delay + duration.	*time*
simpleSeriesResistance	Current will be calculated by the difference in voltage between the target and parent, divided by this value	*resistance*
targetVoltage	Current will be applied to try to get parent to this target voltage	*voltage*

Properties

weight (default: 1)	Dimensionless

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepPointCurrent*)*	*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

Schema

```
<xs:complexType name="VoltageClamp">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="delay" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="duration" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="targetVoltage" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="simpleSeriesResistance" type="Nml2Quantity_resistance" use=
↪"required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import VoltageClamp
from neuroml.utils import component_factory

variable = component_factory(
    VoltageClamp,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    delay: 'a Nml2Quantity_time (required)' = None,
    duration: 'a Nml2Quantity_time (required)' = None,
    target_voltage: 'a Nml2Quantity_voltage (required)' = None,
    simple_series_resistance: 'a Nml2Quantity_resistance (required)' = None,
)
```

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
conditioningVoltage	Target voltage before time delay	*voltage*
delay	Delay before switching from conditioningVoltage to testingVoltage.	*time*
duration	Duration to hold at testingVoltage.	*time*
returnVoltage	Target voltage after time duration	*voltage*
simpleSeriesResistance	Current will be calculated by the difference in voltage between the target and parent, divided by this value	*resistance*
testingVoltage	Target voltage between times delay and delay + duration	*voltage*

Properties

weight (default: 1)	Dimensionless

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepPointCurrent*)*	*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 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

Schema

```
<xs:complexType name="VoltageClampTriple">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="active" type="ZeroOrOne" use="required"/>
      <xs:attribute name="delay" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="duration" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="conditioningVoltage" type="Nml2Quantity_voltage" use=
↪"required"/>
      <xs:attribute name="testingVoltage" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="returnVoltage" type="Nml2Quantity_voltage" use="required"/>
      <xs:attribute name="simpleSeriesResistance" type="Nml2Quantity_resistance" use=
↪"required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import VoltageClampTriple
from neuroml.utils import component_factory

variable = component_factory(
    VoltageClampTriple,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    active: 'a ZeroOrOne (required)' = None,
    delay: 'a Nml2Quantity_time (required)' = None,
    duration: 'a Nml2Quantity_time (required)' = None,
    conditioning_voltage: 'a Nml2Quantity_voltage (required)' = None,
```

(continues on next page)

(continued from previous page)

```
    testing_voltage: 'a Nml2Quantity_voltage (required)' = None,
    return_voltage: 'a Nml2Quantity_voltage (required)' = None,
    simple_series_resistance: 'a Nml2Quantity_resistance (required)' = None,
)
```

Usage: XML

```
<voltageClampTriple id="vClamp0" active="1" delay="50ms" duration="200ms"␣
↪conditioningVoltage="-70mV" testingVoltage="-50mV" returnVoltage="-70mV"␣
↪simpleSeriesResistance="1e6ohm"/>
```

13.1.8 Networks

Network descriptions for NeuroML 2. Describes *network* elements containing *population*s (potentially of type *populationList*, and so specifying a list of cell *location*s), *projection*s (i.e. lists of *connection*s) and *input*s.

Original ComponentType definitions: Networks.xml. Schema against which NeuroML based on these should be valid: NeuroML_v2.3.xsd. Generated on 14/08/24 from this commit. Please file any issues or questions at the issue tracker here.

---

network

extends baseStandalone

Network containing: *population*s (potentially of type *populationList*, and so specifying a list of cell *location*s); *projection*s (with lists of *connection*s) and/or *explicitConnection*s; and *inputList*s (with lists of *input*s) and/or *explicitInput*s. Note: often in NeuroML this will be of type *networkWithTemperature* if there are temperature dependent elements (e.g. ion channels).

Children list

regions	*region*
populations	*basePopulation*
projections	*projection*
synapticConnections	*explicitConnection*
electricalProjection	*electricalProjection*
continuousProjection	*continuousProjection*
explicitInputs	*explicitInput*
inputs	*inputList*

Schema

```
<xs:complexType name="Network">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:sequence>
        <xs:element name="space" type="Space" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="region" type="Region" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="extracellularProperties" type="ExtracellularPropertiesLocal
↪" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="population" type="Population" maxOccurs="unbounded"/>
        <xs:element name="cellSet" type="CellSet" minOccurs="0" maxOccurs="unbounded"/
↪>
        <xs:element name="synapticConnection" type="SynapticConnection" minOccurs="0"␣
↪maxOccurs="unbounded"/>
        <xs:element name="projection" type="Projection" minOccurs="0" maxOccurs=
↪"unbounded"/>
        <xs:element name="electricalProjection" type="ElectricalProjection" minOccurs=
↪"0" maxOccurs="unbounded"/>
        <xs:element name="continuousProjection" type="ContinuousProjection" minOccurs=
↪"0" maxOccurs="unbounded"/>
        <xs:element name="explicitInput" type="ExplicitInput" minOccurs="0" maxOccurs=
↪"unbounded"/>
        <xs:element name="inputList" type="InputList" minOccurs="0" maxOccurs=
↪"unbounded"/>
      </xs:sequence>
      <xs:attribute name="type" type="networkTypes" use="optional"/>
      <xs:attribute name="temperature" type="Nml2Quantity_temperature" use="optional"/
↪>
      <xs:attribute name="neuroLexId" type="NeuroLexId" use="optional"/>
    </xs:extension>
```

(continues on next page)

(continued from previous page)

```
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Network
from neuroml.utils import component_factory

variable = component_factory(
    Network,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    type: 'a networkTypes (optional)' = None,
    temperature: 'a Nml2Quantity_temperature (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    spaces: 'list of Space(s) (optional)' = None,
    regions: 'list of Region(s) (optional)' = None,
    extracellular_properties: 'list of ExtracellularPropertiesLocal(s) (optional)' =␣
 ↪None,
    populations: 'list of Population(s) (required)' = None,
    cell_sets: 'list of CellSet(s) (optional)' = None,
    synaptic_connections: 'list of SynapticConnection(s) (optional)' = None,
    projections: 'list of Projection(s) (optional)' = None,
    electrical_projections: 'list of ElectricalProjection(s) (optional)' = None,
    continuous_projections: 'list of ContinuousProjection(s) (optional)' = None,
    explicit_inputs: 'list of ExplicitInput(s) (optional)' = None,
    input_lists: 'list of InputList(s) (optional)' = None,
)
```

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

temperature	*temperature*

basePopulation

extends baseStandalone

A population of multiple instances of a specific **component,** which anything which extends *baseCell*.

Component References

component	*baseCell*

Child list

notes	*notes*
annotation	*annotation*

Children list

property	*property*

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 *instance*s (with *location*s) 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

Schema

```
<xs:complexType name="Population">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:choice>
        <xs:element name="layout" type="Layout" minOccurs="0"/>
        <xs:element name="instance" type="Instance" maxOccurs="unbounded"/>
      </xs:choice>
      <xs:attribute name="component" type="NmlId" use="required"/>
      <xs:attribute name="size" type="NonNegativeInteger" use="optional"/>
      <xs:attribute name="type" type="populationTypes" use="optional"/>
      <xs:attribute name="extracellularProperties" type="NmlId" use="optional"/>
      <xs:attribute name="neuroLexId" type="NeuroLexId" use="optional"/>
    </xs:extension>
```

(continues on next page)

(continued from previous page)

```
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Population
from neuroml.utils import component_factory

variable = component_factory(
    Population,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    component: 'a NmlId (required)' = None,
    size: 'a NonNegativeInteger (optional)' = None,
    type: 'a populationTypes (optional)' = None,
    extracellular_properties: 'a NmlId (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    layout: 'a Layout (optional)' = None,
    instances: 'list of Instance(s) (required)' = None,
)
```

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 *instance*s (with *location*s) 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

instances	*instance*

instance

Specifies a single instance of a component in a *population* (placed at *location*).

Child list

location	*location*

Schema

```
<xs:complexType name="Instance">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:sequence>
        <xs:element name="location" type="Location"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:nonNegativeInteger"/>
      <xs:attribute name="i" type="xs:nonNegativeInteger"/>
      <xs:attribute name="j" type="xs:nonNegativeInteger"/>
      <xs:attribute name="k" type="xs:nonNegativeInteger"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Instance
from neuroml.utils import component_factory

variable = component_factory(
    Instance,
    id: 'a nonNegativeInteger (optional)' = None,
    i: 'a nonNegativeInteger (optional)' = None,
    j: 'a nonNegativeInteger (optional)' = None,
    k: 'a nonNegativeInteger (optional)' = None,
    location: 'a Location (required)' = None,
)
```

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

Schema

```
<xs:complexType name="Location">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="x" type="xs:float" use="required"/>
      <xs:attribute name="y" type="xs:float" use="required"/>
      <xs:attribute name="z" type="xs:float" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Location
from neuroml.utils import component_factory

variable = component_factory(
    Location,
    x: 'a float (required)' = None,
    y: 'a float (required)' = None,
    z: 'a float (required)' = None,
)
```

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

rectangularExtent	*rectangularExtent*

Schema

```
<xs:complexType name="Region">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:any processContents="skip" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="space" type="NmlId" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Region
from neuroml.utils import component_factory

variable = component_factory(
    Region,
    id: 'a NmlId (required)' = None,
    spaces: 'a NmlId (optional)' = None,
    anytypeobjs_=None,
)
```

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	*baseSynapse*

Children list

connections	*connection*
connectionsWD	*connectionWD*

Schema

```
<xs:complexType name="Projection">
  <xs:complexContent>
    <xs:extension base="BaseProjection">
      <xs:sequence>
        <xs:element name="connection" type="Connection" minOccurs="0" maxOccurs=
 ↪"unbounded"/>
        <xs:element name="connectionWD" type="ConnectionWD" minOccurs="0" maxOccurs=
 ↪"unbounded"/>
      </xs:sequence>
      <xs:attribute name="synapse" type="NmlId" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Projection
from neuroml.utils import component_factory

variable = component_factory(
    Projection,
    id: 'a NmlId (required)' = None,
    presynaptic_population: 'a NmlId (required)' = None,
    postsynaptic_population: 'a NmlId (required)' = None,
    synapse: 'a NmlId (required)' = None,
    connections: 'list of Connection(s) (optional)' = None,
    connection_wds: 'list of ConnectionWD(s) (optional)' = None,
)
```

Usage: XML

```
<projection id="Proj1" presynapticPopulation="Population1" postsynapticPopulation=
 ↪"Population1" synapse="AMPA">

        </projection>
```

```
<projection id="internal1" presynapticPopulation="iafCells" postsynapticPopulation=
 ↪"iafCells" synapse="syn1">
            <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

destination
preFractionAlong
postFractionAlong
preSegmentId
postSegmentId

Paths

preCellId
postCellId

Schema

```
<xs:complexType name="Connection">
  <xs:complexContent>
    <xs:extension base="BaseConnectionOldFormat">
            </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Connection
from neuroml.utils import component_factory

variable = component_factory(
    Connection,
    id: 'a NonNegativeInteger (required)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    pre_cell_id: 'a Nml2PopulationReferencePath (required)' = None,
    pre_segment_id: 'a NonNegativeInteger (optional)' = '0',
    pre_fraction_along: 'a ZeroToOne (optional)' = '0.5',
    post_cell_id: 'a Nml2PopulationReferencePath (required)' = None,
    post_segment_id: 'a NonNegativeInteger (optional)' = '0',
    post_fraction_along: 'a ZeroToOne (optional)' = '0.5',
)
```

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	*baseSynapse*

Schema

```
<xs:complexType name="SynapticConnection">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="neuroLexId" type="NeuroLexId" use="optional"/>
      <xs:attribute name="from" type="Nml2PopulationReferencePath" use="required"/>
      <xs:attribute name="to" type="Nml2PopulationReferencePath" use="required"/>
      <xs:attribute name="synapse" type="NmlId" use="required"/>
      <xs:attribute name="destination" type="NmlId" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SynapticConnection
from neuroml.utils import component_factory

variable = component_factory(
    SynapticConnection,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    from_: 'a Nml2PopulationReferencePath (required)' = None,
    to: 'a Nml2PopulationReferencePath (required)' = None,
    synapse: 'a NmlId (required)' = None,
    destination: 'a NmlId (optional)' = None,
)
```

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	*time*
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	*time*
weight	Dimensionless

Text fields

destination
preFractionAlong
postFractionAlong
preSegmentId
postSegmentId

Paths

preCellId
postCellId

Schema

```
<xs:complexType name="ConnectionWD">
  <xs:complexContent>
    <xs:extension base="BaseConnectionOldFormat">
      <xs:attribute name="weight" type="xs:float" use="required"/>
      <xs:attribute name="delay" type="Nml2Quantity_time" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ConnectionWD
from neuroml.utils import component_factory

variable = component_factory(
    ConnectionWD,
    id: 'a NonNegativeInteger (required)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    pre_cell_id: 'a Nml2PopulationReferencePath (required)' = None,
    pre_segment_id: 'a NonNegativeInteger (optional)' = '0',
    pre_fraction_along: 'a ZeroToOne (optional)' = '0.5',
    post_cell_id: 'a Nml2PopulationReferencePath (required)' = None,
    post_segment_id: 'a NonNegativeInteger (optional)' = '0',
    post_fraction_along: 'a ZeroToOne (optional)' = '0.5',
    weight: 'a float (required)' = None,
    delay: 'a Nml2Quantity_time (required)' = None,
)
```

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

synapse	*gapJunction*

Schema

```
<xs:complexType name="ElectricalConnection">
  <xs:complexContent>
    <xs:extension base="BaseConnectionNewFormat">
      <xs:attribute name="synapse" type="NmlId" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ElectricalConnection
from neuroml.utils import component_factory

variable = component_factory(
    ElectricalConnection,
    id: 'a NonNegativeInteger (required)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    pre_cell: 'a string (required)' = None,
    pre_segment: 'a NonNegativeInteger (optional)' = '0',
    pre_fraction_along: 'a ZeroToOne (optional)' = '0.5',
    post_cell: 'a string (required)' = None,
    post_segment: 'a NonNegativeInteger (optional)' = '0',
    post_fraction_along: 'a ZeroToOne (optional)' = '0.5',
    synapse: 'a NmlId (required)' = None,
    extensiontype_=None,
)
```

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

preFractionAlong	
postFractionAlong	
preSegment	
postSegment	

Paths

preCell	
postCell	

Component References

synapse	*gapJunction*

Schema

```
<xs:complexType name="ElectricalConnectionInstance">
  <xs:complexContent>
    <xs:extension base="ElectricalConnection"/>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ElectricalConnectionInstance
from neuroml.utils import component_factory

variable = component_factory(
    ElectricalConnectionInstance,
    id: 'a NonNegativeInteger (required)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    pre_cell: 'a string (required)' = None,
```

(continues on next page)

(continued from previous page)

```
    pre_segment: 'a NonNegativeInteger (optional)' = '0',
    pre_fraction_along: 'a ZeroToOne (optional)' = '0.5',
    post_cell: 'a string (required)' = None,
    post_segment: 'a NonNegativeInteger (optional)' = '0',
    post_fraction_along: 'a ZeroToOne (optional)' = '0.5',
    synapse: 'a NmlId (required)' = None,
    extensiontype_=None,
)
```

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

Schema

```
<xs:complexType name="ElectricalConnectionInstanceW">
  <xs:complexContent>
    <xs:extension base="ElectricalConnectionInstance">
      <xs:attribute name="weight" type="xs:float" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ElectricalConnectionInstanceW
from neuroml.utils import component_factory

variable = component_factory(
    ElectricalConnectionInstanceW,
    id: 'a NonNegativeInteger (required)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    pre_cell: 'a string (required)' = None,
    pre_segment: 'a NonNegativeInteger (optional)' = '0',
    pre_fraction_along: 'a ZeroToOne (optional)' = '0.5',
    post_cell: 'a string (required)' = None,
    post_segment: 'a NonNegativeInteger (optional)' = '0',
    post_fraction_along: 'a ZeroToOne (optional)' = '0.5',
    synapse: 'a NmlId (required)' = None,
    weight: 'a float (required)' = None,
)
```

electricalProjection

A projection between **presynapticPopulation** to another **postsynapticPopulation** through gap junctions.

Component References

presynapticPopulation	*population*
postsynapticPopulation	*population*

Children list

connections	*electricalConnection*
connectionInstances	*electricalConnectionInstance*

Schema

```
<xs:complexType name="ElectricalProjection">
  <xs:complexContent>
    <xs:extension base="BaseProjection">
      <xs:sequence>
        <xs:element name="electricalConnection" type="ElectricalConnection" minOccurs=
↪"0" maxOccurs="unbounded"/>
        <xs:element name="electricalConnectionInstance" type=
↪"ElectricalConnectionInstance" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="electricalConnectionInstanceW" type=
↪"ElectricalConnectionInstanceW" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ElectricalProjection
from neuroml.utils import component_factory

variable = component_factory(
    ElectricalProjection,
    id: 'a NmlId (required)' = None,
    presynaptic_population: 'a NmlId (required)' = None,
    postsynaptic_population: 'a NmlId (required)' = None,
    electrical_connections: 'list of ElectricalConnection(s) (optional)' = None,
```

(continues on next page)

(continued from previous page)

```
    electrical_connection_instances: 'list of ElectricalConnectionInstance(s)␣
↪(optional)' = None,
    electrical_connection_instance_ws: 'list of ElectricalConnectionInstanceW(s)␣
↪(optional)' = None,
)
```

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

preComponent	*baseGradedSynapse*
postComponent	*baseGradedSynapse*

Schema

```
<xs:complexType name="ContinuousConnection">
  <xs:complexContent>
    <xs:extension base="BaseConnectionNewFormat">
      <xs:attribute name="preComponent" type="NmlId" use="required"/>
      <xs:attribute name="postComponent" type="NmlId" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ContinuousConnection
from neuroml.utils import component_factory

variable = component_factory(
    ContinuousConnection,
    id: 'a NonNegativeInteger (required)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    pre_cell: 'a string (required)' = None,
    pre_segment: 'a NonNegativeInteger (optional)' = '0',
    pre_fraction_along: 'a ZeroToOne (optional)' = '0.5',
    post_cell: 'a string (required)' = None,
    post_segment: 'a NonNegativeInteger (optional)' = '0',
    post_fraction_along: 'a ZeroToOne (optional)' = '0.5',
    pre_component: 'a NmlId (required)' = None,
    post_component: 'a NmlId (required)' = None,
    extensiontype_=None,
)
```

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

preFractionAlong	
postFractionAlong	
preSegment	
postSegment	

Paths

preCell
postCell

Component References

pre-Compo-nent	*baseGradedSynapse*
post-Compo-nent	*baseGradedSynapse*

Schema

```
<xs:complexType name="ContinuousConnectionInstance">
  <xs:complexContent>
    <xs:extension base="ContinuousConnection"/>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ContinuousConnectionInstance
from neuroml.utils import component_factory

variable = component_factory(
    ContinuousConnectionInstance,
    id: 'a NonNegativeInteger (required)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    pre_cell: 'a string (required)' = None,
    pre_segment: 'a NonNegativeInteger (optional)' = '0',
    pre_fraction_along: 'a ZeroToOne (optional)' = '0.5',
    post_cell: 'a string (required)' = None,
    post_segment: 'a NonNegativeInteger (optional)' = '0',
    post_fraction_along: 'a ZeroToOne (optional)' = '0.5',
    pre_component: 'a NmlId (required)' = None,
    post_component: 'a NmlId (required)' = None,
    extensiontype_=None,
)
```

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

preFractionAlong
postFractionAlong
preSegment
postSegment

Paths

preCell
postCell

Schema

```
<xs:complexType name="ContinuousConnectionInstanceW">
  <xs:complexContent>
    <xs:extension base="ContinuousConnectionInstance">
      <xs:attribute name="weight" type="xs:float" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ContinuousConnectionInstanceW
from neuroml.utils import component_factory

variable = component_factory(
    ContinuousConnectionInstanceW,
    id: 'a NonNegativeInteger (required)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    pre_cell: 'a string (required)' = None,
    pre_segment: 'a NonNegativeInteger (optional)' = '0',
    pre_fraction_along: 'a ZeroToOne (optional)' = '0.5',
    post_cell: 'a string (required)' = None,
    post_segment: 'a NonNegativeInteger (optional)' = '0',
    post_fraction_along: 'a ZeroToOne (optional)' = '0.5',
    pre_component: 'a NmlId (required)' = None,
    post_component: 'a NmlId (required)' = None,
    weight: 'a float (required)' = None,
)
```

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

presynapticPopulation	*population*
postsynapticPopulation	*population*

Children list

connections	*continuousConnection*
connectionInstances	*continuousConnectionInstance*

Schema

```
<xs:complexType name="ContinuousProjection">
  <xs:complexContent>
    <xs:extension base="BaseProjection">
      <xs:sequence>
        <xs:element name="continuousConnection" type="ContinuousConnection" minOccurs=
↪"0" maxOccurs="unbounded"/>
        <xs:element name="continuousConnectionInstance" type=
↪"ContinuousConnectionInstance" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="continuousConnectionInstanceW" type=
↪"ContinuousConnectionInstanceW" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ContinuousProjection
from neuroml.utils import component_factory

variable = component_factory(
    ContinuousProjection,
    id: 'a NmlId (required)' = None,
    presynaptic_population: 'a NmlId (required)' = None,
    postsynaptic_population: 'a NmlId (required)' = None,
    continuous_connections: 'list of ContinuousConnection(s) (optional)' = None,
    continuous_connection_instances: 'list of ContinuousConnectionInstance(s)␣
↪(optional)' = None,
    continuous_connection_instance_ws: 'list of ContinuousConnectionInstanceW(s)␣
↪(optional)' = None,
)
```

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"/>
</continuousProjection>
```

explicitInput

An explicit input (anything which extends *basePointCurrent*) to a target cell in a population.

Text fields

destination	
sourcePort	
targetPort	

Paths

target	

Component References

input	*basePointCurrent*

Schema

```
<xs:complexType name="ExplicitInput">
  <xs:complexContent>
    <xs:extension base="BaseWithoutId">
      <xs:attribute name="target" type="Nml2PopulationReferencePath" use="required"/>
      <xs:attribute name="input" type="NmlId" use="required"/>
      <xs:attribute name="destination" type="NmlId"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ExplicitInput
from neuroml.utils import component_factory

variable = component_factory(
    ExplicitInput,
    target: 'a Nml2PopulationReferencePath (required)' = None,
    input: 'a NmlId (required)' = None,
    destination: 'a NmlId (optional)' = None,
)
```

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 *input*s to a **population.**.

Text fields

population

Component References

component	*basePointCurrent*

Children list

inputs	*input*

Schema

```
<xs:complexType name="InputList">
  <xs:complexContent>
    <xs:extension base="Base">
      <xs:sequence>
        <xs:element name="input" type="Input" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="inputW" type="InputW" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="population" type="NmlId" use="required"/>
      <xs:attribute name="component" type="NmlId" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import InputList
from neuroml.utils import component_factory

variable = component_factory(
    InputList,
    id: 'a NonNegativeInteger (required)' = None,
    populations: 'a NmlId (required)' = None,
    component: 'a NmlId (required)' = None,
    input: 'list of Input(s) (optional)' = None,
    input_ws: 'list of InputW(s) (optional)' = None,
)
```

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
fractionAlong	Optional specification of the fraction along the specified segment, default 0.5
destination	

Paths

target

Schema

```
<xs:complexType name="Input">
  <xs:complexContent>
    <xs:extension base="BaseNonNegativeIntegerId">
      <xs:attribute name="target" type="Nml2PopulationReferencePath" use="required"/>
      <xs:attribute name="destination" type="NmlId" use="required"/>
      <xs:attribute name="segmentId" type="NonNegativeInteger"/>
      <xs:attribute name="fractionAlong" type="ZeroToOne"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import Input
from neuroml.utils import component_factory

variable = component_factory(
    Input,
    id: 'a NonNegativeInteger (required)' = None,
    target: 'a Nml2PopulationReferencePath (required)' = None,
    destination: 'a NmlId (required)' = None,
    segment_id: 'a NonNegativeInteger (optional)' = None,
    fraction_along: 'a ZeroToOne (optional)' = None,
    extensiontype_=None,
)
```

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

destination

Paths

target

Schema

```
<xs:complexType name="InputW">
  <xs:complexContent>
    <xs:extension base="Input">
      <xs:attribute name="weight" type="xs:float" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import InputW
from neuroml.utils import component_factory

variable = component_factory(
    InputW,
    id: 'a NonNegativeInteger (required)' = None,
    target: 'a Nml2PopulationReferencePath (required)' = None,
    destination: 'a NmlId (required)' = None,
    segment_id: 'a NonNegativeInteger (optional)' = None,
    fraction_along: 'a ZeroToOne (optional)' = None,
    weight: 'a float (required)' = None,
)
```

13.1.9 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_v2.3.xsd. Generated on 14/08/24 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_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
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	*time*
MVOLT = 1mV	*voltage*
NFARAD = 1nF	*capacitance*

Exposures

iSyn		*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out
spike_in_		Direction: in
spike_in_		Direction: in

Schema

```
<xs:complexType name="basePyNNCell">
  <xs:complexContent>
    <xs:extension base="BaseCell">
      <xs:attribute name="cm" type="xs:float" use="required"/>
      <xs:attribute name="i_offset" type="xs:float" use="required"/>
      <xs:attribute name="tau_syn_E" type="xs:float" use="required"/>
      <xs:attribute name="tau_syn_I" type="xs:float" use="required"/>
      <xs:attribute name="v_init" type="xs:float" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

basePyNNIaFCell

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_refra		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_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*)*	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out
spike_in_	*(from* basePyNNCell*)*	Direction: in
spike_in_	*(from* basePyNNCell*)*	Direction: in

Schema

```
<xs:complexType name="basePyNNIaFCell">
  <xs:complexContent>
    <xs:extension base="basePyNNCell">
      <xs:attribute name="tau_m" type="xs:float" use="required"/>
      <xs:attribute name="tau_refrac" type="xs:float" use="required"/>
      <xs:attribute name="v_reset" type="xs:float" use="required"/>
      <xs:attribute name="v_rest" type="xs:float" use="required"/>
      <xs:attribute name="v_thresh" type="xs:float" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

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_refra	*(from* basePyNNIaFCell*)*	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_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*)*	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out
spike_in_	*(from* basePyNNCell*)*	Direction: in
spike_in_	*(from* basePyNNCell*)*	Direction: in

Schema

```
<xs:complexType name="basePyNNIaFCondCell">
  <xs:complexContent>
    <xs:extension base="basePyNNIaFCell">
      <xs:attribute name="e_rev_E" type="xs:float" use="required"/>
      <xs:attribute name="e_rev_I" type="xs:float" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

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_refra	*(from* basePyNNIaFCell*)*	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_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*)*	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out
spike_in_	*(from* basePyNNCell*)*	Direction: in
spike_in_	*(from* basePyNNCell*)*	Direction: in

Attachments

synapses	*baseSynapse*

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))

Schema

```
<xs:complexType name="IF_curr_alpha">
  <xs:complexContent>
    <xs:extension base="basePyNNIaFCell">
            </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IF_curr_alpha
from neuroml.utils import component_factory

variable = component_factory(
    IF_curr_alpha,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    cm: 'a float (required)' = None,
    i_offset: 'a float (required)' = None,
    tau_syn_E: 'a float (required)' = None,
    tau_syn_I: 'a float (required)' = None,
    v_init: 'a float (required)' = None,
    tau_m: 'a float (required)' = None,
    tau_refrac: 'a float (required)' = None,
    v_reset: 'a float (required)' = None,
    v_rest: 'a float (required)' = None,
    v_thresh: 'a float (required)' = None,
)
```

Usage: XML

```
<IF_curr_alpha id="IF_curr_alpha" cm="1.0" i_offset="0.9" tau_m="20.0" tau_refrac="10.
 ↪0" tau_syn_E="0.5" tau_syn_I="0.5" v_init="-65" v_reset="-62.0" v_rest="-65.0" 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_refra	*(from* basePyNNIaFCell*)*	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_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*)*	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out
spike_in_	*(from* basePyNNCell*)*	Direction: in
spike_in_	*(from* basePyNNCell*)*	Direction: in

Attachments

synapses	*baseSynapse*

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))

Schema

```
<xs:complexType name="IF_curr_exp">
  <xs:complexContent>
    <xs:extension base="basePyNNIaFCell">
            </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IF_curr_exp
from neuroml.utils import component_factory

variable = component_factory(
    IF_curr_exp,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    cm: 'a float (required)' = None,
    i_offset: 'a float (required)' = None,
    tau_syn_E: 'a float (required)' = None,
    tau_syn_I: 'a float (required)' = None,
    v_init: 'a float (required)' = None,
    tau_m: 'a float (required)' = None,
    tau_refrac: 'a float (required)' = None,
    v_reset: 'a float (required)' = None,
    v_rest: 'a float (required)' = None,
    v_thresh: 'a float (required)' = None,
)
```

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_refra	*(from* basePyNNIaFCell*)*	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_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*)*	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out
spike_in_	*(from* basePyNNCell*)*	Direction: in
spike_in_	*(from* basePyNNCell*)*	Direction: in

Attachments

synapses	*baseSynapse*

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))

Schema

```
<xs:complexType name="IF_cond_alpha">
  <xs:complexContent>
    <xs:extension base="basePyNNIaFCondCell">
          </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IF_cond_alpha
from neuroml.utils import component_factory

variable = component_factory(
    IF_cond_alpha,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    cm: 'a float (required)' = None,
    i_offset: 'a float (required)' = None,
    tau_syn_E: 'a float (required)' = None,
    tau_syn_I: 'a float (required)' = None,
    v_init: 'a float (required)' = None,
    tau_m: 'a float (required)' = None,
    tau_refrac: 'a float (required)' = None,
    v_reset: 'a float (required)' = None,
    v_rest: 'a float (required)' = None,
    v_thresh: 'a float (required)' = None,
    e_rev_E: 'a float (required)' = None,
    e_rev_I: 'a float (required)' = None,
)
```

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_refra	*(from* basePyNNIaFCell*)*	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_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*)*	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out
spike_in_	*(from* basePyNNCell*)*	Direction: in
spike_in_	*(from* basePyNNCell*)*	Direction: in

Attachments

synapses	*baseSynapse*

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))

Schema

```
<xs:complexType name="IF_cond_exp">
  <xs:complexContent>
    <xs:extension base="basePyNNIaFCondCell">
          </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import IF_cond_exp
from neuroml.utils import component_factory

variable = component_factory(
    IF_cond_exp,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    cm: 'a float (required)' = None,
    i_offset: 'a float (required)' = None,
    tau_syn_E: 'a float (required)' = None,
    tau_syn_I: 'a float (required)' = None,
    v_init: 'a float (required)' = None,
    tau_m: 'a float (required)' = None,
    tau_refrac: 'a float (required)' = None,
    v_reset: 'a float (required)' = None,
    v_rest: 'a float (required)' = None,
    v_thresh: 'a float (required)' = None,
    e_rev_E: 'a float (required)' = None,
    e_rev_I: 'a float (required)' = None,
)
```

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	*(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* 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_refra	*(from* basePyNNIaFCell*)*	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_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_thresh	Dimensionless

eif_threshold = v_spike * H(delta_T-1e-12) + v_thresh * H(-1*delta_T+1e-9)

Exposures

iSyn	*(from* basePyNNCell*)*	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*
w		Dimensionless

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out
spike_in_	*(from* basePyNNCell*)*	Direction: in
spike_in_	*(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

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

Schema

```
<xs:complexType name="EIF_cond_exp_isfa_ista">
  <xs:complexContent>
    <xs:extension base="basePyNNIaFCondCell">
      <xs:attribute name="a" type="xs:float" use="required"/>
      <xs:attribute name="b" type="xs:float" use="required"/>
      <xs:attribute name="delta_T" type="xs:float" use="required"/>
      <xs:attribute name="tau_w" type="xs:float" use="required"/>
      <xs:attribute name="v_spike" type="xs:float" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import EIF_cond_exp_isfa_ista
from neuroml.utils import component_factory

variable = component_factory(
    EIF_cond_exp_isfa_ista,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    cm: 'a float (required)' = None,
    i_offset: 'a float (required)' = None,
    tau_syn_E: 'a float (required)' = None,
    tau_syn_I: 'a float (required)' = None,
    v_init: 'a float (required)' = None,
    tau_m: 'a float (required)' = None,
    tau_refrac: 'a float (required)' = None,
```

(continues on next page)

(continued from previous page)

```
    v_reset: 'a float (required)' = None,
    v_rest: 'a float (required)' = None,
    v_thresh: 'a float (required)' = None,
    e_rev_E: 'a float (required)' = None,
    e_rev_I: 'a float (required)' = None,
    a: 'a float (required)' = None,
    b: 'a float (required)' = None,
    delta_T: 'a float (required)' = None,
    tau_w: 'a float (required)' = None,
    v_spike: 'a float (required)' = None,
    extensiontype_=None,
)
```

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* 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_refra	*(from* basePyNNIaFCell*)*	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_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_thresh	Dimensionless

eif_threshold = v_spike * H(delta_T-1e-12) + v_thresh * H(-1*delta_T+1e-9)

Exposures

iSyn	*(from* basePyNNCell*)*	*current*
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*
w		Dimensionless

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out
spike_in_	*(from* basePyNNCell*)*	Direction: in
spike_in_	*(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

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

Schema

```
<xs:complexType name="EIF_cond_alpha_isfa_ista">
  <xs:complexContent>
    <xs:extension base="EIF_cond_exp_isfa_ista">
            </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import EIF_cond_alpha_isfa_ista
from neuroml.utils import component_factory

variable = component_factory(
    EIF_cond_alpha_isfa_ista,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    cm: 'a float (required)' = None,
    i_offset: 'a float (required)' = None,
    tau_syn_E: 'a float (required)' = None,
    tau_syn_I: 'a float (required)' = None,
    v_init: 'a float (required)' = None,
    tau_m: 'a float (required)' = None,
    tau_refrac: 'a float (required)' = None,
    v_reset: 'a float (required)' = None,
    v_rest: 'a float (required)' = None,
    v_thresh: 'a float (required)' = None,
    e_rev_E: 'a float (required)' = None,
    e_rev_I: 'a float (required)' = None,
```

(continues on next page)

(continued from previous page)

```
    a: 'a float (required)' = None,
    b: 'a float (required)' = None,
    delta_T: 'a float (required)' = None,
    tau_w: 'a float (required)' = None,
    v_spike: 'a float (required)' = None,
)
```

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_leal		Dimensionless
g_leak		Dimensionless
gbar_K		Dimensionless
gbar_Na		Dimensionless
i_offset	*(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_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* basePyNNCell*)*	*current*
m		Dimensionless
n		Dimensionless
v	Membrane potential *(from* baseCellMembPot*)*	*voltage*

Event Ports

spike	Spike event *(from* baseSpikingCell*)*	Direction: out
spike_in_	*(from* basePyNNCell*)*	Direction: in
spike_in_	*(from* basePyNNCell*)*	Direction: in

Attachments

synapses	*baseSynapse*

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_leak * (e_rev_leak - (v / MVOLT))

iNa = gbar_Na * (m * m * m) * h * (e_rev_Na - (v / MVOLT))

iK = gbar_K * (n * n * n * n) * (e_rev_K - (v / MVOLT))

iMemb = iLeak + iNa + iK + i_offset

alpham = 0.32 * (13 - (v / MVOLT) + v_offset) / (exp((13 - (v / MVOLT) + v_offset) / 4.0) - 1)

betam = 0.28 * ((v / MVOLT) - v_offset - 40) / (exp(((v / MVOLT) - v_offset - 40) / 5.0) - 1)

alphah = 0.128 * exp((17 - (v / MVOLT) + v_offset) / 18.0)

betah = 4.0 / (1 + exp((40 - (v / MVOLT) + v_offset) / 5))

alphan = 0.032 * (15 - (v / MVOLT) + v_offset) / (exp((15 - (v / MVOLT) + v_offset) / 5) - 1)

betan = 0.5 * exp((10 - (v / MVOLT) + v_offset) / 40)

Time Derivatives

d **v** /dt = (MVOLT * (iMemb / cm) / MSEC) + (iSyn / (cm * NFARAD))

d **m** /dt = (alpham * (1 - m) - betam * m) / MSEC

d **h** /dt = (alphah * (1 - h) - betah * h) / MSEC

d **n** /dt = (alphan * (1 - n) - betan * n) / MSEC

Schema

```
<xs:complexType name="HH_cond_exp">
  <xs:complexContent>
    <xs:extension base="basePyNNCell">
      <xs:attribute name="v_offset" type="xs:float" use="required"/>
      <xs:attribute name="e_rev_E" type="xs:float" use="required"/>
      <xs:attribute name="e_rev_I" type="xs:float" use="required"/>
      <xs:attribute name="e_rev_K" type="xs:float" use="required"/>
      <xs:attribute name="e_rev_Na" type="xs:float" use="required"/>
      <xs:attribute name="e_rev_leak" type="xs:float" use="required"/>
      <xs:attribute name="g_leak" type="xs:float" use="required"/>
      <xs:attribute name="gbar_K" type="xs:float" use="required"/>
      <xs:attribute name="gbar_Na" type="xs:float" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import HH_cond_exp
from neuroml.utils import component_factory

variable = component_factory(
    HH_cond_exp,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    cm: 'a float (required)' = None,
    i_offset: 'a float (required)' = None,
    tau_syn_E: 'a float (required)' = None,
    tau_syn_I: 'a float (required)' = None,
    v_init: 'a float (required)' = None,
    v_offset: 'a float (required)' = None,
    e_rev_E: 'a float (required)' = None,
    e_rev_I: 'a float (required)' = None,
    e_rev_K: 'a float (required)' = None,
    e_rev_Na: 'a float (required)' = None,
    e_rev_leak: 'a float (required)' = None,
    g_leak: 'a float (required)' = None,
    gbar_K: 'a float (required)' = None,
```

(continues on next page)

(continued from previous page)

```
    gbar_Na: 'a float (required)' = None,
)
```

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	*time*
MVOLT = 1mV	*voltage*
NAMP = 1nA	*current*

Exposures

i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepSynapse*)*	*voltage*

Event Ports

in	*(from* baseSynapse*)*	Direction: in

Schema

```
<xs:complexType name="BasePynnSynapse">
  <xs:complexContent>
    <xs:extension base="BaseSynapse">
      <xs:attribute name="tau_syn" type="xs:float" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import BasePynnSynapse
from neuroml.utils import component_factory

variable = component_factory(
    BasePynnSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    tau_syn: 'a float (required)' = None,
    extensiontype_=None,
)
```

expCondSynapse

extends basePynnSynapse

Conductance based synapse with instantaneous rise and single exponential decay (with time constant tau_syn).

Parameters

e_rev		Dimensionless
tau_syn	*(from* basePynnSynapse*)*	Dimensionless

Properties

weight (default: 1)	Dimensionless

Exposures

g		Dimensionless
i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepSynapse*)*	*voltage*

Event Ports

in	*(from* baseSynapse*)*	Direction: in

Dynamics

State Variables
g: Dimensionless (exposed as **g**)

On Events
EVENT IN on port: **in**

g = g+weight

Derived Variables
i = g * (e_rev - (v/MVOLT)) * NAMP (exposed as **i**)

Time Derivatives
d **g** /dt = -g / (tau_syn*MSEC)

Schema

```
<xs:complexType name="ExpCondSynapse">
  <xs:complexContent>
    <xs:extension base="BasePynnSynapse">
      <xs:attribute name="e_rev" type="xs:float" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ExpCondSynapse
from neuroml.utils import component_factory

variable = component_factory(
    ExpCondSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    tau_syn: 'a float (required)' = None,
    e_rev: 'a float (required)' = None,
)
```

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*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepSynapse*)*	*voltage*

Event Ports

in	*(from* baseSynapse*)*	Direction: in

Dynamics

State Variables
I: Dimensionless

On Events
EVENT IN on port: **in**

I = I + weight

Derived Variables
i = I * NAMP (exposed as **i**)

Time Derivatives
d **I** /dt = -I / (tau_syn*MSEC)

Schema

```
<xs:complexType name="ExpCurrSynapse">
  <xs:complexContent>
    <xs:extension base="BasePynnSynapse">

            </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import ExpCurrSynapse
from neuroml.utils import component_factory

variable = component_factory(
    ExpCurrSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
```

(continues on next page)

(continued from previous page)

```
    tau_syn: 'a float (required)' = None,
)
```

Usage: XML

```
<expCurrSynapse id="syn3" tau_syn="5"/>
```

alphaCondSynapse

extends 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*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepSynapse*)*	*voltage*

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 + weight

Derived Variables

i = g * (e_rev - (v/MVOLT)) * NAMP (exposed as **i**)

Time Derivatives

d **g** /dt = (2.7182818*A* - *g*)/(*tau_syn*MSEC)

d **A** /dt = -A /(tau_syn*MSEC)

Schema

```
<xs:complexType name="AlphaCondSynapse">
  <xs:complexContent>
    <xs:extension base="BasePynnSynapse">
      <xs:attribute name="e_rev" type="xs:float" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import AlphaCondSynapse
from neuroml.utils import component_factory

variable = component_factory(
    AlphaCondSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    tau_syn: 'a float (required)' = None,
    e_rev: 'a float (required)' = None,
)
```

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		*current*
i	The total (usually time varying) current produced by this ComponentType *(from* basePointCurrent*)*	*current*

Requirements

v	The current may vary with the voltage exposed by the ComponentType on which this is placed *(from* baseVoltageDepSynapse*)*	*voltage*

Event Ports

in	*(from* baseSynapse*)*	Direction: in

Dynamics

State Variables

I: Dimensionless

A: Dimensionless (exposed as **A**)

On Events

EVENT IN on port: **in**

A = A + weight

Derived Variables

i = I * NAMP (exposed as **i**)

Time Derivatives

d **I** /dt = (2.7182818*A* - *I*)/(*tau_syn*MSEC)

d **A** /dt = -A /(tau_syn*MSEC)

Schema

```
<xs:complexType name="AlphaCurrSynapse">
  <xs:complexContent>
    <xs:extension base="BasePynnSynapse">

            </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import AlphaCurrSynapse
from neuroml.utils import component_factory

variable = component_factory(
    AlphaCurrSynapse,
    id: 'a NmlId (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    neuro_lex_id: 'a NeuroLexId (optional)' = None,
    tau_syn: 'a float (required)' = None,
)
```

Usage: XML

```
<alphaCurrSynapse id="syn4" tau_syn="5"/>
```

SpikeSourcePoisson

extends baseSpikeSource

Spike source, generating spikes according to a Poisson process.

Parameters

duration		*time*
rate		*per_time*
start		*time*

Constants

LONG_TIME = 1e9hour	*time*
SMALL_TIME = 1e-9ms	*time*

Derived parameters

end	*time*

end = start + duration

Exposures

isi		*time*
tnextIdeal		*time*
tnextUsed		*time*
tsince	Time since the last spike was emitted *(from* baseSpikeSource*)*	*time*

Event Ports

in		Direction: in
spike	Port on which spikes are emitted *(from* baseSpikeSource*)*	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

Schema

```
<xs:complexType name="SpikeSourcePoisson">
  <xs:complexContent>
    <xs:extension base="Standalone">
      <xs:attribute name="start" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="duration" type="Nml2Quantity_time" use="required"/>
      <xs:attribute name="rate" type="Nml2Quantity_pertime" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Usage: Python

Go to the libNeuroML documentation

```
from neuroml import SpikeSourcePoisson
from neuroml.utils import component_factory

variable = component_factory(
    SpikeSourcePoisson,
    id: 'a NonNegativeInteger (required)' = None,
    metaid: 'a MetaId (optional)' = None,
    notes: 'a string (optional)' = None,
    properties: 'list of Property(s) (optional)' = None,
    annotation: 'a Annotation (optional)' = None,
    start: 'a Nml2Quantity_time (required)' = None,
    duration: 'a Nml2Quantity_time (required)' = None,
    rate: 'a Nml2Quantity_pertime (required)' = None,
)
```

Usage: XML

```
<SpikeSourcePoisson id="spikes1" start="50ms" duration="400ms" rate="50Hz"/>
```

```
<SpikeSourcePoisson id="spikes2" start="50ms" duration="300ms" rate="80Hz"/>
```

13.1.10 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 *OutputFile*s.

Original ComponentType definitions: Simulation.xml. Schema against which NeuroML based on these should be valid: NeuroML_v2.3.xsd. Generated on 14/08/24 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 *Display*s, *OutputFile*s and *EventOutputFile*s to record. Specifies a **target** component to run, usually the id of a *network*.

Parameters

length	Duration of the simulation run	*time*
step	Time step (dt) to use in the simulation	*time*

Text fields

seed	The seed to use in the random number generator for stochastic entities

Component References

target	schema:component

Children list

metas	*Meta*
displays	*Display*
outputs	*OutputFile*
events	*EventOutputFile*

Dynamics

State Variables

t: *time*

Display

Details of a display to generate (usually a set of traces given by *Line*s 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	*time*
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	*Line*

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.	*Dimensions*
timeScale	An optional scaling of the time axis, e.g. 1ms means display in milliseconds. Note: if present, this overrides timeScale from *Display*	*Dimensions*

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

outputColumn	*OutputColumn*

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 *EventSelection*) in each row of the file

Children list

eventSelection	*EventSelection*

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

Meta

Metadata to add to simulation.

Text fields

for	Simulator name
method	Integration method to use
abs_toleranc	Absolute tolerance for NEURON's cvode method
rel_toleranc	Relative tolerance for NEURON's cvode method

13.1.11 Index

- *adExIaFCell*
- *alphaCondSynapse*
- *alphaCurrentSynapse*
- *alphaCurrSynapse*
- *alphaSynapse*
- *annotation*
- *area*
- *baseBlockMechanism*
- *baseBqbiol*
- *baseCell*
- *baseCellMembPot*

- *baseCellMembPotCap*
- *baseCellMembPotDL*
- *baseChannelDensity*
- *baseChannelDensityCond*
- *baseChannelPopulation*
- *baseConductanceBasedSynapse*
- *baseConductanceBasedSynapseTwo*
- *baseConductanceScaling*
- *baseConductanceScalingCaDependent*
- *baseCurrentBasedSynapse*
- *baseGate*
- *baseGradedSynapse*
- *baseHHRate*
- *baseHHVariable*
- *baseIaf*
- *baseIafCapCell*
- *baseIonChannel*
- *basePlasticityMechanism*
- *basePointCurrent*
- *basePointCurrentDL*
- *basePopulation*
- *basePyNNCell*
- *basePyNNIaFCell*
- *basePyNNIaFCondCell*
- *basePynnSynapse*
- *baseQ10Settings*
- *baseSpikeSource*
- *baseSpikingCell*
- *baseStandalone*
- *baseSynapse*
- *baseSynapseDL*
- *baseVoltageConcDepRate*
- *baseVoltageConcDepTime*
- *baseVoltageConcDepVariable*
- *baseVoltageDepPointCurrent*
- *baseVoltageDepPointCurrentDL*

- *baseVoltageDepPointCurrentSpiking*
- *baseVoltageDepRate*
- *baseVoltageDepSynapse*
- *baseVoltageDepTime*
- *baseVoltageDepVariable*
- *biophysicalProperties*
- *biophysicalProperties2CaPools*
- *blockingPlasticSynapse*
- *bqbiol_encodes*
- *bqbiol_hasPart*
- *bqbiol_hasProperty*
- *bqbiol_hasTaxon*
- *bqbiol_hasVersion*
- *bqbiol_is*
- *bqbiol_isDescribedBy*
- *bqbiol_isEncodedBy*
- *bqbiol_isHomologTo*
- *bqbiol_isPartOf*
- *bqbiol_isPropertyOf*
- *bqbiol_isVersionOf*
- *bqbiol_occursIn*
- *bqmodel_is*
- *bqmodel_isDerivedFrom*
- *bqmodel_isDescribedBy*
- *capacitance*
- *cell*
- *cell2CaPools*
- *channelDensity*
- *channelDensityGHK*
- *channelDensityGHK2*
- *channelDensityNernst*
- *channelDensityNernstCa2*
- *channelDensityNonUniform*
- *channelDensityNonUniformGHK*
- *channelDensityNonUniformNernst*
- *channelDensityVShift*

- *channelPopulation*
- *channelPopulationNernst*
- *charge*
- *charge_per_mole*
- *closedState*
- *compoundInput*
- *compoundInputDL*
- *concentration*
- *concentrationModel*
- *conductance*
- *conductance_per_voltage*
- *conductanceDensity*
- *connection*
- *connectionWD*
- *continuousConnection*
- *continuousConnectionInstance*
- *continuousConnectionInstanceW*
- *continuousProjection*
- *current*
- *currentDensity*
- *decayingPoolConcentrationModel*
- *Display*
- *distal*
- *distalDetails*
- *doubleSynapse*
- *EIF_cond_alpha_isfa_ista*
- *EIF_cond_exp_isfa_ista*
- *electricalConnection*
- *electricalConnectionInstance*
- *electricalConnectionInstanceW*
- *electricalProjection*
- *EventOutputFile*
- *EventSelection*
- *expCondSynapse*
- *expCurrSynapse*
- *explicitConnection*

- *explicitInput*
- *expOneSynapse*
- *expThreeSynapse*
- *expTwoSynapse*
- *fitzHughNagumoCell*
- *fixedFactorConcentrationModel*
- *fixedFactorConcentrationModelTraub*
- *fixedTimeCourse*
- *forwardTransition*
- *from*
- *gapJunction*
- *gate*
- *gateFractional*
- *gateHHInstantaneous*
- *gateHHrates*
- *gateHHratesInf*
- *gateHHratesTau*
- *gateHHratesTauInf*
- *gateHHtauInf*
- *gateKS*
- *gradedSynapse*
- *HH_cond_exp*
- *HHExpLinearRate*
- *HHExpLinearVariable*
- *HHExpRate*
- *HHExpVariable*
- *HHSigmoidRate*
- *HHSigmoidVariable*
- *hindmarshRose1984Cell*
- *iafCell*
- *iafRefCell*
- *iafTauCell*
- *iafTauRefCell*
- *idealGasConstantDims*
- *IF_cond_alpha*
- *IF_cond_exp*

- *IF_curr_alpha*
- *IF_curr_exp*
- *include*
- *IncludeType*
- *inhomogeneousParameter*
- *inhomogeneousValue*
- *initMembPotential*
- *input*
- *inputList*
- *inputW*
- *instance*
- *intracellularProperties*
- *intracellularProperties2CaPools*
- *ionChannel*
- *ionChannelHH*
- *ionChannelKS*
- *ionChannelPassive*
- *ionChannelVShift*
- *izhikevich2007Cell*
- *izhikevichCell*
- *KSState*
- *KSTransition*
- *length*
- *Line*
- *linearGradedSynapse*
- *location*
- *member*
- *membraneProperties*
- *membraneProperties2CaPools*
- *Meta*
- *morphology*
- *network*
- *networkWithTemperature*
- *NeuroMLDocument*
- *notes*
- *openState*

- *OutputColumn*
- *OutputFile*
- *parent*
- *path*
- *per_time*
- *per_voltage*
- *permeability*
- *pinskyRinzelCA3Cell*
- *point3DWithDiam*
- *pointCellCondBased*
- *pointCellCondBasedCa*
- *poissonFiringSynapse*
- *population*
- *populationList*
- *projection*
- *property*
- *proximal*
- *proximalDetails*
- *pulseGenerator*
- *pulseGeneratorDL*
- *q10ConductanceScaling*
- *q10ExpTemp*
- *q10Fixed*
- *rampGenerator*
- *rampGeneratorDL*
- *rdf_Bag*
- *rdf_Description*
- *rdf_li*
- *rdf_RDF*
- *rectangularExtent*
- *region*
- *resistance*
- *resistivity*
- *reverseTransition*
- *rho_factor*
- *segment*

- *segmentGroup*
- *silentSynapse*
- *Simulation*
- *sineGenerator*
- *sineGeneratorDL*
- *species*
- *specificCapacitance*
- *spike*
- *spikeArray*
- *spikeGenerator*
- *spikeGeneratorPoisson*
- *spikeGeneratorRandom*
- *spikeGeneratorRefPoisson*
- *SpikeSourcePoisson*
- *spikeThresh*
- *stdpSynapse*
- *subGate*
- *substance*
- *subTree*
- *synapticConnection*
- *synapticConnectionWD*
- *tauInfTransition*
- *temperature*
- *time*
- *timedSynapticInput*
- *to*
- *transientPoissonFiringSynapse*
- *tsodyksMarkramDepFacMechanism*
- *tsodyksMarkramDepMechanism*
- *variableParameter*
- *vHalfTransition*
- *voltage*
- *voltageClamp*
- *voltageClampTriple*
- *voltageConcDepBlockMechanism*
- *volume*

13.2 NeuroML v1

> **Warning**
>
> NeuroML v1.x is deprecated. This page is maintained for archival purposes only.
>
> Please use *NeuroML v2*.
>
> *neuroConstruct* can be used for converting NeuroML v1 models into NeuroML v2.

There are three Levels of compliance to the NeuroML v1 specifications:

13.2.1 Level 1

- Metadata v1.8.1
- MorphML v1.8.1

Any Level 1 NeuroML v1 file will also be compliant to this schema.

13.2.2 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.

13.2.3 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.

13.3 LEMS

The current version of the LEMS specification is 0.7.6 and the schema for this can be seen here. The following figure, taken from Cannon et al. 2014 ([CGC+14]) shows the structure of LEMS models. The following pages give details of all the elements that are included in LEMS. For examples on LEMS, and using LEMS to extend NeuroML, please see the relevant sections in the documentation.

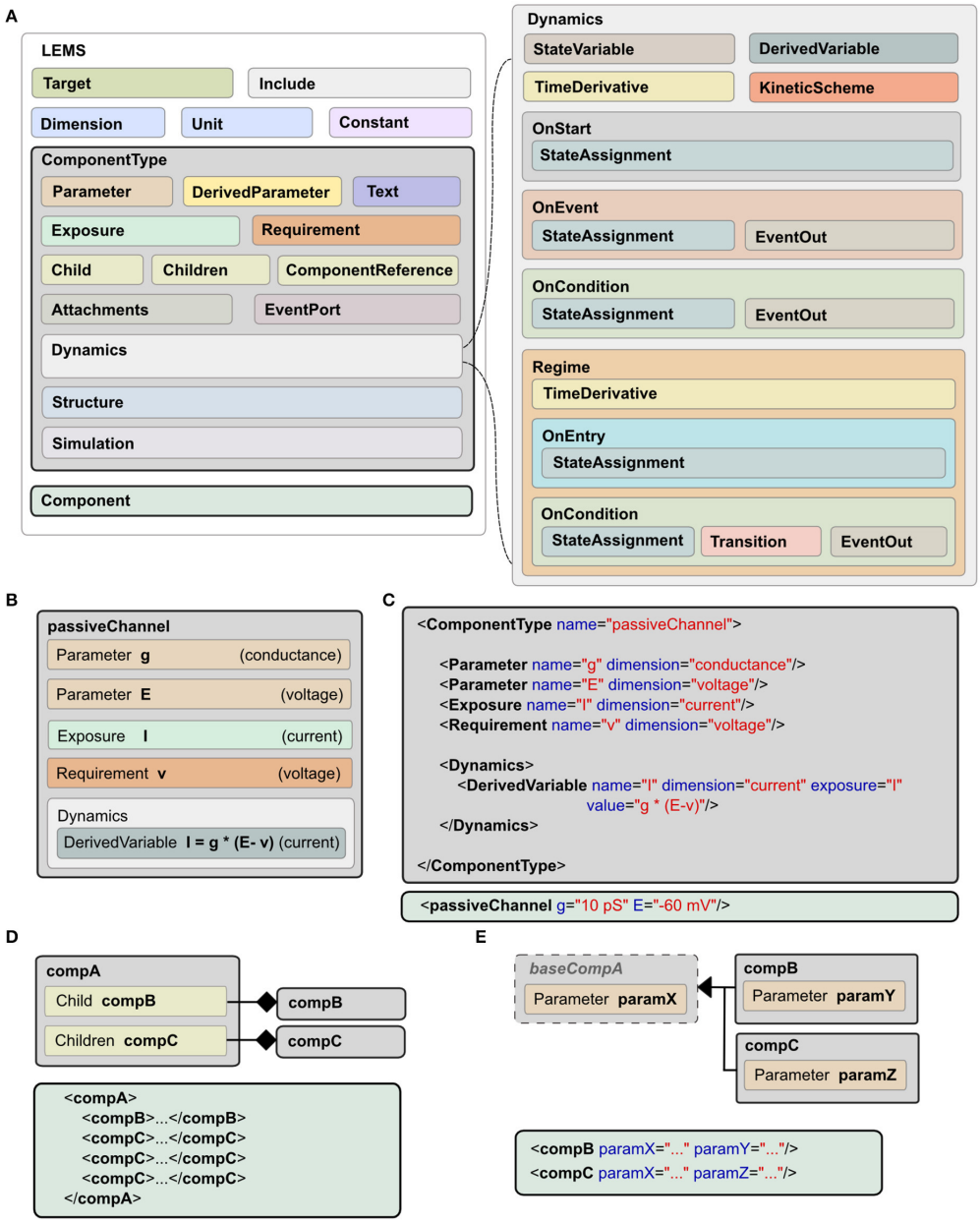


Fig. 13.2: *(A)* Models in LEMS are specified using ComponentType definitions with nested Dynamics elements. Any Parameter or StateVariable declaration must refer to a Dimension element defined at the top level. A Component element sets parameter values for a particular instance of a ComponentType. Each Parameter value must refer to one of the Unit elements defined at the top level. The Dynamics element supports continuous time systems defined in terms of first order differential equations, and event driven processing as specified by the various "On…" elements. Multiple Regimes, each with independent TimeDerivative expressions can be defined, along with the rules to transition between them. *(B)* Example of a ComponentType, the passive channel model from Figure 1. © The XML equivalent of the ComponentType (top) and Component (bottom) for this model. *(D)* Defining containment in LEMS, using Child (exactly one sub element of the given type) or Children (zero or multiple copies). **(E)** Extension in LEMS. Extending ComponentTypes inherit the structure of the base type. Example Components in XML are shown in *(D,E)*.

13.3.1 Model structure

Models can be spread over multiple files. The root element in each file is Lems.

Schema against which LEMS based on these should be valid: LEMS_v0.7.6.xsd. Generated on 18/06/24 from this commit. Please file any issues or questions at the issue tracker here.

Lems

Root element for any lems content

can contain these elements

dimensions	*Dimension*
constants	*Constant*
units	*Unit*
assertions	*Assertion*
componentTypes	*ComponentType*
components	*Component*
targets	*Target*

Target

A lems file can contain many component definitions. A Target elements specifies that a components should be treated as the entry point for simulation or other processing

Properties

component	String	Reference to the entry point component
reportFile	String	Optional attribute specifying file in which to save short report of simulation
timesFile	String	Optional attribute specifying file in which to save times used in simulation

Schema

```
<xs:complexType name="Target">
  <xs:attribute name="component" type="xs:string" use="required"/>
  <xs:attribute name="reportFile" type="xs:string" use="optional">
    <xs:annotation>
      <xs:documentation>jLEMS only optional attribute to also write a short report␣
↪with simulation duration, version, etc.</xs:documentation>
```

(continues on next page)

(continued from previous page)

```
    </xs:annotation>
  </xs:attribute>
  <xs:attribute name="timesFile" type="xs:string" use="optional">
    <xs:annotation>
      <xs:documentation>jLEMS only optional attribute to also write a file containing␣
↪actual times used in the simulation.</xs:documentation>
    </xs:annotation>
  </xs:attribute>
</xs:complexType>
```

Usage: XML

```
<Target component="sim1"/>
```

```
<Target component="sim1"/>
```

```
<Target component="sim1"/>
```

```
<Target component="sim1"/>
```

```
<Target component="sim1"/>
```

Constant

A constant quantity: like a parameter for which the value is supplied in the class definition itself rather than when a component is defined.

Properties

name	String	A readable name for the constant.
symbol	String	The symbol used in expressions to refer to this constant.
value	String	The value of a constant must be a plain number (no units) giving the SI magnitude of the quantity or an expression involving only plain numbers or other constants.
dimension	String	

Schema

```
<xs:complexType name="Constant">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="dimension" type="xs:string" use="optional" default="none"/>
  <xs:attribute name="value" type="PhysicalQuantity" use="required"/>
  <xs:attribute name="description" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<Constant name="kte" dimension="voltage" value="25.3mV"/>
```


```
<Constant name="kte" dimension="voltage" value="25.3mV"/>
```


Include

Include LEMS files in other LEMS files. Files are included where the Include declaration occurs. The enclosing Lems block is stripped off and the rest of the content included as is

Properties

file	String	the name or relative path of a file to be included

Schema

```
<xs:complexType name="Include">
  <xs:attribute name="file" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: XML

```
<Include file="SimpleNetwork.xml"/>
```

```
<Include file="SingleSimulation.xml"/>
```

```
<Include file="ex2dims.xml"/>
```

```
<Include file="hhchannel.xml"/>
```

```
<Include file="hhcell.xml"/>
```

13.3.2 Units and dimensions

Schema against which LEMS based on these should be valid: LEMS_v0.7.6.xsd. Generated on 18/06/24 from this commit. Please file any issues or questions at the issue tracker here.

Dimension

A Dimenson element associated a name with a particular combination of the standards SI base dimensions, mass, lenght, time, current, temperature and amount if substance (moles). Fractional dimensions are not currently supported.

Properties

name	String	The name to be used when referring to this dimension from variable declaration or units
m	int	Mass
l	int	Length
t	int	Time
i	int	Current
k	int	Temperature
n	int	Amount of substance
j	int	Luminous intensity

Schema

```
<xs:complexType name="Dimension">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="m" type="xs:integer" use="optional" default="0"/>
  <xs:attribute name="l" type="xs:integer" use="optional" default="0"/>
  <xs:attribute name="t" type="xs:integer" use="optional" default="0"/>
  <xs:attribute name="i" type="xs:integer" use="optional" default="0"/>
  <xs:attribute name="k" type="xs:integer" use="optional" default="0"/>
  <xs:attribute name="n" type="xs:integer" use="optional" default="0"/>
</xs:complexType>
```

Usage: XML

```
<Dimension name="voltage" m="1" l="2" t="-3" i="-1"/>
```

```
<Dimension name="time" t="1"/>
```

```
<Dimension name="conductance" m="-1" l="-2" t="3" i="2"/>
```

```
<Dimension name="capacitance" m="-1" l="-2" t="4" i="2"/>
```

```
<Dimension name="current" i="1"/>
```

Unit

A Unit asociates a symbol with a dimension and a power of ten. For non-metric units a scale can be provided, as in '1 inch = 0.0254 m'. In this case there is a degeneracy between the power and the scale which is best resolved by not using the two together. The offset parameter is available for units which are not zero-offset, such as farenheit.

Properties

name	String	As with constants, units are only referred to within expressions using their symbols, so the name is just for readability.
symbol	String	The symbol is used to refer to this unit inside compound expressions coutaining a number and a unit symbol. Such expressions can only occur on the right hand side of assignments statements.
dimension	String	Reference to the dimension for this unit
power	int	Power of ten
scale	double	Scale, only to be used for scales which are not powers of ten
offset	double	Offset for non zero-offset units

Schema

```
<xs:complexType name="Unit">
  <xs:attribute name="symbol" type="xs:string" use="required"/>
  <xs:attribute name="dimension" type="xs:string" use="required"/>
  <xs:attribute name="power" type="xs:integer" use="optional" default="0">
    <xs:annotation>
      <xs:documentation>Some have asked whether fractional dimensions should be␣
↪allowed. Disallowing it until needed...</xs:documentation>
    </xs:annotation>
  </xs:attribute>
  <xs:attribute name="scale" type="xs:float" use="optional" default="1"/>
  <xs:attribute name="offset" type="xs:float" use="optional" default="0"/>
</xs:complexType>
```

Usage: XML

```
<Unit symbol="mV" dimension="voltage" power="-3"/>
```

```
<Unit symbol="ms" dimension="time" power="-3"/>
```

```
<Unit symbol="pS" dimension="conductance" power="-12"/>
```

```
<Unit symbol="nS" dimension="conductance" power="-9"/>
```

```
<Unit symbol="uF" dimension="capacitance" power="-6"/>
```

Assertion

Assertions are not strictly part of the model, but can be included in a file as a consistency check.

Properties

dimension	String	The name of a dimension
matches	String	An expression involving dimensions. The dimensionality of the expression should match the dimensionality of the dimension reference.

13.3.3 Defining component types

Schema against which LEMS based on these should be valid: LEMS_v0.7.6.xsd. Generated on 18/06/24 from this commit. Please file any issues or questions at the issue tracker here.

ComponentType

Root element for defining LEMS Component Types.

Properties

name	String	The name of the component type. This can be uses as an XML element name in the shorthand form whendefining components.
eXtends	String	The component type that this type inherits field definitions for, if any

can contain these elements

parameters	*Parameter*
indexParameters	*IndexParameter*
derivedParameters	*DerivedParameter*
pathParameters	*PathParameter*
requirements	*Requirement*
componentRequirements	*ComponentRequirement*
instanceRequirements	*InstanceRequirement*
exposures	*Exposure*
childs	*Child*
childrens	*Children*
links	*Link*
componentReferences	*ComponentReference*

continues on next page

Table 13.1 – continued from previous page

componentTypeReferences	*ComponentTypeReference*
locations	*Location*
propertys	*Property*
dynamicses	*Dynamics*
structures	*Structure*
simulations	*Simulation*
equilibriums	*Equilibrium*
procedures	*Procedure*
geometrys	*Geometry*
fixeds	*Fixed*
constants	*Constant*
attachmentses	*Attachments*
eventPorts	*EventPort*
paths	*Path*
texts	*Text*
collections	*Collection*
pairCollections	*PairCollection*
abouts	*About*
metas	*Meta*

Schema

```
<xs:complexType name="ComponentType">
  <xs:sequence>
    <xs:element name="Property" type="Property" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="Parameter" type="Parameter" minOccurs="0" maxOccurs="unbounded"/
↪>
    <xs:element name="DerivedParameter" type="DerivedParameter" minOccurs="0"␣
↪maxOccurs="unbounded"/>
    <xs:element name="IndexParameter" type="IndexParameter" minOccurs="0" maxOccurs=
↪"unbounded"/>
    <xs:element name="Constant" type="Constant" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="Child" type="Child" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="Children" type="Children" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="Fixed" type="Fixed" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="Link" type="Link" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="ComponentReference" type="ComponentReference" minOccurs="0"␣
↪maxOccurs="unbounded"/>
    <xs:element name="Attachments" type="Attachments" minOccurs="0" maxOccurs=
↪"unbounded"/>
    <xs:element name="EventPort" type="EventPort" minOccurs="0" maxOccurs="unbounded"/
↪>
    <xs:element name="Exposure" type="Exposure" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="Requirement" type="Requirement" minOccurs="0" maxOccurs=
↪"unbounded"/>
    <xs:element name="ComponentRequirement" type="ComponentRequirement" minOccurs="0"␣
↪maxOccurs="unbounded"/>
    <xs:element name="InstanceRequirement" type="InstanceRequirement" minOccurs="0"␣
↪maxOccurs="unbounded"/>
    <xs:element name="Path" type="Path" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="Text" type="Text" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="Dynamics" type="Dynamics" minOccurs="0" maxOccurs="unbounded"/>
```

(continues on next page)

(continued from previous page)

```
    <xs:element name="Structure" type="Structure" minOccurs="0" maxOccurs="1"/>
    <xs:element name="Simulation" type="Simulation" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="extends" type="xs:string" use="optional"/>
  <xs:attribute name="description" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<ComponentType name="Population">
    <ComponentReference name="component" type="Component"/>
    <Parameter name="size" dimension="none"/>
    <Structure>
        <MultiInstantiate number="size" component="component"/>
    </Structure>
</ComponentType>
```

```
<ComponentType name="EventConnectivity">
    <Link name="source" type="Population"/>
    <Link name="target" type="Population"/>
    <Child name="Connections" type="ConnectionPattern"/>
</ComponentType>
```

```
<ComponentType name="Network">
    <Children name="populations" type="Population"/>
    <Children name="connectivities" type="EventConnectivity"/>
</ComponentType>
```

```
<ComponentType name="AllAll" extends="ConnectionPattern">
    <Structure>
        <ForEach instances="../source" as="a">
            <ForEach instances="../target" as="b">
                <EventConnection from="a" to="b"/>
            </ForEach>
        </ForEach>
    </Structure>
</ComponentType>
```

```
<ComponentType name="ConnectionPattern"/>
```

Parameter

A quantity, defined by name and dimension, that must be supplied when a Component of the enclosing ComponentType is defined

Properties

name	String	The name of the parameter. This is the name of the attribute to be used when the parameter is supplied in a component definition
dimension	String	The dimension, or 'none'. This should be the name of an already defined dimension element
description	String	An optional description of the parameter

Schema

```
<xs:complexType name="Parameter">
  <xs:complexContent>
    <xs:extension base="NamedDimensionalType"/>
  </xs:complexContent>
</xs:complexType>
```

Usage: XML

```
<Parameter name="size" dimension="none"/>
```

```
<Parameter name="xmin" dimension="none"/>
```

```
<Parameter name="xmax" dimension="none"/>
```

```
<Parameter name="ymin" dimension="none"/>
```

```
<Parameter name="ymax" dimension="none"/>
```

PathParameter

A parameter of which the value is a path expression. When a ComponentType declares a PathParameter, a corresponding Component definition should have an attibute with that name whose value is a path expression that evaluates within the instance tree of the built model. This is used, for example, in the definition of a group component class, where the coresponding component specifies a path over the instance tree which selectesthe items that should go in the group.

Properties

name	String	Name of the parameter

Property

An property on an instance of a component. Unlike a Parameter, a Property can very from instance to instance. It should be set with an Assign element within the build specification.

Properties

name	String	
dimension	String	
defaultValue	String	The defaultValue for the property must be a plain number (no units) giving the SI magnitude of the quantity.

Schema

```
<xs:complexType name="Property">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="dimension" type="xs:string" use="optional" default="none"/>
  <xs:attribute name="description" type="xs:string" use="optional"/>
  <xs:attribute name="defaultValue" type="xs:double" use="optional"/>
</xs:complexType>
```

DerivedParameter

A parameter that is a function of the Component's Parameters, which does not change with time. Its value can be supplied either with the 'value' attribute that evaluates within the scope of the definition, or with the 'select' attribute which gives a path to 'primary' version of the parameter. For example, setting select='//MembranePotential[species=channel/species]/reversal' within the appropriate context allows a channel's reversal potential to taken from a single global setting according to its permeant ion, rather than explicitly supplied locally.

Properties

name	String	The name of the derived parameter
dimension	String	The dimension, or 'none'. This should be the name of an already defined dimension element
description	String	An optional description of the derived parameter
select	String	Path to the parameter that supplies the value. Exactly one of 'select' and 'value' is required.
value	String	A string defining the value of the element

Schema

```
<xs:complexType name="DerivedParameter">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="dimension" type="xs:string" use="optional" default="none"/>
  <xs:attribute name="value" type="xs:string" use="required"/>
  <xs:attribute name="description" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<DerivedParameter name="erev" dimension="voltage" select="//
↪MembranePotential[species=channel/species]/reversal"/>
```

Fixed

Fixes the value of a parameter in the parent class, so that it does not have to be supplied separately in component definitions.

Properties

parameter	String
value	String

Schema

```
<xs:complexType name="Fixed">
  <xs:attribute name="parameter" type="xs:string" use="required"/>
  <xs:attribute name="value" type="PhysicalQuantity" use="required"/>
  <xs:attribute name="description" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<Fixed parameter="threshold" value="-45mV"/>
```

```
<Fixed parameter="relativeConductance" value="0"/>
```

```
<Fixed parameter="relativeConductance" value="1"/>
```

```
<Fixed parameter="relativeConductance" value="0"/>
```

```
<Fixed parameter="relativeConductance" value="1"/>
```

Requirement

A Requirement gives the name and dimension of a quantity (parameter or variable) that should be accessible within the scope of a model component. This is only applicable for elements that can be included as children of other elements, where the scope comprises its own parameters and those in the scope of its enclosing element. Once a requirement has been declared, then the quantity can be used within the Dynamics definition of the component. It is the responsibility of an implementation to check that the component is only used in a context in which the requirement is met. A typical example is in defining membrand bound components which require access to the membrane potential but where the variable that holds the potential itself is defined in the top level component.

Properties

name	String	name
dimension	String	The dimension, or 'none'. This should be the name of an already defined dimension element
description	String	An optional description of the requirement

Schema

```
<xs:complexType name="Requirement">
  <xs:complexContent>
    <xs:extension base="NamedDimensionalType"/>
  </xs:complexContent>
</xs:complexType>
```

Usage: XML

```
<Requirement name="v" dimension="voltage"/>
```

```
<Requirement name="v" dimension="voltage"/>
```

```
<Requirement name="v" dimension="voltage"/>
```

```
<Requirement name="v" dimension="voltage"/>
```

```
<Requirement name="v" dimension="voltage"/>
```

ComponentRequirement

The name of a component or component reference that must exist in the component hierarchy

Properties

name	String	name

Schema

```
<xs:complexType name="ComponentRequirement">
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
```

InstanceRequirement

An instance that must be supplied at build time. Expressions can contain references to quantities in the instance

Properties

name	String	name

Schema

```
<xs:complexType name="InstanceRequirement">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="type" type="xs:string" use="required"/>
</xs:complexType>
```

Exposure

A quantity that is made available to other components in the simulation. Note that all variables in a Dynamics definition are private. If other components need access to them, then the definition should explicitly link them to an exposure defined in the component class

Properties

name	String	Name of the exposure element
dimension	String	The dimension, or 'none'. This should be the name of an already defined dimension element
description	String	An optional description of the element

Schema

```
<xs:complexType name="Exposure">
  <xs:complexContent>
    <xs:extension base="NamedDimensionalType"/>
  </xs:complexContent>
</xs:complexType>
```

Usage: XML

```
<Exposure name="v" dimension="voltage"/>
```

```
<Exposure name="tsince" dimension="time"/>
```

```
<Exposure name="r" dimension="per_time"/>
```

```
<Exposure name="fcond" dimension="none"/>
```

```
<Exposure name="fcond" dimension="none"/>
```

Child

Specifies that a component can have a child of a particular type. The name supplied here can be used in path expressions to access the component. This is useful, for example, where a component can have multiple children of the same type but with different roles, such as the forward and reverse transition rates in a channel.

Properties

name	String	Name of the child
type	String	Reference to a component class, the value should be the name of the target class.
description	String	An optional description of the child

Schema

```
<xs:complexType name="Child">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="type" type="xs:string" use="required"/>
  <xs:attribute name="description" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<Child name="Connections" type="ConnectionPattern"/>
```

```
<Child name="Forward" type="HHRate"/>
```

```
<Child name="Reverse" type="HHRate"/>
```

```
<Child name="Forward" type="HHRate"/>
```

```
<Child name="Reverse" type="HHRate"/>
```

Children

Specifies that a component can have children of a particular class. The class may refer to an extendedtype, in which case components of any class that extends the specified target class should be valid as child components

Properties

name	String	Name of the children
type	String	The class of component allowed as children.

Schema

```
<xs:complexType name="Children">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="type" type="xs:string" use="optional"/>
  <xs:attribute name="min" type="xs:integer" use="optional"/>
  <xs:attribute name="max" type="xs:integer" use="optional"/>
  <xs:attribute name="description" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<Children name="populations" type="Population"/>
```

```
<Children name="connectivities" type="EventConnectivity"/>
```

```
<Children name="lines" type="Line"/>
```

```
<Children name="outputColumn" type="OutputColumn"/>
```

```
<Children name="displays" type="Display"/>
```

Link

Like a ComponentRef, but resolved relative to the enclosing object so the target must already be in the model. One or the other should be deprecated. The Link element has the same properties as ComponentRef. The Link element just establishes a connection with the target component, but leaves it in its existing place in the hierarchy. Variables in the target component can be accessed via the name of the link element.

Properties

name	String	A name for the ComponentReference
type	String	The type of the target Component
description	String	An optional description of the ComponentReference

Schema

```
<xs:complexType name="Link">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="type" type="xs:string" use="required"/>
  <xs:attribute name="description" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<Link name="source" type="Population"/>
```

```
<Link name="target" type="Population"/>
```

```
<Link name="from" type="KSState"/>
```

```
<Link name="to" type="KSState"/>
```

```
<Link name="from" type="KSState"/>
```

ComponentReference

A reference to another component. The target component can be accessed with path expressions in the same way as a child component, but can be defined independently

Properties

name	String	A name for the ComponentReference
type	String	The type of the target Component
description	String	An optional description of the ComponentReference

Schema

```
<xs:complexType name="ComponentReference">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="type" type="xs:string" use="required"/>
  <xs:attribute name="local" type="xs:string" use="optional"/>
  <xs:attribute name="description" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<ComponentReference name="component" type="Component"/>
```

```
<ComponentReference name="target" type="Component"/>
```

```
<ComponentReference name="channel" type="HHChannel"/>
```

```
<ComponentReference name="component" type="Component"/>
```

```
<ComponentReference name="synapse" type="Synapse"/>
```

ComponentTypeReference

This is used in conjunction with PathParameter elements to specify the target class of selections defined within components operating over the instance tree.

Properties

name	String

Collection

Specifies that instances of components based on this class can containe a named collection of other instances. This provides for containers for oprating on groups of instances with path and filter expressions defined in components to operate over the instance tree.

Properties

name	String

PairCollection

Defines a named collection of paris of instances, similar to the Collection element.

Properties

name	String

EventPort

A port on a component that can send or receive events, depending on the direction specified

Properties

name	String	Name of the EventPort
direction	String	'IN' or 'OUT'
description	String	An optional description of the EventPort

Schema

```
<xs:complexType name="EventPort">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="direction" type="xs:string" use="required"/>
  <xs:attribute name="description" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<EventPort name="spikes-in" direction="in"/>
```

```
<EventPort name="a" direction="out"/>
```

```
<EventPort name="in" direction="in"/>
```

```
<EventPort name="out" direction="out"/>
```

```
<EventPort name="in" direction="in"/>
```

Text

Holds textual information that does not change the model but is needed for other purposes such as labelling graphs.

Properties

name	String	The textual content
description	String	An optional description of the element

Schema

```
<xs:complexType name="Text">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="description" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<Text name="title"/>
```

```
<Text name="color"/>
```

```
<Text name="path"/>
```

```
<Text name="fileName"/>
```

```
<Text name="destination"/>
```

Path

Duplicates some functionality of PathParameter - the two should be merged.

Properties

name	String

Schema

```
<xs:complexType name="Path">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="description" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<Path name="quantity"/>
```

```
<Path name="quantity"/>
```

```
<Path name="from"/>
```

```
<Path name="to"/>
```

```
<Path name="quantity"/>
```

Attachments

Specifies that a component can accept attached components of a particular class. Attached components can be added at build time dependent on other events. For scoping and access purposes they are like child components. The cannonical use of attachments is in adding synapses to a cell when a network connection is made.

Properties

name	String	A name for the Attachments
type	String	The type of the Attachments

Schema

```
<xs:complexType name="Attachments">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="type" type="xs:string" use="required"/>
  <xs:attribute name="description" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<Attachments name="synapses" type="Synapse"/>
```

Insertion

IntegerParameter

Properties

name	String	The name of the parameter. This is the name of the attribute to be used when the parameter is supplied in a component definition
dimension	String	The dimension, or 'none'. This should be the name of an already defined dimension element
description	String	An optional description of the parameter

IndexParameter

Properties

name	String	The name of the parameter. This is the name of the attribute to be used when the parameter is supplied in a component definition
dimension	String	The dimension, or 'none'. This should be the name of an already defined dimension element
description	String	An optional description of the parameter

Schema

```
<xs:complexType name="IndexParameter">
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
```

About

Meta

Meta element to provide arbitrary metadata to LEMS simulations. Note that this is not processed by the LEMS interpreter.

Schema

```
<xs:complexType name="Meta">
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="unbounded" processContents="lax"/>
  </xs:sequence>
  <xs:anyAttribute processContents="skip"/>
</xs:complexType>
```

13.3.4 Dynamics

Schema against which LEMS based on these should be valid: LEMS_v0.7.6.xsd. Generated on 18/06/24 from this commit. Please file any issues or questions at the issue tracker here.

Dynamics

Specifies the dynamical behavior of components build from this ComponentType. Note that all variables in a Dynamics definition are private. If other components need access to them, then the definition should explicitly link them to an Exposure defined in the component class

can contain these elements

supers	*Super*
derivedVariables	*DerivedVariable*
conditionalDerivedVariables	*ConditionalDerivedVariable*
stateVariables	*StateVariable*
timeDerivatives	*TimeDerivative*
kineticSchemes	*KineticScheme*
onStarts	*OnStart*
onEvents	*OnEvent*
onConditions	*OnCondition*
stateScalarFields	*StateScalarField*
derivedScalarFields	*DerivedScalarField*
derivedPunctateFields	*DerivedPunctateField*
regimes	*Regime*

Schema

```
<xs:complexType name="Dynamics">
  <xs:sequence>
    <xs:element name="StateVariable" type="StateVariable" minOccurs="0" maxOccurs=
↪"unbounded"/>
    <xs:element name="DerivedVariable" type="DerivedVariable" minOccurs="0" maxOccurs=
↪"unbounded"/>
    <xs:element name="ConditionalDerivedVariable" type="ConditionalDerivedVariable"␣
↪minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="TimeDerivative" type="TimeDerivative" minOccurs="0" maxOccurs=
↪"unbounded"/>
    <xs:element name="OnStart" type="OnStart" minOccurs="0" maxOccurs="1"/>
    <xs:element name="OnEvent" type="OnEvent" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="OnCondition" type="OnCondition" minOccurs="0" maxOccurs=
↪"unbounded"/>
    <xs:element name="Regime" type="Regime" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="KineticScheme" type="KineticScheme" minOccurs="0" maxOccurs="1"/
↪>
  </xs:sequence>
</xs:complexType>
```

Usage: XML

```
<Dynamics>
    <StateVariable name="t" dimension="time"/>
</Dynamics>
```

```
<Dynamics>
    <StateVariable name="x" dimension="none"/>
    <DerivedVariable name="ex" dimension="none" value="exp(x)"/>
    <DerivedVariable name="q" dimension="none" value="ex / (1 + ex)"/>
```

(continues on next page)

(continued from previous page)

```
    <DerivedVariable name="rf" dimension="per_time" select="Forward/r"/>
    <DerivedVariable name="rr" dimension="per_time" select="Reverse/r"/>
    <TimeDerivative variable="x" value="(1 + ex)^2 / ex * (rf * (1 - q) - rr * q)"/>
    <DerivedVariable name="fcond" dimension="none" exposure="fcond" value="q^power"/>
</Dynamics>
```

```
<Dynamics>
    <StateVariable name="q" dimension="none"/>
    <DerivedVariable dimension="per_time" name="rf" select="Forward/r"/>
    <DerivedVariable dimension="per_time" name="rr" select="Reverse/r"/>
    <TimeDerivative variable="q" value="rf * (1 - q) - rr * q"/>
    <DerivedVariable name="fcond" dimension="none" exposure="fcond" value="q^power"/>
</Dynamics>
```

```
<Dynamics>
    <OnStart>
        <StateAssignment variable="v" value="v0"/>
    </OnStart>
    <DerivedVariable name="totcurrent" dimension="current" select="populations[*]/
 ↪current" reduce="add"/>
    <StateVariable name="v" exposure="v" dimension="voltage"/>
    <TimeDerivative variable="v" value="(totcurrent + injection) / capacitance"/>
</Dynamics>
```

```
<Dynamics>
    <StateVariable name="v" exposure="v" dimension="voltage"/>
    <TimeDerivative variable="v" value="leakConductance * (leakReversal - v) /␣
 ↪capacitance"/>
    <OnEvent port="spikes-in">
        <StateAssignment variable="v" value="v + deltaV"/>
    </OnEvent>
</Dynamics>
```

StateVariable

Specifies a state variable that stores values that define the state of the system at a point in time. The dynamics of state variables may be defined as Ordinary Differential Equations (ODEs) using the *TimeDerivative* entity. They can also be discontinuously modified using elements in *Dynamics* such as *OnStart*s, *OnEvent*s, and *OnCondition*s. StateVariables may be linked to exposures.

Properties

name	String	Name of the state variable
dimension	String	The dimension, or 'none'. This should be the name of an already defined dimension element
exposure	String	If this variable is to be accessed from outside, it should be linked to an Exposure that is defined in the ComponentType.
description	String	An optional description of the state variable

Schema

```
<xs:complexType name="StateVariable">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="dimension" type="xs:string" use="optional" default="none"/>
  <xs:attribute name="exposure" type="xs:string" use="optional"/>
  <xs:attribute name="description" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<StateVariable name="t" dimension="time"/>
```

```
<StateVariable name="v" exposure="v" dimension="voltage"/>
```

```
<StateVariable name="tsince" exposure="tsince" dimension="time"/>
```

```
<StateVariable name="tlast" dimension="time"/>
```

```
<StateVariable name="q" dimension="none"/>
```

StateAssignment

Has 'variable' and 'value' fields

Properties

variable	String	The name of the variable
value	String	A string defining the value of the element

Schema

```
<xs:complexType name="StateAssignment">
  <xs:attribute name="variable" type="xs:string" use="required"/>
  <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: XML

```
<StateAssignment variable="v" value="v + deltaV"/>
```

```
<StateAssignment variable="tsince" value="0"/>
```

```
<StateAssignment variable="tlast" value="t"/>
```

```
<StateAssignment variable="v" value="v0"/>
```

```
<StateAssignment variable="geff" value="0"/>
```

TimeDerivative

First order differential equations, functions of StateVariables and Parameters, for how StateVariables change with time. Has a variable and a value. The value is the rate of change of the variable.

Properties

variable	String	The name of the variable
value	String	A string defining the value of the element

Schema

```
<xs:complexType name="TimeDerivative">
  <xs:attribute name="variable" type="xs:string" use="required"/>
  <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: XML

```
<TimeDerivative variable="v" value="leakConductance * (leakReversal - v) / capacitance
↪"/>
```

```
<TimeDerivative variable="tsince" value="1"/>
```

```
<TimeDerivative variable="q" value="rf * (1 - q) - rr * q"/>
```

```
<TimeDerivative variable="x" value="(1 + ex)^2 / ex * (rf * (1 - q) - rr * q)"/>
```

```
<TimeDerivative variable="v" value="(totcurrent + injection) / capacitance"/>
```

DerivedVariable

A quantity that depends algebraically on other quantities in the model. The 'value' field can be set to a mathematical expression, or the 'select' field to a path expression. If the path expression produces multiple matches, then the 'reduce' field says how these are reduced to a single value by taking the sum or product.

Properties

name	String	Name of the derived variable
select	String	A path to the variable that supplies the value. Note that to select a variable from another component, the variable must be marked as an Exposure. Exactly one of 'select' and 'value' is required
dimension	String	The dimension, or 'none'. This should be the name of an already defined dimension element
description	String	An optional description of the derived variable
reduce	String	Either 'add' or 'multiply'. This applies if ther are multiple matches to the path or if 'required' is false. In the latter case, for multiply mode, multiplicative expressions in this variable behave as if the term was absent. Additive expressions generate an error. Conversely, if set to 'add' then additive expressions behave as if it was not there and multiplicative ones generateand error.
exposure	String	
required	boolean	Set to true if it OK for this variable to be absent. See 'reduce' for what happens in this case
value	String	A string defining the value of the element

Schema

```
<xs:complexType name="DerivedVariable">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="dimension" type="xs:string" use="optional" default="none"/>
  <xs:attribute name="exposure" type="xs:string" use="optional"/>
  <xs:attribute name="description" type="xs:string" use="optional"/>
  <xs:attribute name="select" type="xs:string" use="optional"/>
  <xs:attribute name="value" type="xs:string" use="optional"/>
  <xs:attribute name="reduce" type="xs:string" use="optional"/>
  <xs:attribute name="required" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<DerivedVariable name="tsince" dimension="time" exposure="tsince" value="t - tlast"/>
```

```
<DerivedVariable name="r" dimension="per_time" exposure="r" value="rate * exp((v -␣
↪midpoint)/scale)"/>
```

```
<DerivedVariable name="r" dimension="per_time" exposure="r" value="rate / (1 + exp( -
↪(v - midpoint)/scale))"/>
```

```
<DerivedVariable name="x" dimension="none" value="(v - midpoint) / scale"/>
```

```
<DerivedVariable name="r" dimension="per_time" exposure="r" value="rate * x / (1 -␣
↪exp(-x))"/>
```

OnStart

can contain these elements

stateAssignments	*StateAssignment*
eventOuts	*EventOut*
transitions	*Transition*

Schema

```
<xs:complexType name="OnStart">
  <xs:sequence>
    <xs:element name="StateAssignment" type="StateAssignment" minOccurs="1" maxOccurs=
↪"unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Usage: XML

```
<OnStart>
    <StateAssignment variable="v" value="v0"/>
</OnStart>
```

```
<OnStart>
    <StateAssignment variable="geff" value="0"/>
</OnStart>
```

```
<OnStart>
    <StateAssignment variable="v" value="v0"/>
</OnStart>
```

```
<OnStart>
    <StateAssignment variable="v" value="v0"/>
</OnStart>
```

```
<OnStart>
    <StateAssignment variable="v" value="v0"/>
</OnStart>
```

OnCondition

can contain these elements

stateAssignments	*StateAssignment*
eventOuts	*EventOut*
transitions	*Transition*

Schema

```
<xs:complexType name="OnCondition">
  <xs:sequence>
    <xs:element name="StateAssignment" type="StateAssignment" minOccurs="0" maxOccurs=
↪"unbounded"/>
    <xs:element name="EventOut" type="EventOut" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="Transition" type="Transition" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="test" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: XML

```
<OnCondition test="tsince .gt. period">
    <StateAssignment variable="tsince" value="0"/>
    <EventOut port="a"/>
</OnCondition>
```

```
<OnCondition test="t - tlast .gt. period">
    <StateAssignment variable="tlast" value="t"/>
    <EventOut port="a"/>
</OnCondition>
```

```
<OnCondition test="v .gt. threshold">
    <EventOut port="out"/>
    <Transition regime="refr"/>
</OnCondition>
```

```
<OnCondition test="t .gt. tin + refractoryPeriod">
    <Transition regime="int"/>
</OnCondition>
```

```
<OnCondition test="tsince .gt. period">
    <StateAssignment variable="tsince" value="0"/>
    <EventOut port="a"/>
</OnCondition>
```

OnEvent

Event handler block

Properties

port	String	the port to listen on

can contain these elements

stateAssignments	*StateAssignment*
eventOuts	*EventOut*
transitions	*Transition*

Schema

```
<xs:complexType name="OnEvent">
  <xs:sequence>
    <xs:element name="StateAssignment" type="StateAssignment" minOccurs="0" maxOccurs=
↪"unbounded"/>
    <xs:element name="EventOut" type="EventOut" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="port" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: XML

```
<OnEvent port="spikes-in">
    <StateAssignment variable="v" value="v + deltaV"/>
</OnEvent>
```

```
<OnEvent port="in">
    <StateAssignment variable="geff" value="geff + deltaG"/>
</OnEvent>
```

```
<OnEvent port="in">
    <StateAssignment variable="v" value="v + deltaV"/>
</OnEvent>
```

```
<OnEvent port="spikes-in">
    <StateAssignment variable="v" value="v + deltaV"/>
</OnEvent>
```

EventOut

Schema

```
<xs:complexType name="EventOut">
  <xs:attribute name="port" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: XML

```
<EventOut port="a"/>
```

```
<EventOut port="a"/>
```

```
<EventOut port="out"/>
```

```
<EventOut port="a"/>
```

```
<EventOut port="a"/>
```

KineticScheme

Allows the specification of systems that can be in one of a small number of states at any time with probabilistic transitions between states. This includes continuous time Markov processes as are used for stochastic models of ion channels. A kinetic scheme does not itself introduce any new elements or state variables. It is rather a way of connecting quantities in existing components by saying that quantities in the edge elements should be interpreted as transition rates among quantities in the node elements.

Properties

name	String	Name of kinetic scheme
nodes	String	Source of notes for scheme
edges	String	The element that provides the transitions for the scheme
stateVariable	String	Name of state variable in state elements
edgeSource	String	The name of the attribute in the rate element that defines the source of the transition
edgeTarget	String	Attribute tha defines the target
forwardRate	String	Name of forward rate exposure
reverseRate	String	Name of reverse rate exposure

Schema

```
<xs:complexType name="KineticScheme">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="nodes" type="xs:string" use="required"/>
  <xs:attribute name="stateVariable" type="xs:string" use="required"/>
  <xs:attribute name="edges" type="xs:string" use="required"/>
  <xs:attribute name="edgeSource" type="xs:string" use="required"/>
  <xs:attribute name="edgeTarget" type="xs:string" use="required"/>
  <xs:attribute name="forwardRate" type="xs:string" use="required"/>
  <xs:attribute name="reverseRate" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: XML

```
<KineticScheme name="ks" nodes="states" stateVariable="occupancy" edges="transitions"␣
↪edgeSource="from" edgeTarget="to" forwardRate="rf" reverseRate="rr"/>
```

```
<KineticScheme name="ks" nodes="states" stateVariable="occupancy" edges="transitions"␣
↪edgeSource="from" edgeTarget="to" forwardRate="rf" reverseRate="rr" dependency="v"␣
↪step="deltaV"/>
```

Regime

Allows the dynamics of a ComponentType to be expressed via a finite state machine. Each regime has its internal dynamics, and conditions on which transitions between regimes occur are specified using the OnCondition element

Properties

name	String	The name of the regime
initial	String	'True' if this is the initial regime of the system

can contain these elements

derivedVariables	*DerivedVariable*
stateVariables	*StateVariable*
timeDerivatives	*TimeDerivative*
onStarts	*OnStart*
onEntrys	*OnEntry*
onEvents	*OnEvent*
onConditions	*OnCondition*
requiredVars	lemsschema:requiredvar_

Schema

```
<xs:complexType name="Regime">
  <xs:sequence>
    <xs:element name="TimeDerivative" type="TimeDerivative" minOccurs="0" maxOccurs=
↪"unbounded"/>
    <xs:element name="OnEntry" type="OnEntry" minOccurs="0" maxOccurs="1"/>
    <xs:element name="OnCondition" type="OnCondition" minOccurs="0" maxOccurs=
↪"unbounded"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="initial" type="TrueOrFalse" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<Regime name="int" initial="true">
    <TimeDerivative variable="v" value="(current + gleak * (vleak - v)) / capacitance
↪"/>
    <OnCondition test="v .gt. threshold">
        <EventOut port="out"/>
        <Transition regime="refr"/>
    </OnCondition>
    <OnEvent port="in">
        <StateAssignment variable="v" value="v + deltaV"/>
    </OnEvent>
</Regime>
```

```
<Regime name="refr">
    <OnEntry>
        <StateAssignment variable="tin" value="t"/>
        <StateAssignment variable="v" value="vreset"/>
    </OnEntry>
    <OnCondition test="t .gt. tin + refractoryPeriod">
        <Transition regime="int"/>
    </OnCondition>
</Regime>
```

OnEntry

can contain these elements

stateAssignments	*StateAssignment*
eventOuts	*EventOut*
transitions	*Transition*

Schema

```
<xs:complexType name="OnEntry">
  <xs:sequence>
    <xs:element name="StateAssignment" type="StateAssignment" minOccurs="1" maxOccurs=
↪"unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Usage: XML

```
<OnEntry>
    <StateAssignment variable="tin" value="t"/>
    <StateAssignment variable="v" value="vreset"/>
</OnEntry>
```

Transition

Schema

```
<xs:complexType name="Transition">
  <xs:attribute name="regime" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: XML

```
<Transition regime="int"/>
```

```
<Transition regime="refr"/>
```

Super

ConditionalDerivedVariable

Properties

name	String
dimension	String
exposure	String

can contain these elements

cases	*Case*

Schema

```
<xs:complexType name="ConditionalDerivedVariable">
  <xs:sequence>
    <xs:element name="Case" type="Case" minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="dimension" type="xs:string" use="optional" default="none"/>
  <xs:attribute name="exposure" type="xs:string" use="optional"/>
</xs:complexType>
```

Case

Properties

value	String	A string defining the value of the element

Schema

```
<xs:complexType name="Case">
  <xs:attribute name="condition" type="xs:string" use="optional"/>
  <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>
```

Equilibrium

can contain these elements

derivedVariables	*DerivedVariable*

StateScalarField

DerivedScalarField

DerivedPunctateField

13.3.5 Structure

Schema against which LEMS based on these should be valid: LEMS_v0.7.6.xsd. Generated on 18/06/24 from this commit. Please file any issues or questions at the issue tracker here.

Structure

By default, each Component in a model gives rise to a single instance of its state variables when the model is executed. The state variables are then governed by the dynamics definition in the associated ComponentType. Elements in the Structure declaration can be used to change this behavior, for example to make multiple instances of the state variables, or to instantiate a different component. A typical application for the latter would be a Component that defines a population of cells. The population Component might define the number of cells it contains but would refer to a Component defined elsewhere for the actual cell model to use.

can contain these elements

buildElements	*BuildElement*

Schema

```
<xs:complexType name="Structure">
  <xs:sequence>
    <xs:element name="ChildInstance" type="ChildInstance" minOccurs="0" maxOccurs=
 ↪"unbounded"/>
    <xs:element name="MultiInstantiate" type="MultiInstantiate" minOccurs="0"␣
 ↪maxOccurs="1"/>
    <xs:element name="ForEach" type="ForEach" minOccurs="0" maxOccurs="1"/>
    <xs:element name="With" type="With" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="Tunnel" type="Tunnel" minOccurs="0" maxOccurs="1"/>
    <xs:element name="EventConnection" type="EventConnection" minOccurs="0" maxOccurs=
 ↪"unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Usage: XML

```
<Structure>
    <MultiInstantiate number="size" component="component"/>
</Structure>
```

```
<Structure>
    <ForEach instances="../source" as="a">
        <ForEach instances="../target" as="b">
            <EventConnection from="a" to="b"/>
        </ForEach>
    </ForEach>
</Structure>
```

```
<Structure>
    <ChildInstance component="channel"/>
</Structure>
```

```
<Structure>
    <With instance="from" as="a"/>
    <With instance="to" as="b"/>
    <EventConnection from="a" to="b" receiver="synapse" receiverContainer="destination
 ↪" sourcePort="sourcePort" targetPort="targetPort"/>
</Structure>
```

```
<Structure>
    <MultiInstantiate number="size" component="component"/>
</Structure>
```

BuildElement

Base class for elements that can be used in Structures

can contain these elements

buildElements	*BuildElement*

MultiInstantiate

can contain these elements

assigns	*Assign*
buildElements	*BuildElement*

Schema

```
<xs:complexType name="MultiInstantiate">
  <xs:attribute name="component" type="xs:string" use="required"/>
  <xs:attribute name="number" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: XML

```
<MultiInstantiate number="size" component="component"/>
```

```
<MultiInstantiate number="size" component="component"/>
```

```
<MultiInstantiate number="size" component="component"/>
```

```
<MultiInstantiate number="size" component="component"/>
```

CoInstantiate

can contain these elements

assigns	*Assign*
buildElements	*BuildElement*

Assign

Schema

```
<xs:complexType name="Assign">
  <xs:attribute name="property" type="xs:string" use="required"/>
  <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>
```

Choose

can contain these elements

buildElements	*BuildElement*

ChildInstance

can contain these elements

assigns	*Assign*
buildElements	*BuildElement*

Schema

```
<xs:complexType name="ChildInstance">
  <xs:attribute name="component" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: XML

```
<ChildInstance component="channel"/>
```

```
<ChildInstance component="channel"/>
```

```
<ChildInstance component="channel"/>
```

```
<ChildInstance component="channel"/>
```

ForEach

can contain these elements

buildElements	*BuildElement*

Schema

```
<xs:complexType name="ForEach">
  <xs:sequence>
    <xs:element name="MultiInstantiate" type="MultiInstantiate" minOccurs="0"␣
↪maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="instances" type="xs:string" use="required"/>
  <xs:attribute name="as" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: XML

```
<ForEach instances="../source" as="a">
    <ForEach instances="../target" as="b">
        <EventConnection from="a" to="b"/>
    </ForEach>
</ForEach>
```

```
<ForEach instances="../target" as="b">
    <EventConnection from="a" to="b"/>
</ForEach>
```

```
<ForEach instances="../source" as="a">
    <ForEach instances="../target" as="b">
        <EventConnection from="a" to="b"/>
    </ForEach>
</ForEach>
```

```
<ForEach instances="../target" as="b">
    <EventConnection from="a" to="b"/>
</ForEach>
```

```
<ForEach instances="../source" as="a">
    <ForEach instances="../target" as="b">
        <EventConnection from="a" to="b"/>
    </ForEach>
</ForEach>
```

EventConnection

can contain these elements

assigns	*Assign*
buildElements	*BuildElement*

Schema

```
<xs:complexType name="EventConnection">
  <xs:sequence>
    <xs:element name="Assign" type="Assign" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="from" type="xs:string" use="required"/>
  <xs:attribute name="to" type="xs:string" use="required"/>
  <xs:attribute name="sourcePort" type="xs:string" use="optional"/>
  <xs:attribute name="targetPort" type="xs:string" use="optional"/>
  <xs:attribute name="receiver" type="xs:string" use="optional"/>
  <xs:attribute name="receiverContainer" type="xs:string" use="optional"/>
  <xs:attribute name="delay" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<EventConnection from="a" to="b"/>
```

```
<EventConnection from="a" to="b" receiver="synapse" receiverContainer="destination"␣
↪sourcePort="sourcePort" targetPort="targetPort"/>
```

```
<EventConnection from="a" to="b"/>
```

```
<EventConnection from="a" to="b"/>
```

Tunnel

can contain these elements

assigns	*Assign*
buildElements	*BuildElement*

Schema

```
<xs:complexType name="Tunnel">
  <xs:sequence>
    <xs:element name="Assign" type="Assign" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="endA" type="xs:string" use="required"/>
  <xs:attribute name="endB" type="xs:string" use="required"/>
  <xs:attribute name="componentA" type="xs:string" use="required"/>
  <xs:attribute name="componentB" type="xs:string" use="required"/>
</xs:complexType>
```

PairsEventConnection

can contain these elements

buildElements	*BuildElement*

PairFilter

can contain these elements

buildElements	*BuildElement*

IncludePair

can contain these elements

buildElements	*BuildElement*

With

can contain these elements

buildElements	*BuildElement*

Schema

```
<xs:complexType name="With">
  <xs:attribute name="instance" type="xs:string" use="optional"/>
  <xs:attribute name="list" type="xs:string" use="optional"/>
  <xs:attribute name="index" type="xs:string" use="optional"/>
  <xs:attribute name="as" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: XML

```
<With instance="from" as="a"/>
```

```
<With instance="to" as="b"/>
```

If

can contain these elements

buildElements	*BuildElement*

Apply

can contain these elements

buildElements	*BuildElement*

Gather

can contain these elements

buildElements	*BuildElement*

GatherPairs

can contain these elements

buildElements	*BuildElement*

13.3.6 Simulation

Schema against which LEMS based on these should be valid: LEMS_v0.7.6.xsd. Generated on 18/06/24 from this commit. Please file any issues or questions at the issue tracker here.

Simulation

can contain these elements

records	*Record*
eventRecords	*EventRecord*
runs	*Run*
dataDisplays	*DataDisplay*
dataWriters	*DataWriter*
eventWriters	*EventWriter*

Schema

```
<xs:complexType name="Simulation">
  <xs:sequence>
    <xs:element name="DataDisplay" type="DataDisplay" minOccurs="0" maxOccurs=
↪"unbounded"/>
    <xs:element name="Record" type="Record" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="EventRecord" type="EventRecord" minOccurs="0" maxOccurs=
↪"unbounded"/>
    <xs:element name="Run" type="Run" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="DataWriter" type="DataWriter" minOccurs="0" maxOccurs="unbounded
↪"/>
    <xs:element name="EventWriter" type="EventWriter" minOccurs="0" maxOccurs=
↪"unbounded"/>
    <xs:element name="Meta" type="Meta" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Usage: XML

```
<Simulation>
    <DataDisplay title="title" dataRegion="xmin,xmax,ymin,ymax"/>
</Simulation>
```

```
<Simulation>
    <Record quantity="quantity" timeScale="timeScale" scale="scale" color="color"/>
</Simulation>
```

```
<Simulation>
    <DataWriter path="path" fileName="fileName"/>
</Simulation>
```

```
<Simulation>
    <Record quantity="quantity"/>
</Simulation>
```

```
<Simulation>
    <Run component="target" variable="t" increment="step" total="length"/>
</Simulation>
```

Record

Properties

quantity	String	path to the parameter that will contain the path to the quantity to be recorded
scale	String	path to the element that defines the scale for rendering the quantity dimensionless
color	String	hex format color suggestion for how the data should be displayed

Schema

```
<xs:complexType name="Record">
  <xs:attribute name="quantity" type="xs:string" use="required"/>
  <xs:attribute name="timeScale" type="xs:string" use="optional"/>
  <xs:attribute name="scale" type="xs:string" use="optional"/>
  <xs:attribute name="color" type="xs:string" use="optional"/>
</xs:complexType>
```

Usage: XML

```
<Record quantity="quantity" timeScale="timeScale" scale="scale" color="color"/>
```

```
<Record quantity="quantity"/>
```

```
<Record quantity="quantity" timeScale="timeScale" scale="scale" color="color"/>
```

EventRecord

Properties

quantity	String	path for the component which will emit spikes to be recorded
eventPort	String	event port for the component which will emit spikes

Schema

```
<xs:complexType name="EventRecord">
  <xs:attribute name="quantity" type="xs:string" use="required"/>
  <xs:attribute name="eventPort" type="xs:string" use="required"/>
</xs:complexType>
```

DataDisplay

Schema

```
<xs:complexType name="DataDisplay">
  <xs:attribute name="title" type="xs:string" use="required"/>
  <xs:attribute name="dataRegion" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: XML

```
<DataDisplay title="title" dataRegion="xmin,xmax,ymin,ymax"/>
```


```
<DataDisplay title="title" dataRegion="xmin,xmax,ymin,ymax"/>
```


DataWriter

Schema

```
<xs:complexType name="DataWriter">
  <xs:attribute name="path" type="xs:string" use="required"/>
  <xs:attribute name="fileName" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: XML

```
<DataWriter path="path" fileName="fileName"/>
```

EventWriter

Schema

```
<xs:complexType name="EventWriter">
  <xs:attribute name="path" type="xs:string" use="required"/>
  <xs:attribute name="fileName" type="xs:string" use="required"/>
  <xs:attribute name="format" type="xs:string" use="required"/>
</xs:complexType>
```

Run

The run element provides a way to make a model runnable. It should point to the parameters that set the step size etc. The target parameters have to be dimensionally consistent.

Properties

component	String	name of the component reference that will set the component to be run
variable	String	
increment	String	path to the parameter that sets the step size
total	String	path to the parameter that sets the total span of the independent variable to be run

Schema

```
<xs:complexType name="Run">
  <xs:attribute name="component" type="xs:string" use="required"/>
  <xs:attribute name="variable" type="xs:string" use="required"/>
  <xs:attribute name="increment" type="xs:string" use="required"/>
  <xs:attribute name="total" type="xs:string" use="required"/>
</xs:complexType>
```

Usage: XML

```
<Run component="target" variable="t" increment="step" total="length"/>
```


```
<Run component="target" variable="t" increment="step" total="length"/>
```

13.3.7 Procedure

Schema against which LEMS based on these should be valid: LEMS_v0.7.6.xsd. Generated on 18/06/24 from this commit. Please file any issues or questions at the issue tracker here.

Procedure

can contain these elements

statements	lemsschema:statement_

Equilibrate

ForEachComponent

can contain these elements

statements	lemsschema:statement_

Print

13.3.8 Defining Components

Schema against which LEMS based on these should be valid: LEMS_v0.7.6.xsd. Generated on 18/06/24 from this commit. Please file any issues or questions at the issue tracker here.

Component

Properties

id	String	
name	String	Name by which the component was declared - this shouldn't be accessible.
declaredType	String	Name by which the component was declared - this shouldn't be accessible.
type	String	
eXtends	String	

can contain these elements

insertions	*Insertion*
components	*Component*
abouts	*About*
metas	*Meta*

Schema

```
<xs:complexType name="Component">
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="unbounded" processContents="lax"/>
  </xs:sequence>
  <xs:anyAttribute processContents="skip"/>
</xs:complexType>
```

Usage: XML

```
<Component id="ctb" type="iaf1" threshold="-30 mV" refractoryPeriod="2 ms"␣
↪capacitance="1uF"/>
```

```
<Component id="celltype_c" type="iaf3" leakConductance="3 pS" refractoryPeriod="3 ms"␣
↪threshold="45 mV" leakReversal="-50 mV" deltaV="5mV" capacitance="1uF"/>
```

```
<Component id="gen1" type="spikeGenerator" period="30ms"/>
```

```
<Component id="gen2" type="spikeGenerator2" period="32ms"/>
```

```
<Component id="iaf3cpt" type="iaf3" leakReversal="-50mV" deltaV="50mV" threshold="-
↪30mV" leakConductance="50pS" refractoryPeriod="4ms" capacitance="1pF"/>
```

13.3.9 Geometry

Schema against which LEMS based on these should be valid: LEMS_v0.7.6.xsd. Generated on 18/06/24 from this commit. Please file any issues or questions at the issue tracker here.

Geometry

Specifies the geometrical interpretation of the properties of components realizing this ComponentType.

can contain these elements

frustums	*Frustum*
solids	*Solid*
skeletons	*Skeleton*

Frustum

Solid

Location

Skeleton

can contain these elements

scalarFields	*ScalarField*

ScalarField

CHAPTER

FOURTEEN

NEUROML 2 AND LEMS

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).

14.1 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:

Fig. 14.1: 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**.

```
<xs:complexType name="Izhikevich2007Cell">
    <xs:complexContent>
        <xs:extension base="BaseCellMembPotCap">
            <xs:attribute name="v0" type="Nml2Quantity_voltage" use="required"/>
            <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"/>
```

(continues on next page)

(continued from previous page)

```
        <TimeDerivative variable="v" value="iMemb / C"/>
        <TimeDerivative variable="u" value="a * (b * (v-vr) - u)"/>

        <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 **Component**s 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 **Component**s are defined in the NeuroML document, we can use **Instance**s 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.

14.2 More information

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?*

14.3 Conventions

This page documents various conventions in use in NeuroML.

14.3.1 Prefer underscores instead of spaces

In general, please prefer underscores _ instead of spaces wherever possible, in filenames and ids.

14.3.2 Component IDs: NmlId

Some Components take an `id` parameter of type `NmlId` to set an ID for them. They can then be referred to using their IDs when constructing paths and so on.

IDs of type `NmlId` 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.

14.3.3 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`.

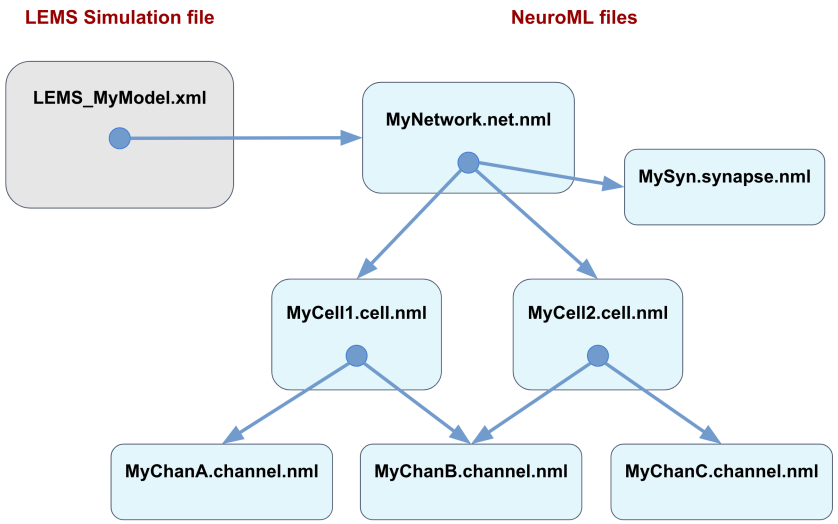


Fig. 14.2: Typical organisation for a NeuroML simulation. The main NeuroML model is specified in a file with the network (`*.net.nml`), which can include/point to files containing individual synapses (`*.synapse.nml`) or cell files (`*.cell.nml`). If the latter are conductance based, they may include 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.

14.3.4 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
- `dendrite_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.

14.4 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, **V = I * 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.

14.5 Paths

Since NeuroMLv2 and LEMS are both XML based, entities in models and simulations must be referred to using *paths* (XPath like). 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*).

operator	description	function	example
`/`	forward slash	used to split the levels in a path string	see *below*
`.`	single period	refers to the level of the current node (usually omitted)	see *below*
`..`	two periods	refers to the level of the current node's parent node	see *below*
`[x]`	square brackets	used to refer to a particular instance (in this case, `x`) in Components/Elements that have a `size` attribute (like *population*)	see *below*
`:`	colon	used to refer to a particular Component instance for `attachments`	ex

Paths start from any element and ascend/descend to refer to the entity that is to be referenced.

14.5.1 Example

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>
                <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".

14.5.2 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 pyneuroml.pynml
>>> help(pynml.list_recording_paths_for_exposures)

Help on function list_recording_paths_for_exposures in module pyneuroml.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 parsers 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+.

14.6 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.

14.6.1 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).

14.6.2 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).

14.7 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.

14.7.1 Specification of format

See *here* for definition of the main elements used in the file, including *Display*, *OutputFile*, etc.

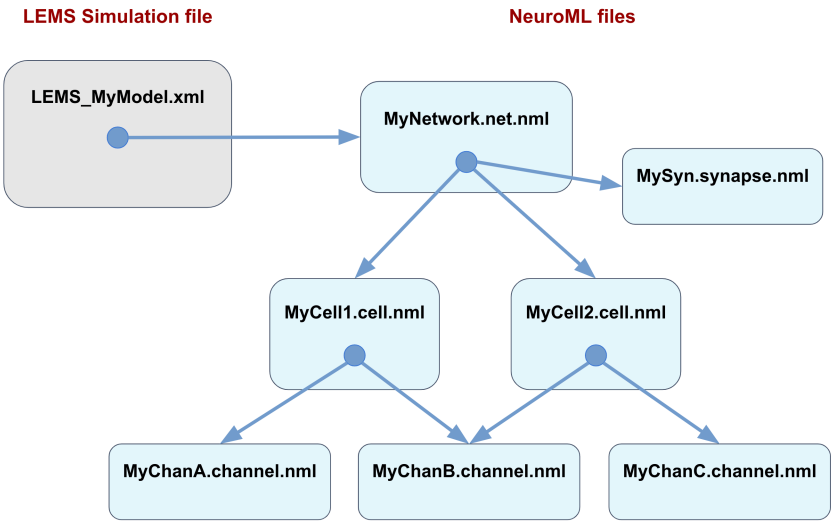


Fig. 14.3: Typical organisation for a NeuroML simulation. The main NeuroML model is specified in a file with the network (`*.net.nml`), which can include/point to files containing individual synapses (`*.synapse.nml`) or cell files (`*.cell.nml`). If the latter are conductance based, they may include 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.

14.7.2 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.

14.7.3 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.

14.7.4 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
```

CHAPTER

FIFTEEN

EXTENDING NEUROML

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.

Having definitions in LEMS allows their re-use, and all new ComponentTypes can be submitted for inclusion to the NeuroMLv2 specification to be made accessible to other users.

- Like NeuroML, LEMS also has a well defined schema (XSD) that is used to validate LEMS XML files.
- 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.

The NeuroML2 standard is a list of *curated LEMS 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 to extend the NeuroMLv2 standard.

15.1 Creating new ComponentTypes with existing NeuroML ComponentTypes

Existing ComponentTypes defined in the NeuroMLv2 standard, when sufficient, should be used to create new ComponentTypes. These new ComponentTypes, since they consist of NeuroMLv2 ComponentTypes, *will be valid against the NeuroMLv2 schema* (must use a `<neuroml ..>` root element). For convenience, the NeuroMLv2 schema includes a subset of the *LEMS elements*.

An example of this type of extension of NeuroML can be see here where a new Calcium dependent ion channel Component requires a new ComponentType `Ca_LVAst_m_tau_tau` that implements the time course of the gate.

However, please note that while the ComponentType will be valid NeuroML, the new Components (instances) one creates of this ComponentType (and models where Components are referenced/used) *will not*—since the NeuroML schema *does not know of the new ComponentType*. The new Components (and the models) will be valid LEMS. For this reason, while the ComponentType file will use a `<neuroml ..>` root tag, the file containing its instantiated Components will use the `<Lems ..>` root tag.

15.2 Creating new ComponentTypes with LEMS elements

When ComponentTypes from the NeuroMLv2 standard are not sufficient for the creation of new ComponentTypes, one must use LEMS elements to do so. The definitions of the *NeuroMLv2 standard core ComponentTypes* are examples of this.

15.2.1 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.

15.2.2 Example: Lorenz model for cellular convection

To see how to create new ComponentTypes using LEMS, let us create 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:

$$\frac{dx}{dt} = \sigma(y - x) \tag{15.1}$$

$$\frac{dy}{dt} = x(\rho - z) - y \tag{15.2}$$

$$\frac{dz}{dt} = xy - \beta z \tag{15.3}$$

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 `x0`, `y0`, and `z0` 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 atomspheric 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␣
↪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 NeuroMLv2 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 atomspheric 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"/>
    <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 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.

15.2.3 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 atomspheric 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"))
```

(continues on next page)

(continued from previous page)

```
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)


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 atomspheric 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"/>
```

(continues on next page)

(continued from previous page)

```
    <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>
  </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.

15.3 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

CHAPTER

SIXTEEN

SOFTWARE AND TOOLS

16.1 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. 16.1: Relationship between *jLEMS*, *jNeuroML*, the *NeuroML 2 LEMS definitions*, *libNeuroML*, *pyLEMS* and *pyNeuroML*.

16.1.1 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.

16.1.2 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.

16.1.3 NeuroML support in other languages

There are preliminary APIs for using NeuroML in *C++* and *MATLAB*.

16.2 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.

16.3 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. 16.2: Relationship between *jLEMS*, *jNeuroML*, the *NeuroML 2 LEMS definitions*, *libNeuroML*, *pyLEMS* and *pyNeuroML*.

16.3.1 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:

- simulation backends
  - `pip install pyneuroml[neuron]`: NEURON
  - `pip install pyneuroml[brian]`: Brian2
  - `pip install pyneuroml[netpyne]`: NetPyNE
  - `pip install pyneuroml[tellurium]`: Tellurium
- COMBINE formats (SEDML/SBML):
  - `pip install pyneuroml[combine]`
- analysis and visualization
  - `pip install pyneuroml[analysis]`
- visualization
  - VisPy:
    - `pip install pyneuroml[vispy]`: Qt6 (default):
    - `pip install pyneuroml[vispy-qt5]` : Qt5
    - `pip install pyneuroml[vispy-common]`: to manually use another supported backend:
  - `pip install pyneuroml[povray]`: Povray
  - `pip install pyneuroml[plotly]`: PlotLy
- model fitting
  - `pip install pyneuroml[tune]`
- HDF5 support
  - `pip install pyneuroml[hdf5]`
- Neuroscience Gateway (NSG) support
  - `pip install pyneuroml[nsg]`
- RDF annotations
  - `pip install pyneuroml[annotations]`

A number of "meta" packages are also available:

- `pip install pyneuroml[all]`: install everything
- `pip install pyneuroml[dev]`: install for development
- `pip install pyneuroml[doc]`: install for building documentation

Installing optional dependencies

The optional "extras" provided by pyNeuroML may require some additional software to be installed that is not Python based, and so cannot be automatically installed using `pip`:

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.

Povray

Requires the installation of the Povray tool.

Vispy

Requires installation of a back end. The default is Qt6, but one can also use Qt5, or a different back end. Vispy also makes use of GPUs via OpenGL. So a recent GPU is recommended for larger scale models.

For more information on individual simulation backends and extras, 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.

16.3.2 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)
```

(continues on next page)

(continued from previous page)

```
DATA
    JNEUROML_VERSION = '0.10.2'

VERSION
    0.5.9

FILE
    /usr/lib/python3.9/site-packages/pyneuroml/__init__.py
```

```
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, custom_amps_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)

FILE
    /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.

16.3.3 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*.

16.3.4 Development

pyNeuroML is developed on GitHub at https://github.com/NeuroML/pyNeuroML under the LPGL-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.

16.4 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.

16.4.1 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
```

16.4.2 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.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
```

16.4.3 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*.

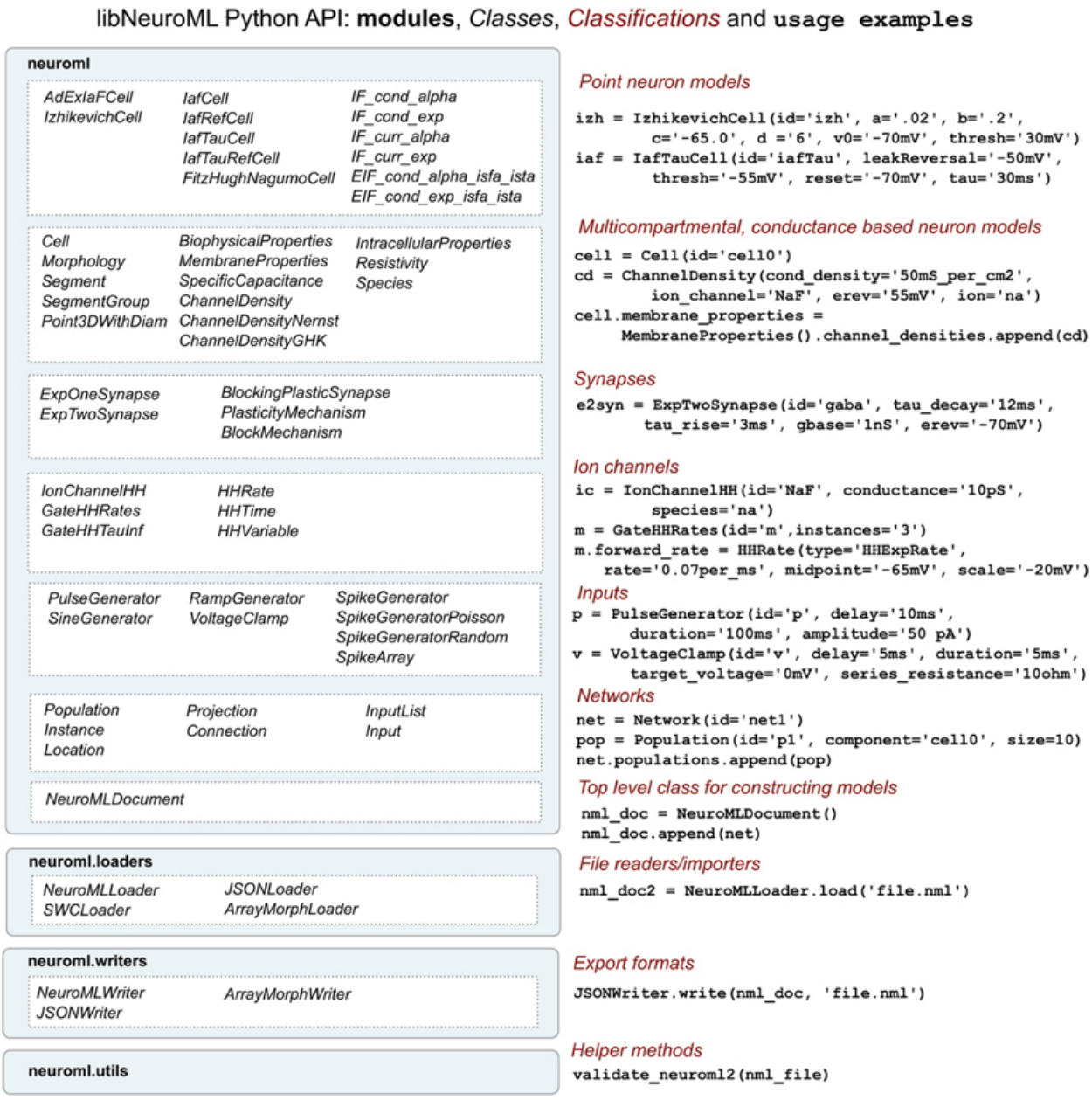


Fig. 16.3: Examples of mapping between Component names in the NeuroML schema and their corresponding libNeuroML Python classes.

16.4.4 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.

16.5 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.

16.5.1 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
```

16.5.2 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)
    @contact: gautham@lisphacker.org
```

(continues on next page)

(continued from previous page)

```
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.

16.5.3 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*.

16.5.4 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.

16.6 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.

16.6.1 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
```

16.6.2 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
```

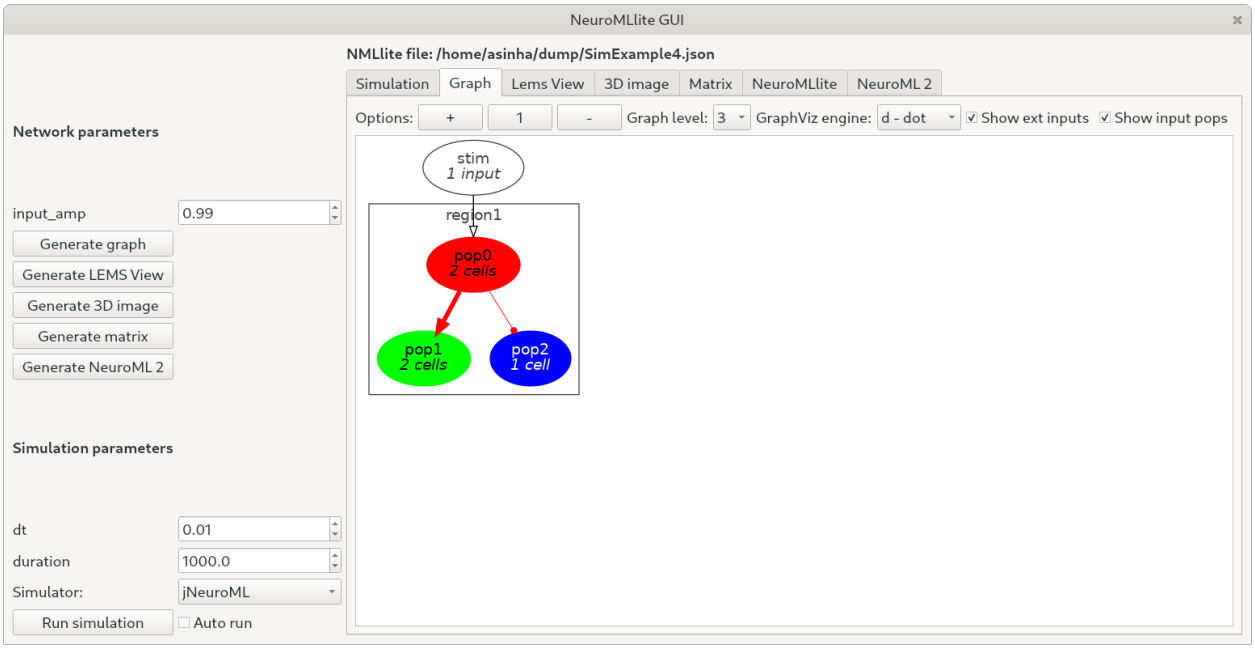


Fig. 16.4: Screenshot of NeuroMLlite UI showing an example simulation

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
    BaseTypes
    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 -pynnbrian       # Generate & run the network in Brian using PyNN
...
```

16.6.3 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*.

16.6.4 Development

pyNeuroML is developed on GitHub at https://github.com/NeuroML/NeuroMLlite under the LPGL-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.

16.7 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.

16.7.1 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
```

16.7.2 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)

16.7.3 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*.

16.7.4 Development

jNeuroML is developed on GitHub at https://github.com/NeuroML/jNeuroML under the LPGL-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
```

16.8 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 at the key reference for how to implement/interpret the language.

16.8.1 Quick start

Since jLEMS is included in *jNeuroML*, it does need not to be installed it 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.

16.8.2 Documentation

Detailed documentation on LEMS is maintained here. For more information on LEMS, please also see Cannon et al. ([CGC+14])

16.8.3 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*.

16.8.4 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.

16.9 NeuroML C++ API

A C++ API for NeuroML.

16.9.1 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.

16.9.2 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.

16.9.3 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*.

16.9.4 Development

The C++ NeuroML API is developed on GitHub at https://github.com/NeuroML/NeuroML_API under the MIT license.

16.10 MatLab NeuroML Toolbox

The NeuroML 2 Toolbox for MATLAB facilitates access to the Java NeuroML 2 API functionality (*jNeuroML*) directly within Matlab.

16.10.1 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())
```

16.10.2 Documentation

Please refer to the *jNeuroML documentation* for information on the Java NeuroML API. Examples on using the Matlab toolbox are available here.

16.10.3 Getting help

For any questions regarding the NeuroML Matlab toolbix, please open an issue on the GitHub issue tracker here. Any bugs and feature requests can also be filed there.

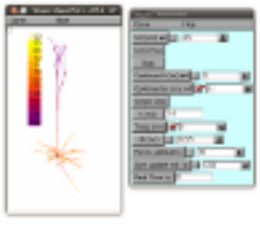
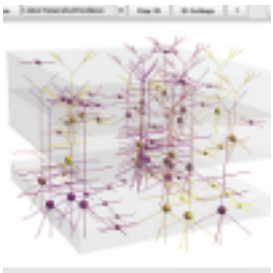
You can also use any of the *communication channels of the NeuroML community*.

16.10.4 Development

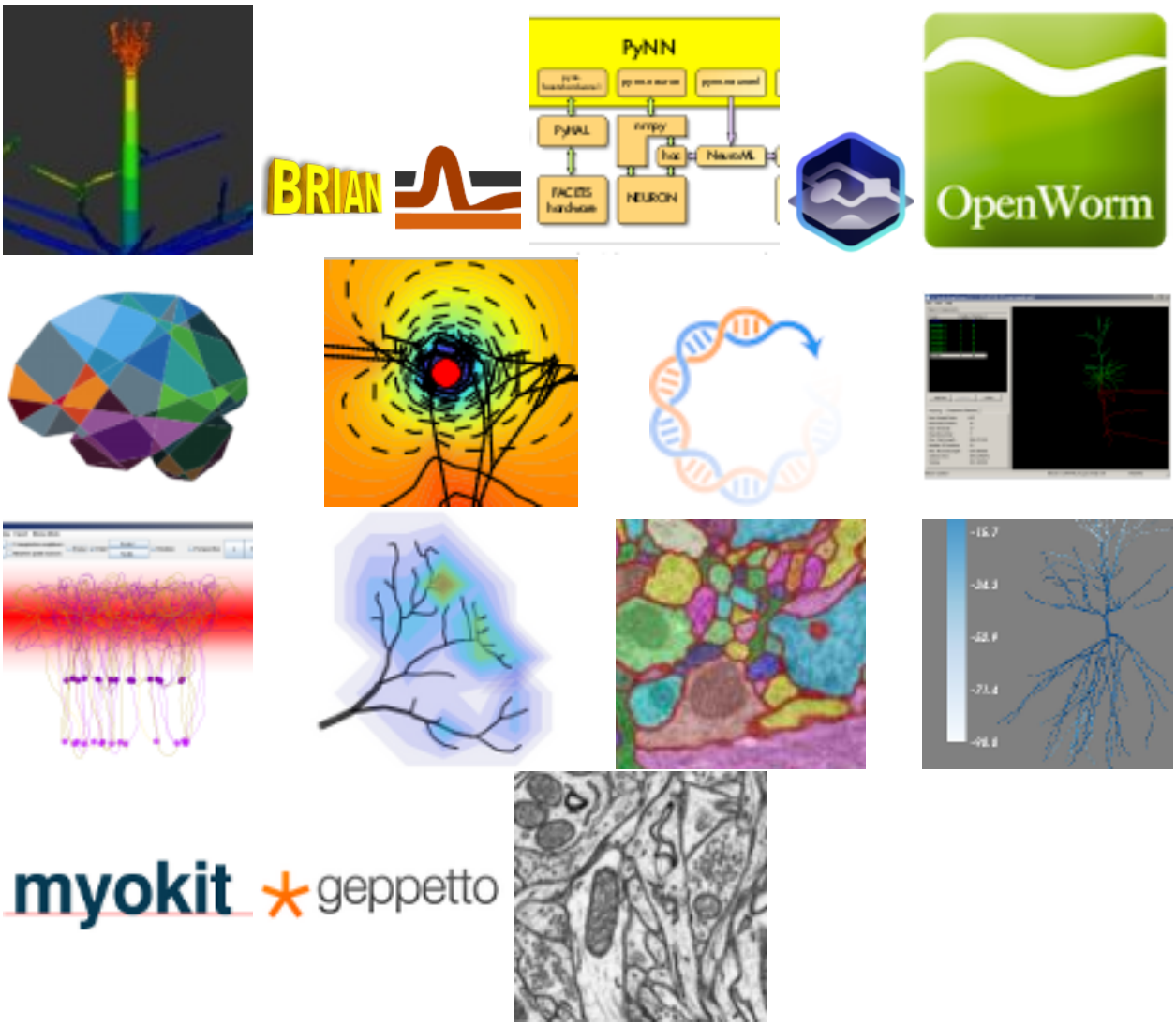
The NeuroML Matlab toolbox is developed on GitHub at https://github.com/NeuroML/NeuroMLToolbox.

16.11 Tools and resources with NeuroML support

Apart from the ***core NeuroML tools*** (e.g. *pyNeuroML*, *jNeuroML*) there are many other applications, libraries and databases which support NeuroML 2 and LEMS.



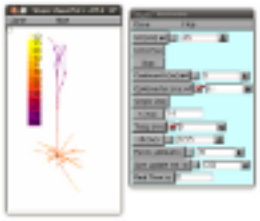
These tools take a *number of different approaches* to adding NeuroML support, from dealing with the format natively to allowing import/export of (subsets of) the language, to an external application generating scripts/code for use in the simulator.

Please help us keep this page up to date.

Tools listed here may have moved to new locations, or may no longer be maintained, and others may be missing. Please file issues if you can help update this information.

16.11.1 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.

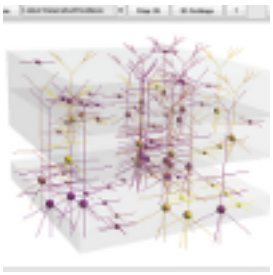
There is a **dedicated page on NEURON/NeuroML interactions** *here*.

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.

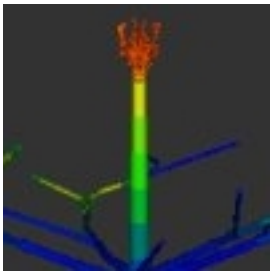
GENESIS



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

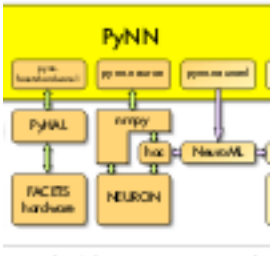


Arbor is a high performance multicompartmental neural simulation library. Addition of support for NeuroML2 and LEMS is under active development. See here.

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.

NEST

nest::

NEST is a simulator for spiking neural network models that focuses on the dynamics, size and structure of neural systems rather than on the exact morphology of individual neurons.

There is a **dedicated page on NEST/NeuroML interactions** *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 *Geppetto*.

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.

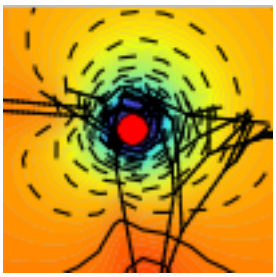
The Virtual Brain



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.

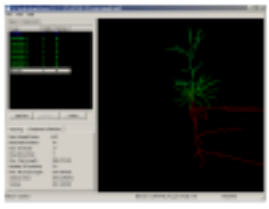
See for example https://biosimulators.org/simulators/pyneuroml/latest.

N2A

"Neurons to Algorithms" (N2A) is a language for modeling neural systems, along with a software tool for editing models and simulating them.

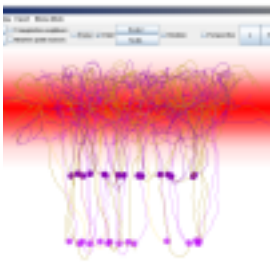
There is a **dedicated page on N2A/NeuroML interactions** *here*.

NeuronLand



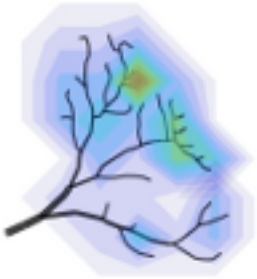
NeuronLand provides NLMorphologyConverter, which is a command line program for converting between over 20 different 3D neuron morphology formats, and NLMorphologyViewer, which provides a simple interface for viewing these data. Both of these tools provide import and export of MorphML.

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.

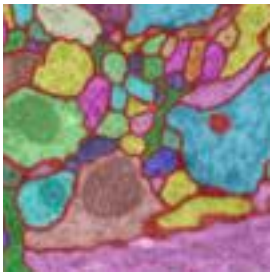
TREES toolbox



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.

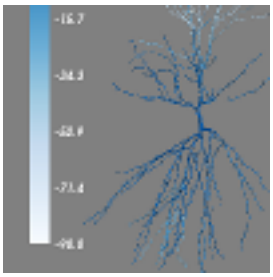
The latest version of the TREES toolbox includes basic functionality for exporting cells in NeuroML v1.x Level 1 (MorphML) or as a NeuroML v2alpha morphology file.

TrakEM2



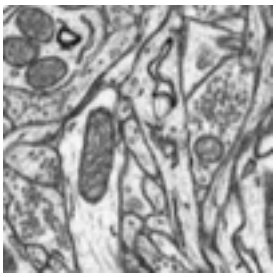
TrakEM2 is an ImageJ plugin for morphological data mining, three-dimensional modelling and image stitching, registration, editing and annotation. As of v0.8n, a menu item “Export - NeuroML…” gives an option to export to MorphML (the anatomy of the arbors only) or NeuroML (the whole network with anatomy 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 easy to select and investigate sections' properties, it offers easy integration with matplotlib for the 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 reload 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

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.

16.11.2 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 multicompartmental 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.

16.11.3 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

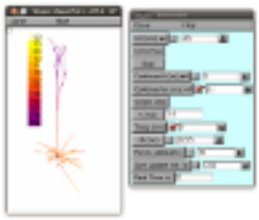
Some simulators allow models to be created with their preferred native model description format, and then exported in valid NeuroML.

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*.

16.11.4 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)
```

Setting the NEURON_HOME environment variable

Since it is possible to install multiple versions of NEURON in different places, the NeuroML tools need to be told where the NEURON tools are. To do this, they look at the `NEURON_HOME` environment variable. This needs to hold the path to where the binary (`bin`) folder holding the NEURON tools such as `nrniv` are located. On Linux like systems, one can use `which` to find these tools and set the variable:

```
$ which nrniv
~/.local/share/virtualenvs/neuroml-311-dev/bin/nrniv

$ export NEURON_HOME="~/.local/share/virtualenvs/neuroml-311-dev/"
```

One can combine these commands together also:

```
$ export NEURON_HOME="$(dirname $(dirname $(which nrniv)))"
```

Using neuroConstruct

NEURON simulations can also be generated from NeuroML model components by *neuroConstruct*, but most of this functionality is related to *NeuroML v1*.

16.11.5 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.

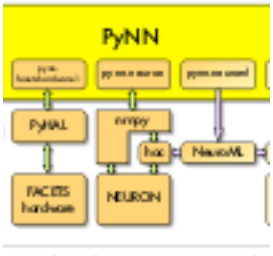
Running NetPyNE on OSBv2

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.

16.11.6 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*.

16.11.7 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.

16.11.8 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
```

16.11.9 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.

16.11.10 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.

16.11.11 N2A and NeuroML

"Neurons to Algorithms" (N2A) is a language for modeling neural systems, along with a software tool for editing models and simulating them

See https://github.com/sandialabs/n2a/wiki/Backend%20LEMS for information on the interactions between NeuroML/LEMS and N2A.

16.11.12 NEST and NeuroML

nest::

NEST is a simulator for spiking neural network models that focuses on the dynamics, size and structure of neural systems rather than on the exact morphology of individual neurons. The development of NEST is coordinated by the NEST Initiative.

NEST is ideal for networks of spiking neurons of any size, for example:

- Models of information processing e.g. in the visual or auditory cortex of mammals,
- Models of network activity dynamics, e.g. laminar cortical networks or balanced random networks,
- Models of learning and plasticity.

See https://github.com/OpenSourceBrain/NESTShowcase for examples of usage of NeuroML and NEST.

16.11.13 SWC and NeuroML

The SWC format was developed to cover most of the information common between Neurolucida, NEURON, and GENESIS formats. It is used by resources such as NeuroMorpho.org.

Information on the SWC format can be found in the NeuroMorpho FAQ under the "What is SWC format" entry.

Recommended applications for converting SWC into NeuroML are CVApp and neuroConstruct (see below).

Tools

A number of tools support conversion of SWC to NeuroML.

CVApp

CVApp is a standalone Java tool that can visualize SWC files (for example from NeuroMorpho.org) and export them into NeuroML2.

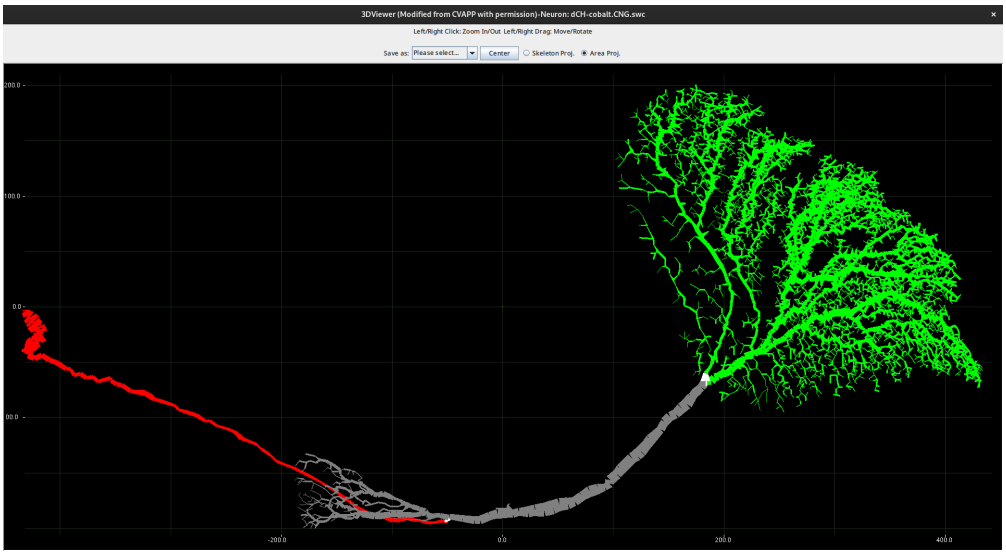


Fig. 16.5: Screenshot of CVApp

One can select "NeuroMLv2" from the "Save As" drop down box to export the loaded reconstruction to NeuroML.

neuroConstruct

neuroConstruct includes functionality to interactively convert CVapp (SWC) files to NeuroML2. Please see the neuroConstruct documentation for more information.

CHAPTER

SEVENTEEN

CITING NEUROML AND RELATED PUBLICATIONS

This page documents how one can cite NeuroML in their work, and lists publications associated with the NeuroML initiative.

17.1 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.

17.1.1 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) libNeuroML 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{\'{a}}s 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},
}
```

17.1.2 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.

17.2 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.

17.2.1 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.

17.2.2 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.

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.

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

EIGHTEEN

FREQUENTLY ASKED QUESTIONS (FAQ)

Please help improve the FAQ.

This page lists some commonly asked questions related to NeuroML. Please open issues to add more entries to this FAQ.

18.1 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.

18.2 2. What is the difference between reader/writer methods in pyNeuroML and libNeuroML?

Both *libNeuroML* and *pyNeuroML* include methods that can read and write NeuroML files. However, they are not the same.

libNeuroML is the low level Python API for working with NeuroML. The loaders/writers included here can therefore read/write NeuroML files. However, these are "low level" functions and do not include additional features.

The readers/writers in pyNeuroML use these low level functions from libNeuroML but also run other checks and include other features.

So,

- pyneuroml.io.read_neuroml2_file should be preferred over neuroml.loaders.read_neuroml2_file: it also allows pre-loading validation checks, and it also handles morphologies referenced in other files.

- pyneuroml.io.write_neuroml2_file should be preferred over neuroml.writers.NeuroMLWriter.write: it also validates the file after writing it.

CHAPTER

NINETEEN

WALK THROUGHS

This chapter documents a number of real-world tasks for users to refer to.

19.1 Converting Ray et al 2020 to NeuroML

This section documents the conversion of Ray et al 2020 [RAS20], which was originally implemented in NEURON, to NeuroML. It broadly follows the steps outlined in the *converting models* section.

For any queries, please contact Ankur Sinha on any of the NeuroML channels.

19.1.1 Setting up

Step 1) Find the original model code

The original code is published on ModelDB.

Step 2) Create GitHub and Open Source Brain accounts for sharing the code

2a) Sign up to GitHub and Open Source Brain

We signed in to GitHub and OSBv1

2b) Create GitHub repository

ModelDB provides GitHub repositories for all its models now. This model is available on GitHub here: https://github.com/ModelDBRepository/262670. The Open Source Brain (OSB) organization on GitHub also keeps a "fork" of these repositories to allow users to easily add them to both Open Source Brain v1 and v2. This fork is here, and is the one that we will work with: https://github.com/OpenSourceBrain/262670.

For the conversion, I (Ankur) created a fork of this repository with a new branch to work in: https://github.com/sanjayankur31/262670. A pull request work flow was used to submit converted bits back to the repository.

The first step was to re-organise the code to prepare it for conversion. All the existing code was moved to a new NEURON folder, and a new NeuroML2 folder set up to store the NeuroML version.

2c) Create Open Source Brain project

A new project was created on OSBv1 and linked to the OSB repository: https://v1.opensourcebrain.org/projects/locust-mushroom-body.

19.1.2 Converting to NeuroML

On inspection of the model, we see that it has two biophysically detailed cell models:

- the GGN (Giant GABAergic Neuron)
- the KC (Kenyon cell)

Converting the Giant GABAergic Neuron

Step 1) Exporting morphology of the GGN

Let's start with the GGN first. It's morphology is defined as an SWC file in this file. One can download this file and view the morphology in a tool, like the HBP morphology viewer.

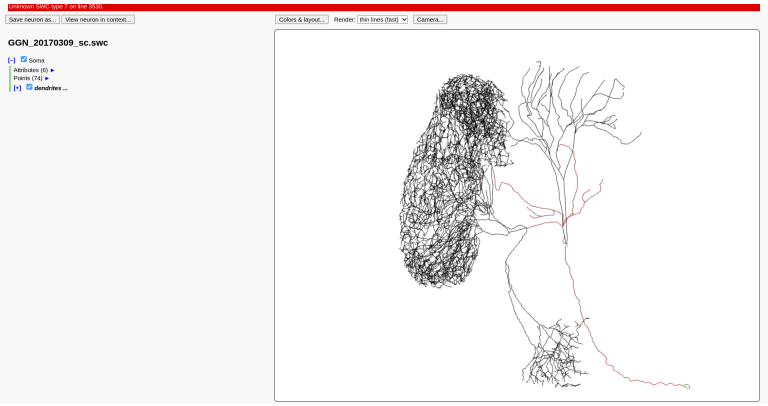


Fig. 19.1: Visualisation of the GGN in the HBP morphology viewer.

A NEURON HOC script that includes the full morphology and the biophysics is also included.

Let us export the morphology first. pyNeuroML includes the `export_to_neuroml2` helper function that exports a cell model in NEURON to NeuroML. We can write a short script to use this function to export the morphology from the provided HOC script.

```
#!/usr/bin/env python3
"""
Convert cell morphology to NeuroML.

We only export morphologies here. We add the biophysics manually.

File: NeuroML2/scripts/cell2nml.py
"""

import os
import sys
```

(continues on next page)

(continued from previous page)

```
import pyneuroml
from pyneuroml.neuron import export_to_neuroml2
from neuron import h


def main(acell):
    """Main runner method.

    :param acell: name of cell
    :returns: None

    """
    loader_hoc_file = f"{acell}_loader.hoc"
    loader_hoc_file_txt = """
    /*load_file("nrngui.hoc")*/
    load_file("stdrun.hoc")
    xopen("../../NEURON/mb/cell_templates/GGN_20170309_sc.hoc")
    objref cell
    cell = new GGN_20170309_sc()
    """

    with open(loader_hoc_file, 'w') as f:
        print(loader_hoc_file_txt, file=f)

    export_to_neuroml2(loader_hoc_file, f"{acell}.morph.cell.nml",
                       includeBiophysicalProperties=False, validate=False)

    os.remove(loader_hoc_file)
    # Note--a couple of diameters are 0.0, modified to 0.001 to validate the
    # model


if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("This script only accepts one argument.")
        sys.exit(1)
    main(sys.argv[1])
```

What we're doing here is using the HOC script to build the cell model in NEURON, and then exporting it to NeuroML. Calling it as `python cellmorph2nml.py GGN` will create a new file: `GGN.morph.cell.nml` which contains the morphology of the cell in NeuroML format. Note that while `export_to_neuroml2` does allow exporting the biophysics of the cell, it is better to add these manually later once one has gone through and converted the required ion channels and so on.

We can visualise the morphology using the pyNeuroML tools:

```
pynml-plotmorph -i GGN.morph.cell.nml
```

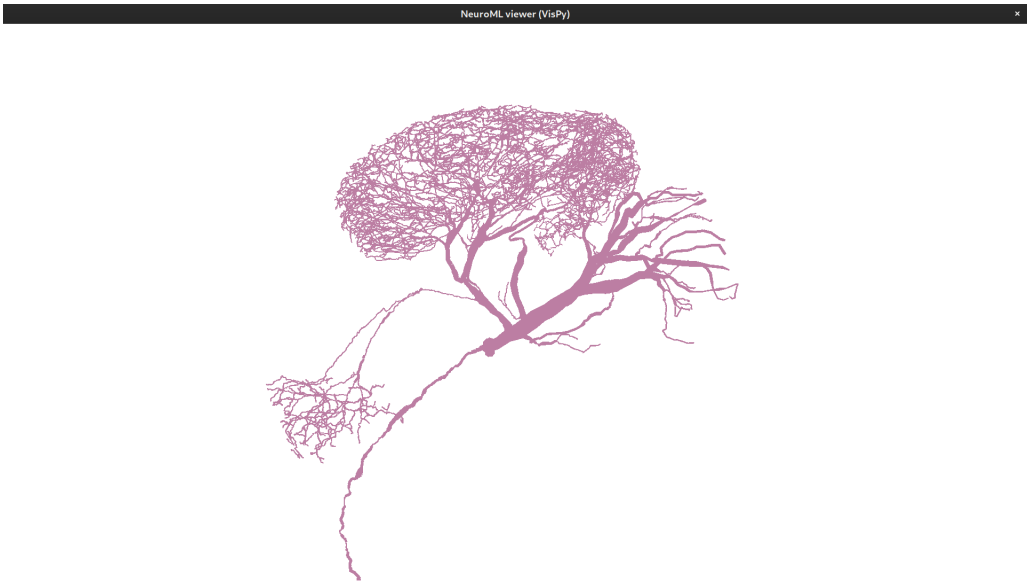


Fig. 19.2: Visualisation of the GGN using `pynml-plotmorph`

Step 2) Adding biophysics to the GGN

Now that we have the morphology of the GGN exported, we can add the biophysics. We need to inspect the original model code to learn about the biophysics. In this model, for the GGN cell, the biophysics are included in the HOC script:

```
proc biophys() {
  forsec all {
    Ra = 100.0
    cm = 1
    insert pas
      g_pas = 0.03e-3   // S/cm2 - as per Laurent et al 1990 RM = 33kohm-cm2
      e_pas = -51
  }
}
```

As we see here, this is a passive cell without any ion channels. To add the biophysics, we write a simple Python script that will make use of the pyNeuroML API. The complete script is present in the repository:

```
def load_and_setup_cell(cellname: str):
    """Load a cell, and clean it to prepare it for further modifications.

    These operations are common for all cells.

    :param cellname: name of cell.
        the file containing the cell should then be <cell>.morph.cell.nml
    :returns: document with cell
    :rtype: neuroml.NeuroMLDocument

    """
    celldoc = read_neuroml2_file(
        f"{cellname}.morph.cell.nml"
    )  # type: neuroml.NeuroMLDocument
    cell = celldoc.cells[0]  # type: neuroml.Cell
    celldoc.networks = []
```

(continues on next page)

(continued from previous page)

```
    cell.id = cellname
    cell.notes = cell.notes.replace("GGN_20170309_sc_0_0", cellname)
    cell.notes += ". Reference: Subhasis Ray, Zane N Aldworth, Mark A Stopfer (2020)␣
↪Feedback inhibition and its control in an insect olfactory circuit eLife 9:e53281."

    [
        default_all_group,
        default_soma_group,
        default_dendrite_group,
        default_axon_group,
    ] = cell.setup_default_segment_groups(
        use_convention=True,
        default_groups=["all", "soma_group", "dendrite_group", "axon_group"],
    )

    # populate default groups
    for sg in cell.morphology.segment_groups:
        if "soma" in sg.id and sg.id != "soma_group":
            default_soma_group.add(neuroml.Include(segment_groups=sg.id))
        if "axon" in sg.id and sg.id != "axon_group":
            default_axon_group.add(neuroml.Include(segment_groups=sg.id))
        if "dend" in sg.id and sg.id != "dendrite_group":
            default_dendrite_group.add(neuroml.Include(segment_groups=sg.id))

    cell.optimise_segment_groups()

    return celldoc


def postprocess_GGN():
    """Post process GGN and add biophysics."""
    cellname = "GGN"
    celldoc = load_and_setup_cell(cellname)
    cell = celldoc.cells[0]  # type: neuroml.Cell

    # biophysics
    # all
    cell.add_channel_density(
        nml_cell_doc=celldoc,
        cd_id="pas",
        ion_channel="pas",
        cond_density="0.00003 S_per_cm2",
        erev="-51 mV",
        group_id="all",
        ion="non_specific",
        ion_chan_def_file="channels/pas.channel.nml",
    )
    cell.set_resistivity("0.1 kohm_cm", group_id="all")
    cell.set_specific_capacitance("1 uF_per_cm2", group_id="all")
    cell.set_init_memb_potential("-80mV")

    # L1 validation
    # cell.validate(recursive=True)
    cell.summary(morph=False, biophys=True)
    # use pynml writer to also run L2 validation
    write_neuroml2_file(celldoc, f"{cellname}.cell.nml")
```

The `load_and_setup_cell` function does some basic clean up and set up of the cell. It ensures that the various segments that were exported from NEURON are placed into the conventional segment groups. The `postprocess_GGN` function then adds the passive biophysics to the cell.

The `pas` channel is a standard implementation of a passive ion channel. The rest are membrane properties–resistivity, specific capacitance and so on. Once this is set up, we write the cell to a new file.

The GGN cell has now been converted. Since the GGN is a simple passive cell, we won't test its biophysics just yet.

Converting the Kenyon Cell

The KC cell is defined in the `kc_1_comp.hoc` file. Whereas the GGN cell had a complex morphology but passive biophysics, the KC cell has very simple morphology—a single compartment—but does contain active channels:

```
create soma


objref all
proc subsets() { local i
  objref all
  all = new SectionList()
  soma all.append()
}
proc geom() {
     soma {  // Total Cm = 4 pF
        L = 6.366
        diam = 20
    }
}

proc biophys() {
  forsec all {
    Ra = 35.4
    cm = 1
    insert pas
      g_pas = 9.75e-5          // S/cm2
      e_pas = -70              // mV
    insert kv
      gbar_kv = 1.5e-3        // S/cm2
    insert ka
      gbar_ka = 1.4525e-2     // S/cm2
    insert kst
      gbar_kst = 2.0275e-3    // S/cm2
    insert naf
      gbar_naf = 3.5e-2       // S/cm2
    insert nas
      gbar_nas = 3e-3         // S/cm2
      ek = -81.0              // mV
      ena = 58.0                // mV
  }
}
```

Step 1) Converting ion channels

For this cell, we will first convert the various ion channel models. These are included in the `mod` folder. An inspection tells us that these are all Hodgkin-Huxley type ion channels that use similar formalisms.

The first thing to do is to generate plots of time courses and steady states of the various ion channels. These can be done easily using `pynml-modchananalysis` command line tool included in pyNeuroML. We begin with the `nas` channel:

```
pynml-modchananalysis -modFile nas_wustenberg.mod nas
```

This will generate two plots, one for the steady state dynamics (`inf`), and one for the time course (`tau`) for activation variables in the channels:

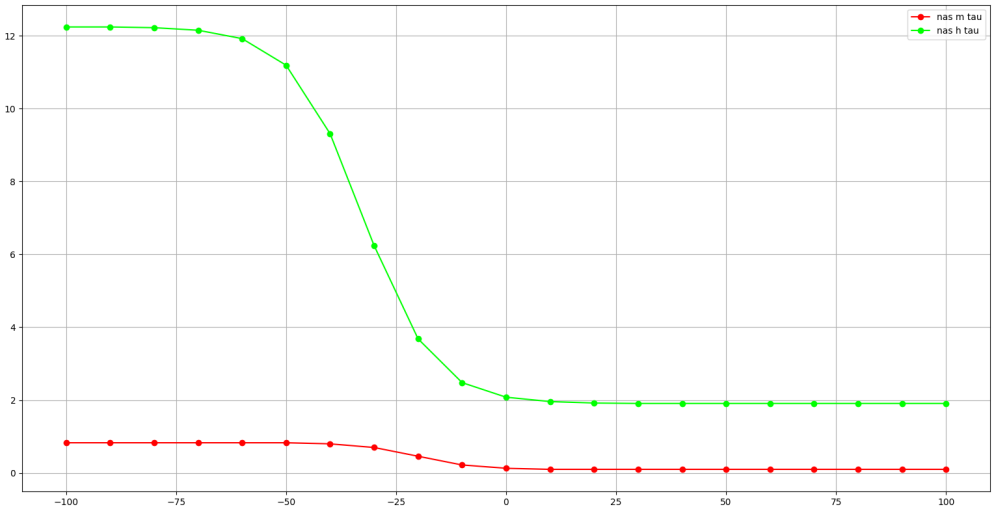


Fig. 19.3: Time course of activation variables of nas channel, generated with pynml-modchananalysis

The mod file defining the nas channel is shown below:

```
: nas_wustenberg.mod ---
:
: Filename: nas_wustenberg.mod
: Description:
: Author: Subhasis Ray
: Maintainer:
: Created: Wed Dec 13 19:06:03 EST 2017
: Version:
: Last-Updated: Mon Jun 18 14:38:15 2018 (-0400)
:           By: Subhasis Ray
: URL:
: Doc URL:
: Keywords:
: Compatibility:

: Commentary:
:
```

(continues on next page)

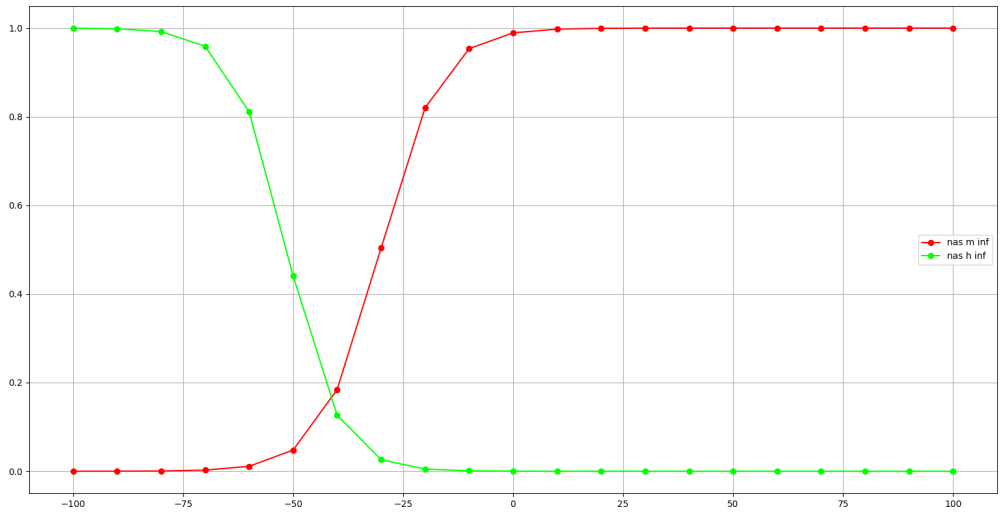


Fig. 19.4: Steady state dynamics of activation variables of nas channel, generated with pynml-modchananalysis

(continued from previous page)

```
: NEURON implementation of slow Na+ channel ( NAS ) from Wustenberg
: DG, Boytcheva M, Grunewald B, Byrne JH, Menzel R, Baxter DA

: This is slow Na+ channel in Apis mellifera Kenyon cells :(cultured).

TITLE Slow NA+ current in honey bee KC from Wustenberg et al 2004

COMMENT
  NEURON implementation by Subhasis Ray (ray dot subhasis at gmail dot com).

ENDCOMMENT

INDEPENDENT { t FROM 0 TO 1 WITH 1 (ms) }

NEURON {
        SUFFIX nas
        USEION na READ ena WRITE ina
        RANGE gbar, ina, g
}

UNITS {
        (S) = (siemens)
        (mV) = (millivolt)
        (mA) = (milliamp)
}

PARAMETER {
        gbar = 0.0      (mho/cm2)
}

ASSIGNED {
```

(continues on next page)

(continued from previous page)

```
        ena         (mV)
        v         (mV)
        ina         (mA/cm2)
        g         (S/cm2)
        minf
        hinf
        mtau         (ms)
        htau         (ms)
}

STATE {
    m
    h
}

BREAKPOINT {
        SOLVE states METHOD cnexp
        g = gbar * m * m * m * h
        ina = g * ( v - ena )
}

INITIAL {
        settables(v)
        m = minf
        h  = hinf
}

DERIVATIVE states {
        settables(v)
        h' = (hinf - h) / htau
        m' = (minf - m ) / mtau
}

: Parameters from the article (Table 2):
:
:       E,mV    g,nS            taumax,ms       taumin,ms       Vh1,mV  s1      Vh2,
 ↪mV  s2      N
: INa                   minf                                    -30.1   6.65        ␣
 ↪          3
:                       taum  0.83            0.093           -20.3   6.45
: INaF  58      140     hinf                                    -51.4   5.9         ␣
 ↪          1
:                       tauh  1.66            0.12            -8.03   8.69
:
: INaS  58      12      hinf                                    -51.4   5.9         ␣
 ↪          1
:                       tauh  12.24           1.9             -32.6   8
: The equations are:
:       minf = 1 / ( 1 + exp((Vh - V) / s))
:       hinf = 1 / ( 1 + exp((V - Vh) / s))
:       taum = (taumax - taumin) / (1 + exp((V - Vh1) / s1)) + taumin

PROCEDURE settables(v (mV)) {
UNITSOFF
        TABLE minf, hinf, mtau, htau FROM -120 TO 40 WITH 641
        minf  = 1.0 / (1 + exp((-30.1 - v) / 6.65))
        hinf  = 1.0 / (1 + exp((v + 51.4) / 5.9 ))
```

(continues on next page)

(continued from previous page)

```
        mtau = (0.83 - 0.093) / (1 + exp((v + 20.3) / 6.45)) + 0.093
        htau = (12.24 - 1.9) / (1 + exp((v + 32.6) / 8.0)) + 1.9
UNITSON
}

: nas_wustenberg.mod ends here
```

Here we have the `m` and `h` activation variables. Although they are described in the standard Hodgkin Huxley formalism, in the `settables` procedure, we can see that their values are only calculated in the range of -120mV to 40mV. The value does not change beyond 40mV. This could be done for a number of reasons. Perhaps the cell's membrane potential does not go beyond 40mV.

We will attempt to remain faithful to the mod file in our conversion, so we will also incorporate this feature.

Since we know this is a Hodgkin Huxley type channel, we can search the schema to see if there are any elements that can describe it. A search shows us that the ionChannelHH component type exists in the schema/standard. In the schema, this is identical to ionChannel. The usage examples included in the documentation indicate that we can use these elements from the standard to describe the ion channel here. We need to:

- include the `m` gate, which has 3 sub units (`m^3`)
- include the `h` gate, which has 1 sub unit (`h^1`)
- formalise the equations that are used to calculate the steady state (`inf`)and time course (`tau`) for these gates/activation variables.

To begin with, let us ignore the restriction included in the mod file at 40mV. Our NeuroML description will look something like this:

```
<ionChannel id="nas" type="ionChannelHH" conductance="1pS" species="na">

<!-- some more details to be added here -->

</ionChannelHH>
```

Now, there are number of different ways of expressing the dynamics of the activation variables. (See this page for an introduction to HH formalism). One can use the values of the forward and reverse rates (called `alpha` and `beta` in general) to calculate the steady state and time course. Another possibility is that `alpha` and `beta` are not given, and instead the equations for the steady state and time course are. The latter is the case here:

```
inf = 1 / ( 1 + exp((Vh - V) / s) )
```

If the rates are given, one can use the gateHHrates component type from the standard. If the steady state and time course are, we can use gateHHtauInf. There are also other components that can be used if a combination of rates and/or steady state and time course are given.

The equation above is in the form of a sigmoid function. Another search in the standard shows us that we have the HHSigmoidVariable component type that can be used to represent a rate that is a sigmoid function. The `dynamics` tab tells us that the equation is represented as:

```
x = rate / (1 + exp(0 - (v - midpoint)/scale))
```

Putting the equation for `minf` and `hinf` together with this form:

```
x = rate / (1 + exp(0 - (v - midpoint)/scale))
minf  = 1.0 / (1 + exp((-30.1 - v) / 6.65))
hinf  = 1.0 / (1 + exp((v + 51.4) / 5.9 ))
```

We can see that for `minf`, `rate = 1 per ms`, `scale = 6.65mV` and `midpoint = -30.1mV`. Similarly, for `hinf`, `rate = 1 per ms`, `scale = -5.9mV` and `midpoint = -51.4mV`.

```
<ionChannel id="nas" type="ionChannelHH" conductance="1pS" species="na">

    <gate id="m" instances="3" type="gateHHtauInf">
        <steadyState type="HHSigmoidVariable" midpoint="-30.1mV" scale="6.65mV" rate=
↪"1" />
        <timeCourse />
    </gate>

    <gate id="h" instances="1" type="gateHHtauInf">
        <steadyState type="HHSigmoidVariable" midpoint="-51.4mV" scale="-5.9mV" rate=
↪"1" />
        <timeCourse />
    </gate>
</ionChannel>
```

The time course is given by:

```
taum = (taumax - taumin) / (1 + exp((V - Vh1) / s1)) + taumin
```

Even though this is also a sigmoid, it is not, unfortunately, a standard form. A component type does not exist in the NeuroML standard that can encapsulate this form (It has more parameters, and has an additional `+ taumin`).

This is not a problem, though, because NeuroML can be easily extended using *LEMS*. We can write a new component type to encapsulate these dynamics based on the LEMS definition of the HHSigmoidVariable component type:

```
<ComponentType name="HHSigmoidVariable"
               extends="baseHHVariable"
               description="Sigmoidal form for variable equation">
    <Dynamics>
        <DerivedVariable name="x" dimension="none" exposure="x" value="rate / (1 +␣
↪exp(0 - (v - midpoint)/scale))"/>
    </Dynamics>
</ComponentType>
```

Our new component will be this, and we will save it in a different file that we can then "include" in the nas channel definition file:

```
<!-- Saved as RayTau.nml -->
<?xml version="1.0" encoding="ISO-8859-1"?>
<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_
↪v2beta4.xsd" id="NeuroML_ionChannel">

    <ComponentType name="Ray_tau"
                   extends="baseVoltageDepTime"
                   description="Tau parameter">

        <Parameter name="max_tau" dimension="time"/>
        <Parameter name="min_tau" dimension="time"/>
        <Parameter name="midpoint" dimension="voltage"/>
        <Parameter name="scale" dimension="voltage"/>
        <Dynamics>
            <DerivedVariable name="t" dimension="time" exposure="t" value="(((max_tau␣
↪- min_tau) / (1 + exp(0 - (v - midpoint) / scale))) + min_tau)"/>
```

(continues on next page)

(continued from previous page)

```
        </Dynamics>
    </ComponentType>
</neuroml>
```

Note that it is very similar to the HHSigmoidVariable definition. The only difference is that we have had to define additional parameters to use in our equation. Also note that `HHSigmoidVariable` extends `baseHHVariable` which extends `baseVoltageDepVariable`. However, since we know that `tau` is a time value, we extend the `baseVoltageDepTime` component type instead. This is very similar to `baseVoltageDepVariable`, but is designed for component types producing time values, such as the time course here.

We save this as a different component type (a class), and we will provide it parameters to create our time courses for both `m` and `h` activation variables. Our completed file will look like this:

```
<!-- saved as nas.channel.nml -->
<?xml version="1.0" encoding="ISO-8859-1"?>
<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_
 ↪v2beta4.xsd" id="NeuroML_ionChannel">

    <notes>NeuroML file containing a single ion channel</notes>
    <include href="RayTau.nml" />

    <ionChannel id="nas" type="ionChannelHH" conductance="1pS" species="na">

        <gate id="m" instances="3" type="gateHHtauInf">
            <steadyState type="HHSigmoidVariable" midpoint="-30.1mV" scale="6.65mV"␣
 ↪rate="1" />
            <timeCourse type="Ray_tau" min_tau="0.83 ms" max_tau="0.093 ms" midpoint=
 ↪"-20.3 mV" scale="6.45mV"/>
        </gate>

        <gate id="h" instances="1" type="gateHHtauInf">
            <steadyState type="HHSigmoidVariable" midpoint="-51.4mV" scale="-5.9mV"␣
 ↪rate="1" />
            <timeCourse type="Ray_tau" min_tau="1.9 ms" max_tau="12.24 ms" midpoint="-
 ↪32.6 mV" scale="-8.0mV"/>
        </gate>
    </ionChannel>

</neuroml>
```

Running `pynml-channelanalysis nas.nml` will generate the graphs for the steady state and time course from our channel definition, and you will see that these are the same as the graphs we generated from the mod files.

We are almost there, but we have a little more work to do here. Remember that the original mod file limited the value of steady state at 40mV? We have not incorporated that into our channel file yet.

Since the `HHSigmoidVariable` we have used for the steady state does not allow multiple equations, we will write a new component type for the steady state also:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<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_
 ↪v2beta4.xsd" id="NeuroML_ionChannel">
```

(continues on next page)

(continued from previous page)

```
    <ComponentType name="Ray_inf"
        extends="baseVoltageDepVariable"
        description="Inf parameter for Ray et al 2020" >

        <Constant name="table_max" dimension="voltage" value="40 mV"/>
        <Parameter name="rate" dimension="none"/>
        <Parameter name="midpoint" dimension="voltage"/>
        <Parameter name="scale" dimension="voltage"/>
        <Dynamics>
            <ConditionalDerivedVariable name="x" dimension="per_time" exposure="x">
                <Case condition="v .gt. table_max" value="(rate / (1 + exp(0 - (table_
↪max - midpoint) / scale)))"/>
                <Case value="(rate / (1 + exp(0 - (v - midpoint) / scale)))"/>
            </ConditionalDerivedVariable>
        </Dynamics>
    </ComponentType>
<neuroml/>
```

The only difference here is that instead of the `DerivedVariable`, we have used a `ConditionalDerivedVariable` that allows conditional dynamics. We have two cases here:

- if `v` is greater than `table_max` (which is 40mV), the rate is the value at `v=40mV`
- otherwise, the rate is calculated from the value of `v`

1a) Kv, Naf, Kst

The nas channel is now done. If we look at the other channels—naf, kv, and kst—they follow similar formalisms. So, we can re-use our newly created component types, `Ray_inf` and `Ray_tau`. In fact, we consolidate them in a single file, `RaySigmoid.nml`, and "include" this in the channel definition files. For example, here is `kv.channel.nml`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<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_
↪v2beta4.xsd" id="NeuroML_ionChannel">

    <notes>NeuroML file containing a single ion channel</notes>
    <include href="RaySigmoid.nml" />

    <ionChannel id="kv" conductance="1pS" type="ionChannelHH" species="k">

        <notes>
                Implementation of A type K+ channel ( KV ) from Wustenberg DG,␣
↪Boytcheva M, Grunewald B, Byrne JH, Menzel R, Baxter DA.
                This is a delayed rectifier type K+ channel in Apis mellifera Kenyon␣
↪cells (cultured).
        </notes>

        <!-- custom component types because the tables in the mod files only go to 40␣
↪-->
        <gate id="m" type="gateHHtauInf" instances="4">
            <steadyState type="Ray_inf" rate="1.0" midpoint="-37.6mV" scale="27.24mV"/
↪>
            <timeCourse type="Ray_tau" min_tau="1.85 ms" max_tau="3.53 ms" midpoint=
↪"45.0 mV" scale="-13.71mV"/>
```

(continues on next page)

(continued from previous page)

```
        </gate>

    </ionChannel>
</neuroml>
```

Plots for the steady state and time course are:

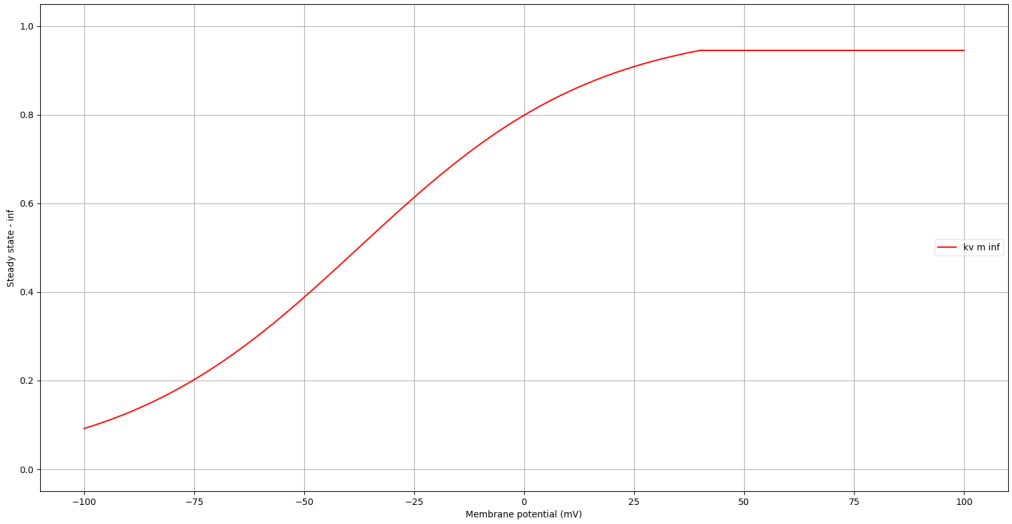


Fig. 19.5: Steady state dynamics of activation variables of kv channel, generated with pynml-channelanalysis

In these graphs, the effect of the conditional at 40mV becomes more apparent.

1b) Ka

The last remaining channel is the ka channel. It's dynamics are defined in the mod file as:

```
PROCEDURE settables(v (mV)) {
UNITSOFF
        TABLE minf, hinf, mtau, htau FROM -120 TO 40 WITH 641
        minf  = 1.0 / (1 + exp((-20.1 - v)/16.1))
        hinf  = 1.0 / ( 1 + exp( ( v + 74.7 ) / 7 ) )
        mtau = (1.65 - 0.35) / ((1 + exp(- (v + 70) / 4.0)) * (1 + exp((v + 20) / 12.
↪0))) + 0.35
        htau = (90 - 2.5) / ((1 + exp(- (v + 60) / 25.0)) * (1 + exp((v + 62) / 16.
↪0))) + 2.5
UNITSON
}
```

The steady state here follows the same formalism, but the time course does not. So, we need to create new component type to encapsulate the time courses here, similar to the `Ray_tau` component type that we did before:

```
    <ComponentType name="Ray_ka_tau"
                   extends="baseVoltageDepTime"
```

(continues on next page)

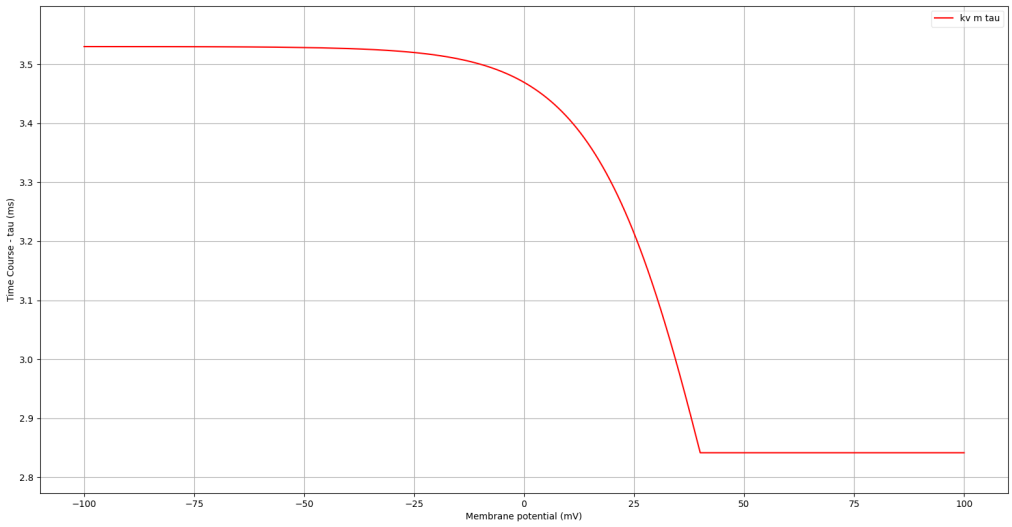


Fig. 19.6: Time course of activation variables of kv channel, generated with pynml-channelanalysis

(continued from previous page)

```
                description="Tau parameter to describe ka">

        <Parameter name="max_tau" dimension="time"/>
        <Parameter name="min_tau" dimension="time"/>
        <Parameter name="midpoint1" dimension="voltage"/>
        <Parameter name="scale1" dimension="voltage"/>
        <Parameter name="midpoint2" dimension="voltage"/>
        <Parameter name="scale2" dimension="voltage"/>
        <Constant name="table_max" dimension="voltage" value="40 mV"/>
        <Dynamics>
            <ConditionalDerivedVariable name="t" dimension="time" exposure="t" >
                <Case condition="v .gt. table_max" value="(max_tau - min_tau) / ((1 +␣
↪exp(-(table_max + midpoint1) / scale1)) * ( 1 + exp((table_max + midpoint2) /␣
↪scale2))) + min_tau"/>
                <Case value="(max_tau - min_tau) / ((1 + exp(-(v + midpoint1) /␣
↪scale1)) * ( 1 + exp((v + midpoint2) / scale2))) + min_tau"/>

            </ConditionalDerivedVariable>
        </Dynamics>
    </ComponentType>
```

We also add this to our `RaySigmoid.nml` file. The ka.channel.nml file will, finally, look like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<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_
↪v2beta4.xsd" id="NeuroML_ionChannel">

    <notes>NeuroML file containing a single ion channel</notes>
    <include href="RaySigmoid.nml" />
```

(continues on next page)

(continued from previous page)

```
    <ionChannel id="ka" conductance="1pS" type="ionChannelHH" species="k">

        <notes>
                Implementation of A type K+ channel ( KA ) from Wustenberg DG,␣
↪Boytcheva M, Grunewald B, Byrne JH, Menzel R, Baxter DA.
                This is transient A type K+ channel in Apis mellifera Kenyon cells␣
↪(cultured).
        </notes>

        <!-- custom component types because the tables in the mod files only go to 40␣
↪-->
        <gate id="m" type="gateHHtauInf" instances="3">
            <timeCourse type="Ray_ka_tau" midpoint1="70mV" midpoint2="2.0mV" scale1=
↪"4.0mV" scale2="12.0mV" min_tau="0.35ms" max_tau="1.65ms"/>
            <steadyState type="Ray_inf" rate="1.0" midpoint="-20.1mV" scale="16.1mV"/>
        </gate>

        <gate id="h" type="gateHHtauInf" instances="1">
            <timeCourse type="Ray_ka_tau" midpoint1="60mV" midpoint2="62.0mV" scale1=
↪"25.0mV" scale2="16.0mV" min_tau="2.5ms" max_tau="90.0ms"/>
            <steadyState type="Ray_inf" rate="1.0" midpoint="-74.7mV" scale="-7.0mV"/>
        </gate>

    </ionChannel>
</neuroml>
```

That is all the ion channels converted.

The ion channels are usually the most involved to convert because one must understand their initial descriptions in the mod files. Even though the NMODL language used in mod files does have a well defined structure, like general programming languages, it is free-flowing. This means that different people can write the same dynamics in different ways. On the other hand, NeuroML and LEMS are more formal with more strict structures, and once channels are converted to these formats, they are much easier to understand.

Step 2) Creating the morphology

Since the morphology of the KC is a single compartment, we don't need to export it from NEURON. We can create it ourselves.

The morphology is given in the HOC script:

```
proc geom() {
     soma {  // Total Cm = 4 pF
        L = 6.366
        diam = 20
    }
}
```

We can create this using a Python script:

```
    celldoc = component_factory(
        "NeuroMLDocument", id="KC_doc"
    )  # type: neuroml.NeuroMLDocument
    cell = celldoc.add("Cell", id="KC", validate=False)  # type: neuroml.Cell
```

(continues on next page)

(continued from previous page)

```
    cell.setup_nml_cell()
    cell.add_segment([0, 0, 0, 20], [0, 0, 6.366, 20], seg_type="soma")
```

The `setup_nml_cell` and `add_segment` methods are part of the Cell class in the standard API.

Now that we have the morphology and the ion channels for the KC, we can add the biophysics to the morphology to complete the cell:

```
    # biophysics
    # all
    cell.set_resistivity("35.4 ohm_cm", group_id="all")
    cell.set_specific_capacitance("1 uF_per_cm2", group_id="all")
    cell.set_init_memb_potential("-70mV")
    cell.set_spike_thresh("-10mV")

    cell.add_channel_density(
        nml_cell_doc=celldoc,
        cd_id="pas",
        ion_channel="pas",
        cond_density="9.75e-5 S_per_cm2",
        erev="-70 mV",
        group_id="all",
        ion="non_specific",
        ion_chan_def_file="channels/pas.channel.nml",
    )

    # K
    cell.add_channel_density(
        nml_cell_doc=celldoc,
        cd_id="kv",
        ion_channel="kv",
        cond_density="1.5e-3 S_per_cm2",
        erev="-81 mV",
        group_id="all",
        ion="k",
        ion_chan_def_file="channels/kv.channel.nml",
    )
    cell.add_channel_density(
        nml_cell_doc=celldoc,
        cd_id="ka",
        ion_channel="ka",
        cond_density="1.4525e-2 S_per_cm2",
        erev="-81 mV",
        group_id="all",
        ion="k",
        ion_chan_def_file="channels/ka.channel.nml",
    )
    cell.add_channel_density(
        nml_cell_doc=celldoc,
        cd_id="kst",
        ion_channel="kst",
        cond_density="2.0275e-3 S_per_cm2",
        erev="-81 mV",
        group_id="all",
        ion="k",
        ion_chan_def_file="channels/kst.channel.nml",
    )
```

(continues on next page)

(continued from previous page)

```
    # Na
    cell.add_channel_density(
        nml_cell_doc=celldoc,
        cd_id="naf",
        ion_channel="naf",
        cond_density="3.5e-2 S_per_cm2",
        erev="58 mV",
        group_id="all",
        ion="na",
        ion_chan_def_file="channels/naf.channel.nml",
    )
    cell.add_channel_density(
        nml_cell_doc=celldoc,
        cd_id="nas",
        ion_channel="nas",
        cond_density="3e-3 S_per_cm2",
        erev="58 mV",
        group_id="all",
        ion="na",
        ion_chan_def_file="channels/nas.channel.nml",
    )
```

This completes the cell. We export it to a NeuroML file.

Step 3) Testing the model

Finally, we want to test our NeuroML conversion against the original cell to see that it exhibits the same dynamics. The `test_kc.py` script runs a simple step current simulation with a single KC cell and shows its membrane potentials:

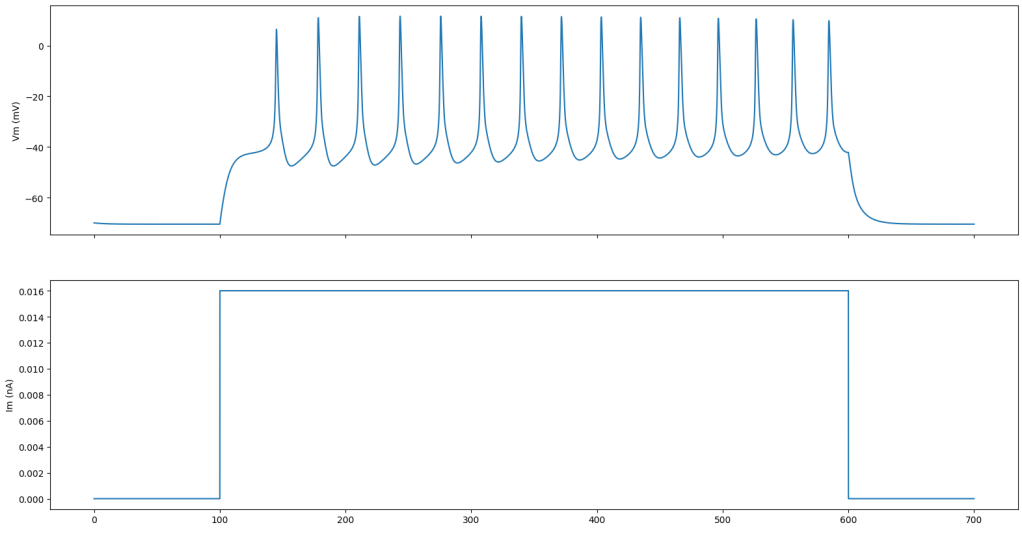


Fig. 19.7: The membrane potential of the NEURON implementation of the KC cell model with a step current.

We write a quick simulation to reproduce these using our NeuroML model:

```
def step_current_omv_kc():
    """Create a step current simulation OMV LEMS file"""
    # read the cell file, modify it, write a new one
    netdoc = read_neuroml2_file("KC.cell.nml")
    kc_cell = netdoc.cells[0]
    net = netdoc.add(neuroml.Network, id="KC_net", validate=False)
    pop = net.add(neuroml.Population, id="KC_pop", component=kc_cell.id, size=1)

    # should be same as test_kc.py
    pg = netdoc.add(
        neuroml.PulseGenerator(
            id="pg", delay="100ms", duration="500ms",
            amplitude="16pA"
        )
    )

    # Add these to cells
    input_list = net.add(
        neuroml.InputList(id="input_list", component=pg.id, populations=pop.id)
    )
    aninput = input_list.add(
        neuroml.Input(
            id="0",
            target="../%s[0]" % (pop.id),
            destination="synapses",
            segment_id="0",
        )
    )
    write_neuroml2_file(netdoc, "KC.net.nml")

    generate_lems_file_for_neuroml(
        sim_id="KC_step_test",
        target=net.id,
        neuroml_file="KC.net.nml",
        duration="700ms",
        dt="0.01ms",
        lems_file_name="LEMS_KC_step_test.xml",
        nml_doc=netdoc,
        gen_spike_saves_for_all_somas=True,
        target_dir=".",
        gen_saves_for_quantities={
            "k.dat": ["KC_pop[0]/biophys/membraneProperties/kv/iDensity"]
        },
        copy_neuroml=False
    )

    data = run_lems_with_jneuroml_neuron(
        "LEMS_KC_step_test.xml", load_saved_data=True, compile_mods=True
    )

    generate_plot(
        xvalues=[data["t"]],
        yvalues=[data["KC_pop[0]/v"]],
        title="Membrane potential: KC",
    )
```

This will generate graphs of the KC's membrane potential.

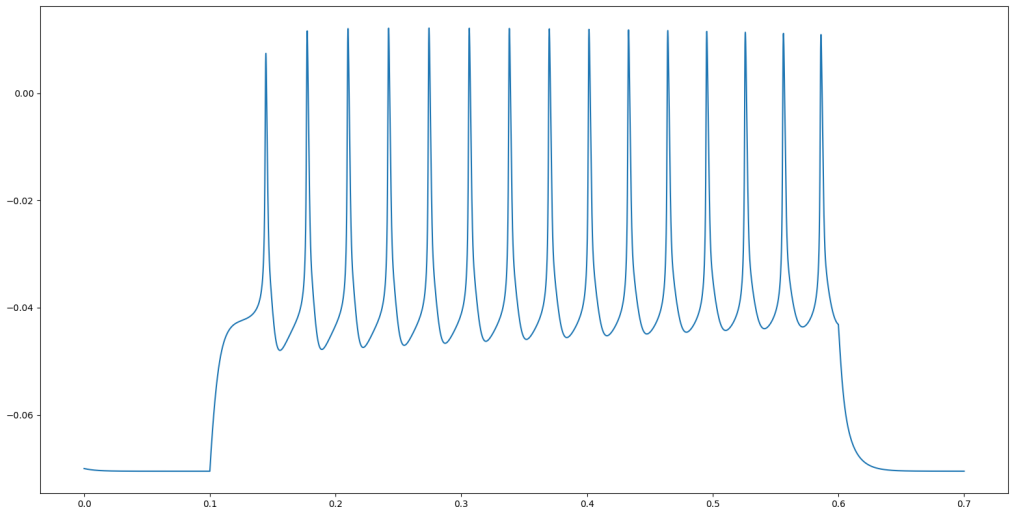


Fig. 19.8: The membrane potential of the NeuroML implementation of the KC cell model with a step current.

As we can see, the membrane potentials look very similar. In the next page, we will also set up some more validation tests to better verify that the NEURON and NeuroML implementations produce the same dynamics.

Since this writes a LEMS simulation file also, we can also run the LEMS file directly for later verification:

```
pynml LEMS_KC_step_test.xml -neuron -nogui
```

19.1.3 Adding OMV tests

Now that we have converted the cell models to NeuroML, we want to ensure that we get the same behaviour from the NeuroML converted cell model as from the original NEURON code. For this, we have the Open Source Model Validation (OMV) framework. The idea here is that we write simple test files that `omv` will run, and we provide data that `omv` can check against to see if the tests results are correct.

We will add OMV tests for the KC here.

Step 1) Getting spike data from the NEURON model

For `omv` to test the model against, we need to generate some spike data that it knows to be the expected values. The `test_kc.py` script already provides us with the spike data. So, we can run the script and note down the spike times in a Model Emergent Properties (MEP) file:

```
system: Testing a detailed cell

experiments:
  stepKC:
    expected:
      spike times: [144.45000000000556, 177.64999999997536, 210.29999999994567, 242.
↪74999999991616, 275.0499999998868, 307.19999999985754, 339.2249999998284, 371.
↪07499999979945, 402.74999999977064, 434.1499999997421, 465.2249999997138, 495.
```

(continues on next page)

(continued from previous page)

```
↪87499999968594, 525.9749999997221, 555.3499999998289, 583.8499999999326]

# generated from test-kc.py
```

Step 2) Adding tests

Next, we can write a OSB Model Test (OMT) file to test the model using the step current simulation we had written before:

```
target: LEMS_KC_step_test.xml
engine: jNeuroML_NEURON
mep: .test.kc.mep
experiments:
  stepKC:
    observables:
      spike times:
        file:
          path: KC_step_test.KC_pop.v.dat
          columns: [0,1]
          scaling: [1000,1000]
        spike detection:
          method: threshold
          threshold: -10.
        tolerance: 0.00
```

Note that we start with a tolerance of 0 here. Let us run the test and see what we get:

```
$ omv test .test.kc.jnmlneuron.omt

[omv]
[omv] Running the tests defined in .test.kc.jnmlnrn.omt
[omv] ================================================
[omv]   Found 1 experiment(s) to run on engine: jNeuroML_NEURON
[omv] PATH: :/home/asinha/.local/share/virtualenvs/neuroml-311-dev/bin
[omv] Env vars: {'PYTHONPATH': '/home/asinha/local/lib/python/site-packages', 'NEURON_
↪HOME': '/home/asinha/.local/share/virtualenvs/neuroml-311-dev', 'JNML_HOME':␣
↪PosixPath('/usr/bin')}
[omv]   Running file ./LEMS_KC_step_test.xml with jNeuroML_NEURON, env: {'NEURON_HOME
↪': '/home/asinha/.local/share/virtualenvs/neuroml-311-dev', 'JNML_HOME': PosixPath(
↪'/usr/bin')}
[omv]     Running the commands: [/usr/bin/jnml /home/asinha/Documents/02_Code/00_mine/
↪models/RayEtAl2020/NeuroML2/LEMS_KC_step_test.xml -neuron -nogui -run] in (/home/
↪asinha/Documents/02_Code/00_mine/models/RayEtAl2020/NeuroML2; cwd=/home/asinha/
↪Documents/02_Code/00_mine/models/RayEtAl2020/NeuroML2; shell=False; env={'NEURON_
↪HOME': '/home/asinha/.local/share/virtualenvs/neuroml-311-dev', 'JNML_HOME':␣
↪PosixPath('/usr/bin')})
[omv]     Commands: ['/usr/bin/jnml', '/home/asinha/Documents/02_Code/00_mine/models/
↪RayEtAl2020/NeuroML2/LEMS_KC_step_test.xml', '-neuron', '-nogui', '-run'] completed␣
↪successfully
[omv]   Success with running jNeuroML_NEURON
[omv]   Running checks for experiment: stepKC
[omv]
[omv] Comparison of
                    (observed data): [143.95, 176.76, 209.23, 241.52, 273.66, 305.68,␣
↪337.57, 369.33, 400.94, 432.38, 463.62, 494.59999999999997, 525.28, 555.59, 585.45]
```

(continues on next page)

(continued from previous page)

```
                    and
                    (expected data): [144.45000000000556, 177.64999999997536, 210.
↪29999999994567, 242.74999999991616, 275.0499999998868, 307.19999999985754, 339.
↪2249999998284, 371.07499999979945, 402.74999999977064, 434.1499999997421, 465.
↪2249999997138, 495.87499999968594, 525.9749999997221, 555.3499999998289, 583.
↪8499999999326]
                    failed against tolerance 0
[omv]   A better tolerance to try is: 0.005087969567027844
[omv]       Observable                        Test Passed
[omv]       --------------------------------------------------
[omv]       spike times                             ?
[omv]       +++++++++++++++++++++ Error info +++++++++++++++++
[omv]        Return code: 0
```

We see that there's a slight difference in the spike times we obtain from our implementation. This is not unexpected. Small variations in how the mod files are written, or how the morphology is set up can result in small variations in the spike times. `omv` will suggest a tolerance value for us to use. The smaller the tolerance, the better.

We update our file to use the suggested tolerance and re-run the test:

```
$ omv test .test.kc.jnmlnrn.omt
[omv]
[omv] Running the tests defined in .test.kc.jnmlnrn.omt
[omv] ==================================================
[omv]   Found 1 experiment(s) to run on engine: jNeuroML_NEURON
[omv] PATH: :/home/asinha/.local/share/virtualenvs/neuroml-311-dev/bin
[omv] Env vars: {'PYTHONPATH': '/home/asinha/local/lib/python/site-packages', 'NEURON_
↪HOME': '/home/asinha/.local/share/virtualenvs/neuroml-311-dev', 'JNML_HOME':␣
↪PosixPath('/usr/bin')}
[omv]   Running file ./LEMS_KC_step_test.xml with jNeuroML_NEURON, env: {'NEURON_HOME
↪': '/home/asinha/.local/share/virtualenvs/neuroml-311-dev', 'JNML_HOME': PosixPath(
↪'/usr/bin')}
[omv]     Running the commands: [/usr/bin/jnml /home/asinha/Documents/02_Code/00_mine/
↪models/RayEtAl2020/NeuroML2/LEMS_KC_step_test.xml -neuron -nogui -run] in (/home/
↪asinha/Documents/02_Code/00_mine/models/RayEtAl2020/NeuroML2; cwd=/home/asinha/
↪Documents/02_Code/00_mine/models/RayEtAl2020/NeuroML2; shell=False; env={'NEURON_
↪HOME': '/home/asinha/.local/share/virtualenvs/neuroml-311-dev', 'JNML_HOME':␣
↪PosixPath('/usr/bin')})
[omv]     Commands: ['/usr/bin/jnml', '/home/asinha/Documents/02_Code/00_mine/models/
↪RayEtAl2020/NeuroML2/LEMS_KC_step_test.xml', '-neuron', '-nogui', '-run'] completed␣
↪successfully
[omv]   Success with running jNeuroML_NEURON
[omv]   Running checks for experiment: stepKC
[omv]
[omv]       Observable                        Test Passed
[omv]       --------------------------------------------------
[omv]       spike times                             ✓
[omv]
[omv]                   ================================
[omv]                   Test passed: .test.kc.jnmlnrn.omt
```

We can also add a simple validation test that will allow `omv` to validate the various NeuroML files in the repository:

```
target: "*.c*.nml"
engine: jNeuroML_validate
```

One can run all the OMV tests in a repository at once:

```
$ omv all
[omv] Python 3. Ignoring tests for non Py3 compatible engines: False
[omv]
[omv]
[omv] Running the tests defined in ./.test.kc.jnmlnrn.omt
[omv] ================================================
[omv]   Found 1 experiment(s) to run on engine: jNeuroML_NEURON
[omv] PATH: :/home/asinha/.local/share/virtualenvs/neuroml-311-dev/bin
[omv] Env vars: {'PYTHONPATH': '/home/asinha/local/lib/python/site-packages', 'NEURON_
 ↪HOME': '/home/asinha/.local/share/virtualenvs/neuroml-311-dev', 'JNML_HOME':␣
 ↪PosixPath('/usr/bin')}
[omv]   Running file ./LEMS_KC_step_test.xml with jNeuroML_NEURON, env: {'NEURON_HOME
 ↪': '/home/asinha/.local/share/virtualenvs/neuroml-311-dev', 'JNML_HOME': PosixPath(
 ↪'/usr/bin')}
[omv]     Running the commands: [/usr/bin/jnml /home/asinha/Documents/02_Code/00_mine/
 ↪models/RayEtAl2020/NeuroML2/LEMS_KC_step_test.xml -neuron -nogui -run] in (/home/
 ↪asinha/Documents/02_Code/00_mine/models/RayEtAl2020/NeuroML2; cwd=/home/asinha/
 ↪Documents/02_Code/00_mine/models/RayEtAl2020/NeuroML2; shell=False; env={'NEURON_
 ↪HOME': '/home/asinha/.local/share/virtualenvs/neuroml-311-dev', 'JNML_HOME':␣
 ↪PosixPath('/usr/bin')})
[omv]     Commands: ['/usr/bin/jnml', '/home/asinha/Documents/02_Code/00_mine/models/
 ↪RayEtAl2020/NeuroML2/LEMS_KC_step_test.xml', '-neuron', '-nogui', '-run'] completed␣
 ↪successfully
[omv]   Success with running jNeuroML_NEURON
[omv]   Running checks for experiment: stepKC
[omv]
[omv]       Observable                          Test Passed
[omv]       ------------------------------------------------
[omv]       spike times                              ✓
[omv]
[omv]
[omv]       [ Test 1 of 2 complete - failed so far: 0 ]
[omv]
[omv]
[omv] Running the tests defined in ./.test.validate.omt
[omv] ================================================
[omv]   No mep file specified. Will only run simulation using: jNeuroML_validate
[omv]   Found 1 experiment(s) to run on engine: jNeuroML_validate
[omv]   Running with jNeuroML_validate, using: ['/usr/bin/jnml', '-validate', '/home/
 ↪asinha/Documents/02_Code/00_mine/models/RayEtAl2020/NeuroML2/GGN.morph.cell.nml', '/
 ↪home/asinha/Documents/02_Code/00_mine/models/RayEtAl2020/NeuroML2/GGN.cell.nml', '/
 ↪home/asinha/Documents/02_Code/00_mine/models/RayEtAl2020/NeuroML2/KC.cell.nml']...
[omv]     Running the commands: [/usr/bin/jnml -validate /home/asinha/Documents/02_
 ↪Code/00_mine/models/RayEtAl2020/NeuroML2/GGN.morph.cell.nml /home/asinha/Documents/
 ↪02_Code/00_mine/models/RayEtAl2020/NeuroML2/GGN.cell.nml /home/asinha/Documents/02_
 ↪Code/00_mine/models/RayEtAl2020/NeuroML2/KC.cell.nml] in (/home/asinha/Documents/02_
 ↪Code/00_mine/models/RayEtAl2020/NeuroML2; cwd=/home/asinha/Documents/02_Code/00_
 ↪mine/models/RayEtAl2020/NeuroML2; shell=False; env={'JNML_HOME': PosixPath('/usr/bin
 ↪')})
[omv]     Commands: ['/usr/bin/jnml', '-validate', '/home/asinha/Documents/02_Code/00_
 ↪mine/models/RayEtAl2020/NeuroML2/GGN.morph.cell.nml', '/home/asinha/Documents/02_
 ↪Code/00_mine/models/RayEtAl2020/NeuroML2/GGN.cell.nml', '/home/asinha/Documents/02_
 ↪Code/00_mine/models/RayEtAl2020/NeuroML2/KC.cell.nml'] completed successfully
[omv]   Running checks for experiment: Dry run
[omv]
[omv]       Observable                          Test Passed
[omv]       ------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
[omv]       dry                                          ✓
[omv]
[omv]
[omv]       [ Test 2 of 2 complete - failed so far: 0 ]
[omv]
[omv]                              -------------
[omv]                              2 test(s) run
[omv]                              -------------
[omv]
[omv]                            All tests passing!
[omv]                            ==================
```

Step 3) Setting up continuous testing on GitHub Actions

Now that we have set up OMV test, we want to set up "continuous testing" (CT). What this means is that we want these test to run automatically whenever we make any changes. On GitHub, we can do these using GitHub Actions.

To set up a GitHub Action, we need to set up a "workflow file" in the `.github/workflows` directory. We create one called `omv-ci.yml`:

```yaml
name: Continuous build using OMV

on:
  schedule:
    - cron: "1 1 1 */2 *"
  push:
    branches: [ master, development, experimental ]
  pull_request:
    branches: [ master, development, experimental ]

jobs:
  build:

    runs-on: ubuntu-latest
    strategy:
      fail-fast: false
      matrix:
        python-version: [ "3.8", "3.10"]
        engine: [ jNeuroML_validate, jNeuroML_NEURON ]

    steps:
    - uses: actions/checkout@v4

    - name: Set up Python  ${{ matrix.python-version }}
      uses: actions/setup-python@v5
      with:
        python-version:  ${{ matrix.python-version }}

    - name: Install OMV
      run: |
        pip install OSBModelValidation
        pip install scipy sympy matplotlib cython pandas tables

    - name: Run OMV tests on engine ${{ matrix.engine }}
      run: |
```

(continues on next page)

(continued from previous page)

```
        omv all -V --engine=${{ matrix.engine }}

    - name: OMV final version info
      run: |
        omv list -V # list installed engines
        env
```

This installs OMV and runs all test using it in the repository. Additionally, it tests this out on a couple of Python versions.

Now, whenever we make a change to the repository—either as a pull request, or as a direct push of a commit, these tests will be run immediately to tell us if the model is still working as it should. These can be seen in the "actions" tab in the GitHub Repository.

Part II

NeuroML events

CHAPTER

TWENTY

NEUROML OUTREACH AND EVENTS

The NeuroML community organises regular training and outreach events. Recent meetings are listed below (please see the individual pages for more details):

- *July 2024: NeuroML tutorial at CNS 2024*
- *April 2024: NeuroML hackathon at HARMONY 2024*
- *June 2022: NeuroML tutorial at CNS*2022 satellite tutorials*
- *April 2022: NeuroML development workshop at HARMONY 2022*
- *October 2021: NeuroML development workshop at COMBINE meeting*
- *August 2021: NeuroML tutorial at INCF Training Weeks*
- *July 2021: NeuroML tutorial at CNS*2021*
- *March 2021: NeuroML hackathon at HARMONY 2021*

CHAPTER

TWENTYONE

JULY 2024: NEUROML TUTORIAL AT CNS 2024

Registration for the annual meeting of the Organisation of Computational Neuroscience, CNS 2024, is open.

Please register for the CNS 2024 here.

We will be running a full day NeuroML tutorial during the annual meeting of the Organisation of Computational Neuroscience, 2024 meeting in Natal, Brazil from July 20–24.

Standardised, data-driven computational modelling with NeuroML using Open Source Brian

Data-driven models of neurons and circuits are important for understanding how the properties of membrane conductances, synapses, dendrites and the anatomical connectivity between neurons generate the complex dynamical behaviours of brain circuits in health and disease. However, even though data and models are being made publicly available in recent years and the use of standards such as Neurodata Without Borders (NWB) and NeuroML to promote FAIR (Findable, Accessible, Interoperable and Reusable) neuroscience is on the rise, development of data driven models remains hampered by the difficulty of finding appropriate data and the inherent complexity involved in their construction.

The Open Source Brain web platform (OSB) brings together data, accompanying analysis tools, and computational models in a single, scalable resource. It indexes repositories from established sources such as the DANDI data archive, the ModelDB model sharing archive, and GitHub to provide easy access to a plethora of experimental data and models, including a large number standardised in NWB and NeuroML formats

OSB also incorporates the NeuroML software ecosystem. NeuroML is an established community standard and software ecosystem that enables the development of biophysically detailed models using a declarative, simulator independent description. The software ecosystem supports all steps of the model lifecycle and allows users to automatically generate code and run their NeuroML models using a number of well established simulation engines (NEURON/NetPyNE).

Read the NeuroML pre-print here: https://www.biorxiv.org/content/10.1101/2023.12.07.570537v1

21.1 Agenda

In this tutorial, attendees will learn about:

- Finding data and models on OSB
- NeuroML and its software ecosystem
- Using NeuroML models on OSB
- Building and simulating new NeuroML models constrained by the data on OSB

We will also provide assistance with advanced tasks, and discuss new features to further aid researchers.

21.2 Times and dates

More details to follow

21.3 Registration

To take part in the tutorial, please register here for CNS 2024.

We look forward to working with the community to drive further uptake of NeuroML compliant models and tools!

CHAPTER

TWENTYTWO

APRIL 2024: NEUROML HACKATHON AT HARMONY 2024

Registration for the COMBINE initiative's HARMONY 2024 meeting is free.

Please register for the COMBINE HARMONY 2024 meeting here if you are coming to our NeuroML workshop. Registration for HARMONY is free.

We will be running a NeuroML workshop during the upcoming COMBINE network's HARMONY 2024 meeting on Tues 9th April 2024 in London, entitled:

NeuroML hackathon: convert your neuron and network models to open, standardised, reusable formats

This will be an opportunity for developers of models in computational neuroscience to get an introduction to the aims and structure of NeuroML, a guide to the tools available for building/converting their models to NeuroML, and to receive hands on help with expressing their models (or other published models they are interested in) in NeuroML format, making them more open, accessible and reusable.

22.1 Agenda

More details to follow

22.2 Times and dates

More details to follow

22.3 Registration

To take part in the workshop, please register here for the HARMONY meeting (registration is free).

22.4 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.

22.5 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

TWENTYTHREE

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.

23.1 Times and dates

- Dates: June 30, 2022
- Time: : 1400–1700 UTC

23.2 Target audience

Anyone who is already familiar with computational modelling, but is keen to standardise, share and collaboratively develop their models.

23.3 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.

23.4 Agenda

23.4.1 Part 1: Introduction to NeuroML

- Overview of NeuroML
- Introduce the NeuroML tool chain
- Introduce main documentation
- Related technologies and initiatives

23.4.2 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

TWENTYFOUR

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.

24.1 Agenda

The agenda for the meeting can be found here.

24.2 Times and dates

The workshop will take place on **Thus 28 April 2022** at 15:00-18:00 UTC (converter).

24.3 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**.

24.4 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.

24.5 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

TWENTYFIVE

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.

25.1 Times and dates

- 13 October 2021
- 8-11am PDT/11-2pm EST/4-7pm UK/5-8pm CET/8:30-11:30 IST

25.2 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.

25.3 Agenda/minutes

The agenda/minutes for the meeting can be found here.

CHAPTER

TWENTYSIX

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.

26.1 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

26.2 Target audience

Anyone who is already familiar with computational modelling, but is keen to standardise, share and collaboratively develop their models.

26.3 Agenda

26.3.1 Part 1: Introduction to NeuroML

- Overview of NeuroML
- Introduce the NeuroML tool chain
- Introduce main documentation
- Related technologies and initiatives

26.3.2 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

CHAPTER

TWENTYSEVEN

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**.

27.1 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.

27.2 Times and dates

- Friday 2nd July 1500UTC.

27.3 Registration

To take part in the tutorial, please register here for the CNS*2021 meeting.

27.4 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.

27.5 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!

CHAPTER

TWENTYEIGHT

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**.

28.1 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

28.2 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

28.3 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.

28.4 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.

28.5 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

TWENTYNINE

MARCH 2012: FOURTH NEUROML DEVELOPMENT WORKSHOP

Convergence in Computational Neuroscience 2012 Joint BrainScaleS CodeJam/NeuroML workshop, Edinburgh, 12-16th March

The NeuroML Development Workshops and the BrainScaleS (previously FACETS) CodeJams have been two important initiatives in recent years for developers of tools in computational and systems neuroscience to present their latest work, exchange ideas and work at achieving interoperability between software applications for investigating brain function. This year these groups held a joint workshop (Convergence in Computational Neuroscience) on March 12th-16th in the Informatics Forum in Edinburgh, UK. The meeting was held at the Informatics Forum in Edinburgh, UK, from 12th to 16th March 2012.

29.1 Meeting report

Note: details of the meeting activities from Wednesday 14th to Friday 16th are available on the NeuralEnsemble.org webpage.

29.1.1 Monday 12th March: NeuroML Development Workshop Day 1

Morning session: Current state of NeuroML 2 development & relationship to other initiatives

Chair: Andrew Davison

Time	Session	Speaker
09:00	**Welcome & goals of meeting** Angus welcomed attendees, thanking in particular out local organisers at University of Edinburgh, Mike Hull and Mika Pelko!	Angus Silver
09:05	**Update on latest developments in NeuroML 2/LEMS (PPT)** Padraig presented an introduction to NeuroML, starting with an overview of the modular nature of NeuroML v1.x, advantages of the use of XML, examples of neuronal models in NeuroML, current tools which support the language, (including the recently added NeuroMorpho.org and Channelpedia). The requirements for v2.0 were presented. Explicit definitions of model component behaviour allows description of the dynamics of model components in a simulator independent, machine readable way. The relationship between LEMS and NeuroML2 was discussed. A short overview of NeuroML 2.0 was given including dimensions/units. Example of adaptive exponential integrate and cell network was presented. An overview of libNeuroML was given. Export to NEURON, neuroConstruct, interaction with SBML was shown.	Padraig Gleeson
09:30	**Introduction to NineML & libNineML (PDF)** Mike gave an introduction to the NineML object model and libNineML. The INCF Task Force in Multiscale Modelling created the language, consisting of an Abstraction Layer and User Layer. Mike's presentation focused on the abstraction layer which contains many terms for the object model. Core object in the Abstraction layer were presented: ComponentClass, Interface with Parameters, Ports (AnalogPorts, EventPorts and Reduce Ports), Dynamics with StateVariables and a Regime Transition Graph (with Transitions, StateAssignments, Aliases). libNineML (in Python) loads and saves models from/to XML to/from Python that helps with code generation, turns models into NEST, NEURON and PyNN.	Mike Hull
10:00	**The COMBINE Initiative (PDF)** Nicolas presented the Combine Initiative: Standards for describing the whole lifecycle of modelling. Different communities favour different types of models that are more suited for their domain. Current standardisation efforts depend on the initial people, individual funding structure, IP issues. Specifications, API's, testsuites, etc. really need industry-grade support which is not compatible with standard academic usages and possibilities. The vision of COMBINE is to pave the space of model descriptions with coordination of standard development (without interference with the development). There are criteria for inclusion in the core COMBINE standards: new standards must be different from those already included, described in technical specification documents, free, open, developed and used by more than one team, democratically elected members, mature software support including API, and must be actively developed. COMBINE organises joint meetings replacing standards specific ones (e.g. SBML Hackathon). Next COMBINE meeting is in Toronto in August. HARMONY for hacking will be in Maastricht in May.	Nicolas le Novere
10:30	**Coffee**	
11:00	**The International Neuroinformatics Coordinating Facility** Sean presented the motivation for, current structure and the aims of the INCF. The goal of neuroscience is to understand the brain. We're at a crisis point in understanding disorders. Big Pharma is pulling out of neuroscience due to the high cost and risk of understanding these disorders. Past centuries have focused on obtaining observations. More recently, models were used to understand these observations. Today we have eScience as a new way of handling large-scale data, modelling, simulations, linking data, etc. One of the INCF's goals is to transform neuroscience into an eScience from level of molecules to clinic. Integration of databases is a goal, which requires standardized data formats. There are 16 member countries in INCF. He gave an introduction to the 4 programs from the past few years: Digital Brain Atlasing, Multi-Scale Modeling, Ontologies of Neural Structures and Standards for Datasharing. He discussed future plans for the INCF "Cyberinfrastructure", including a discussion of the planned INCF cloud "Dropbox" for data which could include metadata	Sean Hill

Afternoon session: Specification of detailed biophysical components in NeuroML 2/LEMS

Chair: Sharon Crook

Time	Session	Speaker
14:00	**Representing channels, synapses & conductance based models (PDF)** Robert gave a presentation on ways to represent synapses and conductances. He defined the Nernst equation with XML based on Hille's description. It is still not clear how some things should be done, in particular how to handle dimensions and units. Currently, dimensions are defined and then assertions about relationships among dimensions can be made. Units are not defined until NeuroML is written (with numerical values, e.g. -70mV). There was some discussion of how dimensions should be defined. Physicists solved this problem by developing SI units. Both space and no space between values and units are allowed in LEMS. There was some discussion of how events can be handled in LEMS.	Robert Cannon
14:20	**Experiences with using NeuroML 2** Avrama gave a brief report of her hands on experience with using NeuroML 2. She has a medium spiny projection neuron model which she's translating to GENESIS. She didn't want to use a GUI unless absolutely necessary and has been manually editing the XML. Many of the (non calcium dependent) channels have already been converted to NeuroML 2. She can only run single compartment versions of her model since LEMS doesn't yet support multi compartmental models. She has produced some some multi segment morphologies in NML2, even though these can't be used in LEMS based simulations yet. She requires a way to specify distance from the soma. Another difficulty is not being able to define a template (dendritic) subbranch and add it multiple times to the cell. She will add spines later. Calcium dependent channels are a work in progress in NeuroML v2.0, but some useful simulations have already been done with her developing model.	Avrama Blackwell
14:40	**Implementing cerebellar models in PyNEURON, neuroConstruct & NeuroML** Sergio is developing cerebellar models (Golgi cells and granular cells) in a network of granule layer. Solinas S, Nieus T and D`Angelo E (2010) A realistic large-scale model of the cerebellum granular layer predicts circuit spatio-temporal filtering properties. Frontiers in Cellular Neuroscience (link) gives an overview of the network. There were improvements made to the model in 2011 and it was translated to Python Neuron for parallel simulation on cluster. Added gap junctions, more realistic inputs. Python eased improvements to the model.	Sergio Solinas
15:00	**Large scale cortical models for studying LFPs (PPT)** Richard presented his work on developing large scale cortical models for studying Local Field Potentials. What network properties cause pathological dynamics? Much data comes from electrodes in vitro. Gaute Einevol's work looks at how dendritic structure affects the field potentials in a network (Linden et al Neuron 2010). Richard focused on Bush and Sejnowski J Neurosci Methods 1993 method to reduce model and see what the LFP looks like (and compared to Linden data). Then since it looked pretty good he created a network of these reduced models for simulation and analysis. Then looked at results from Utah array in Matlab. Next he'll add Gaussian connectivity and some patches and long range connections.	Richard Tomsett
15:15	**Coffee**	
15:30	**Break out sessions** • Channel and synapse specifications • Proposed structure for abstract neuron model hierarchy	
17:30	**Reconvene and presentated discussions**	
18:00	**Close**	

29.1.2 Tuesday 13th March: NeuroML Development Workshop Day 2

Morning session: Representing morphologies/support for detailed neuronal simulators/relationship to connectomics initiatives

Chair: Michael Hines

Time	Session	Speaker
09:00	**The Neural Tissue Simulator(PDF)** James presented his work on the Neural Tissue Simulator, much of which was contained in the recent publication: Kozloski J and Wagner J (2011) An ultrascalable solution to large-scale neural tissue simulation. Frontiers in Neuroinformatics. 5:15. (link). The key goals of this work are: to develop a simulator capable of testing mappings to various machine architectures, both parallel and multithreaded; to develop support for high level, abstract model definitions and simulation specifications; and to create an extensible simulator, able to map arbitrary, domain level models directly to a variety of data arrangements and computational implementations. James discussed the process of defining the model (using the Model Definition & Graph Specification Languages), how the model elements are partitioned on the computing resources, and how these elements communicate during simulation to solve the model equations. He discussed the specific case of simulating cortical columns when synapses were determined through contact detection algorithms. He also presented some results for how the simulator scales for larger networks. The Neural Tissue Simulator is not currently publicly available, but James is keen to make it available, and to build a community of users. NeuroML support is also planned	James Kozloski
09:30	**The Blue Brain Project** Eilif presented an overview of the Blue Brain Project's efforts to reverse engineer a P14 Rat non-barrel somatosensory cortical column. Based on a database of anatomical reconstructions, electrophysiology, etc. they will fill the cortical column with cells based on known location, probability distributions. Morphologies for those classes of cells are taken from library of cells called a collage with some rules about how they fit in based on constraints from reconstructions. Some of these cells have been "repaired" due to axon cuts in reconstructions. Functional circuits are also based on biological data. Electrical behaviors are based on classifications based on firing patterns observed in experiments. This is combinatorial since each morphological class has a number of possible firing patterns. They use genetic algorithm to adjust parameters which are set up based on what is known (gene expression, etc). Channelome project uses cell culture and automated patch clamp by robot and then automated model fitting for data that are then posted to Channelpedia. Channels there are available in ChannelML. Synaptic parameterization and validation for functional synapses are also based on database of recorded synaptic properties. In silico model is compared to in vitro using same protocols as experiments. There was a standards and interoperability discussion: they are mostly using custom formats other than what they use with NEURON. Eilif welcomed greater support for more widely used standards.	Eilif Muller
10:00	**Tools for the dense reconstruction of neuronal circuits** Moritz gave an overview of his recent work with Winfried Denk and Frank Briggman, which is continuing in his own lab. They have used Serial Block-Face Electron Microscopy (SBEM), to investigate the connectivity in blocks of neuronal tissue, which has been the subject of a number of recent publications, e.g. K.L. Briggman, Helmstaedter, M. and W. Denk, Wiring specificity in the direction-selectivity circuit of the mammalian retina. Nature 471, 183-188, 2011. (link). He also discussed the application KNOSSOS which was developed to facilitate the reconstruction of neuronal morphologies from such data. While this tool uses a proprietary format for storing morphologies, it is open source and Mortiz was keen to integrate the application with other tools using NeuroML.	Moritz Helmstaedter
10:30	**Coffee**	
11:00	**The OpenWorm project: Using NeuroML in a highly detailed model of C. elegans (PPT)** Stephen presented the OpenWorm project. This ambitious project aims to build an in silico model of C. elegans. This well studied system with ~1000 cells and 302 identified neurons is an ideal system with which to attempt a full simulation of a living organism down to cellular scale. Many different approaches are being	Stephen Larson

Afternoon session: Best practices when implementing support for NeuroML in simulators

Chair: Avrama Blackwell

Time	Session	Speaker
14:00	**Introduction to SED-ML (PDF)** Dagmar gave an overview of the motivation behind the development of SED-ML, the Simulation Experiment Description language , the current status of the specification, and some of the uses it has been put to so far. It compliments a model description in SBML or NeuroML and allow specification of the simulation algorithm used to run the model, any changes made to the parameters specified in the model description, the simulation duration, what variables were saved during the simulation, and how that data was processed. In SED-ML you can define a uniform time course with an initial time and start and end time. This needs to be expanded to other possible time courses. Multiple tasks (simulations) can be defined. For example, run the original database model and the changed model. Output can be set up as 2D or 3D plots or a datatable. SED-ML has an elected board of editors. Contribution to SED-ML is encouraged. Sourceforge can be used for feature requests and this will move forward as people contribute.	Dagmar Waltemath
14:20	**Introduction to CNO: an ontology for annotating computational neuroscience models (PDF)** Yann presented an introduction to CNO: an ontology for annotating computational neuroscience models. All classes must have a unique identifier, a label (name) and a human-readable definition. Relationships among classes are specified with relations. Examples are subsumption relations, associative relations, etc. We then can associate this semantic information with parts of XML files.	Yann le Franc
14:40	**NeuroLex & NIF update (PPT)** Stephen gave us an update on NeuroLex and the Neuroscience Information Framework (NIF). We need for an online parts list for the brain. NeuroLex is built on Wiki technology with extra functionality to create structured knowledge where anyone can create or edit. It currently as about 18,000 concepts. NIF funds curators from NIH money and also looks for volunteers. In the future they want to dominate Google searches with NeuroLex terms. Looking to Yelp for how they display info including images and related queries and such. Some place for community comments. Another goal is to expose high quality linked data with example of an open linked data graph.	Stephen Larson
15:00	**libSBML and SBML L3 (PDF)** Sarah gave a brief overview of libSBML and SBML Level 3. libSBML, which provides an API for creating, editing and saving SBML in many languages (e.g. C++, Python, Java, Ruby, Perl) has been instrumental in the growth of the number of applications supporting SBML. SBML Level 3 has a modular architecture, featuring a core specification (roughly in line with previous SBML releases) and a number of specialist packages, which applications can choose to support or not. Examples of these packages incluse layout for storing the spatial topology of a model's network diagram, comp for defining how a model is composed from other models and spatial for describing models that involve a spatial component. libSBML already has a generic framework to support extensions for generic packages.	Sarah Keating
15:15	**Coffee**	
15:30	**Break out sessions** • How best to map generic model descriptions to a given simulator • Support for morphologies	
17:30	**Reconvene and presentated discussions**	
18:00	**Close of meeting**	

29.1.3 Wednesday 14th March–Friday 16th March

Full details of the meetings from Wed-Fri are available on the NeuralEnsemble.org webpage for the meeting.

29.2 Funding

The NeuroML part of this workshop was made possible with funding from:

- The National Institutes of Health
- Wellcome
- The UK Neuroinformatics Node
- The International Neuroinformatics Co-ordinating Facility (INCF)

CHAPTER

THIRTY

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

THIRTYONE

GETTING IN TOUCH

We're happy to talk with users, developers and modellers about using NeuroML in their work.

31.1 Mailing list

For announcements, general discussion, queries, and troubleshooting related to NeuroML please use the mailing list: https://lists.sourceforge.net/lists/listinfo/neuroml-technology.

31.2 Chat channels

A Gitter/Matrix chat channels for queries are also available. One can access it either via Gitter or Matrix/Element.

- Gitter
- Matrix/Element

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.

31.3 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*.

31.4 Social media

You can follow NeuroML related updates on Twitter at @NeuroML.

CHAPTER

THIRTYTWO

OVERVIEW OF STANDARDS IN NEUROSCIENCE

In biology, several community standards have been developed to describe experimental data (e.g. Brain Imaging Data Structure BIDS, Neurodata Without Borders NWB and computational models (e.g. Systems Biology Markup Language SBML, CellML, Scalable Open Network Architecture TemplAte SONATA, PyNN, and Neural Open Markup Language (NeuroML). These standards have enabled open and interoperable ecosystems of software applications, libraries, and databases to emerge, facilitating the sharing of research outputs, an endeavour encouraged by a growing number of funding agencies and scientific journals.

32.1 NeuroML as a standard

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.

32.1.1 Endorsed INCF standard

The mission of INCF is to promote the uptake of FAIR data management practices in neuroscience through the development of standards and best practices that support open, FAIR, and citable neuroscience. INCF also provides training on how standards and best practices facilitate reproducibility and enable the sharing of data and code.

NeuroML is an INCF endorsed standard.

32.1.2 COMBINE standard

The "COmputational Modeling in BIology NEtwork" (COMBINE) is an initiative to coordinate the development of the various community standards and formats for computational models. By doing so, it is expected that the federated projects will develop a set of interoperable and non-overlapping standards covering all aspects of modelling in biology. The global COMBINE effort is led by the COMBINE Coordination Board.

Building on the experience of mature projects, which already have stable specifications, software support, user-base and community governance, COMBINE will help foster or support fledgling efforts aimed at filling gaps or new needs. As those efforts mature, they may become part of the core set of COMBINE standards.

One of the initial activities of COMBINE is to coordinate the organization of scientific and technical events common to several standards. Those events, as others related to our field of research are gathered in a calendar.

NeuroML is a COMBINE official standard.

CHAPTER

THIRTYTHREE

A BRIEF HISTORY OF NEUROML

33.1 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.

33.2 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.

For converting NeuroML v1 models/files to NeuroML2, users can use *neuroConstruct*.

33.3 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 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.

33.4 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

THIRTYFOUR

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.

34.1 Current Editorial Board

The five current members of the NeuroML Editorial Board are:

- Padraig Gleeson
- Boris Marin
- Sotirios Panagiotou
- Subhasis Ray
- Ankur Sinha

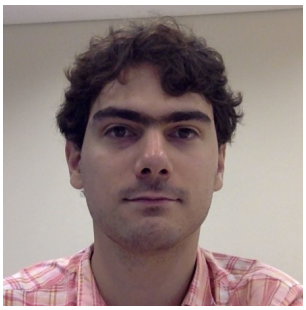
Ankur Sinha and Subhasis Ray were elected for three year terms in 2024 (2025–2027). Padraig Gleeson, Boris Marin and Sotirios Panagiotou were elected for three year terms in 2022 (2023–2025).



Sotirios Panagiotou PhD candidate Erasmus Medical Center Rotterdam, Netherlands Website



Padraig Gleeson University College London UK Website



Boris Marin Universidade Federal do ABC Brazil Website



Subhasis Ray TCG CREST Kolkata, India Website



Ankur Sinha University College London UK Website

Information on past editors and elections can be found *here*.

34.2 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.

34.3 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.

34.4 History of the NeuroML Editorial Board

This page documents the previous members of the NeuroML Board.

34.4.1 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) was the returning officer for this initial election.

34.4.2 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) was the returning officer for this election.
- Results were announced here.

34.4.3 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) was the returning officer for this election.

34.4.4 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.

34.4.5 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.

34.4.6 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.

34.4.7 Election of editors (2024)

The seventh election of an editorial board for NeuroML took place in Nov/Dec 2023.

- 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, Sotirios Panagiotou and Subhasis Ray were nominated. Padraig Gleeson, Boris Marin, Sotirios Panagiotou received highest votes and were elected to serve as editors.
- Cengiz Gunay was the returning officer for this election.

34.4.8 Election of editors (2025)

The eight election of an editorial board for NeuroML took place in Nov/Dec 2024.

- 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.
- Ankur Sinha and Subhasis Ray were nominated. Ankur Sinha and Subhasis were both elected to serve as editors.
- Cengiz Gunay was the returning officer for this election.

34.5 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

THIRTYFIVE

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.

35.1 Current Members



Upi Bhalla NCBS Bangalore, India Website



Avrama Blackwell Krasnow Institute of Advanced Studies George Mason University, USA Website



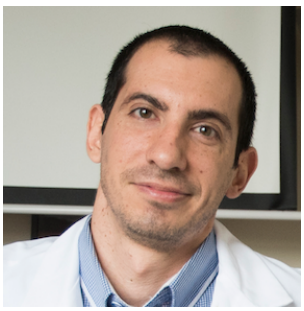
Hugo Cornells K.U. Leuven Belgium Website



Sharon Crook Arizona State University USA Website



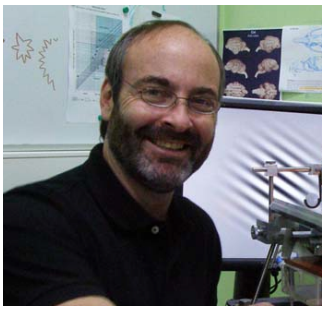
Andrew Davison CNRS, Gif-sur-Yvette France Website



Salvador Dura-Bernal SUNY Downstate Brooklyn, USA 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

35.2 Past Members

(Note: past members who are currently members of the *NeuroML Editorial Board* are not listed.)

- Robert Cannon

CHAPTER

THIRTYSIX

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

THIRTYSEVEN

OUTREACH AND TRAINING

The NeuroML community has a strong record of participating in training and outreach activities. Information on tutorials and workshops can be seen in the Events pages.

37.1 Google summer of Code

The NeuroML community participates annually in the Google Summer of Code under the INCF organisation. Projects are centred around the standardisation of published models in NeuroML to make these standardised versions available on the Open Source Brain platform and improving the NeuroML tools wherever possible.

37.1.1 2025

This round of Google Summer of Code is currently in progress. We are participating under the INCF.

Hengye Zhu is working with us on the project, "Developing standardised biophysically detailed neuronal circuit models using NeuroML/PyNN/Open Source Brain".

37.1.2 2024

- Aditya Pandey: Implementation of an SWC to NeuroML converter in PyNeuroML.
- Ioannis Daras: Incorporation of new features into an advanced cross platform 3D viewer for NeuroML cells and networks.

37.1.3 2022

- Anuja Negi: Simulating multiscale models of the mouse visual cortex in NeuroML
- Shayan Shafquat: Open source, cross simulator, large scale cortical models in NeuroML

37.1.4 Other past GSoC projects

- 2020: Ronaldo Nunes: Conversion of large scale cortical models into PyNN/NeuroML
- 2018: Jessica Dafflon: Implementation of Two Neural Mass Models on the Open Source Brain Platform
- 2017: Andras Ecker: Conversion of a large scale hippocampal network model to NeuroML
- 2016: Rokas Stanislavos: Building cortical network models in NeuroML2 using procedural and declarative programming approaches
- 2015: Justas Birgiolas: Convert several large-scale thalamocortical models to NeuroML & PyNN
- 2014: Ramón Martínez Mayorquin: Open source, cross simulator, large scale cortical models
- 2013: Vitor Chaud: Open source, cross simulator, models of cortical circuits
- 2012: Mike Vella: Simulator-independent Python API for multi-compartmental modeling

CHAPTER

THIRTYEIGHT

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 29/07/24. See also the current *NeuroML Editorial Board* and the *Scientific Committee*. The list of repositories can be seen on *the repositories page*.

- @pgleeson
- @JustasB
- @sanjayankur31
- @tarelli
- @borismarin
- @mattearnshaw
- @adrianq
- @epiasini
- @RokasSt
- @dilawar
- @russelljjarvis
- @FinnK
- @kapilkd13
- @gidili
- @wvangeit
- @hugh-osborne
- @lungd
- @orena1
- @ChihweiLHBird
- @rgerkin
- @allcontributors[bot]
- @shayanshafquat
- @ccluri
- @mwatts15
- @34383c

- @andrisecker
- @jrieke
- @scrook
- @jonc125
- @davidt0x
- @doorkn-b
- @ramcdougal
- @keszybz
- @avrama
- @jsnowacki
- @rocapp
- @prex1030
- @frothga
- @fywang
- @DavidPoliakoff
- @jonsalaz
- @cewarr
- @Ermentrout
- @miscco
- @lisphacker
- @dokato
- @robertcannon
- @JLLeitschuh
- @waffle-iron
- @volker01
- @theiera
- @apdavison
- @vitorchaud
- @hglabska
- @adityagilra
- @kkalou
- @osb-admin
- @tcstewar
- @jplang
- @marutosi
- @edavis10

- @winterheart
- @jbbarth
- @jgoerzen
- @h-mayorquin
- @robertvi
- @stellaprins
- @harveymannering
- @acardona
- @rizland
- @SteMasoli
- @MFarinella
- @yates9
- @Neurophile
- @EmmanuelleChaigneau
- @slarson
- @usama57
- @dasj-osb
- @muratkrty
- @arnabiswas
- @souravsingh
- @pshwetank
- @yuumi15
- @albada
- @jhnnsnk
- @anujanegi
- @nikiita013
- @kedoxey
- @ChristophMetzner
- @evakh
- @BiaDarkia
- @Mcdonoughd
- @sdrsd
- @salvadord
- @kperun
- @ddanny
- @jrmartin

- @koppen
- @tofi86
- @yuba
- @smith
- @abronte
- @dpbus
- @korun
- @wronglink
- @clairvy
- @RobFerrer
- @ryansch
- @yujideveloper
- @chantra
- @mattmueller
- @tfliss
- @BKriener
- @JessyD
- @jasonxanthakis
- @jimboH
- @vrhaynes
- @abhineet99
- @filippomc
- @kmantel
- @zsinnema
- @Muhaddatha
- @D-GopalKrishna
- @Aiga115
- @vidhya-metacell
- @vidhya-longani
- @etowett
- @SimaoBolota-MetaCell
- @Salam-Dalloul
- @ronaldonunes
- @pfeffer90
- @stephprince
- @hrani

- @upibhalla
- @asiaszmek
- @subhacom
- @aviralg
- @malav4994
- @bhanu151
- @analkumar2
- @bitdeli-chef
- @physicalist
- @anhknguyen96
- @Daksh1603
- @yarikoptic
- @musicinmybrain
- @tommorse
- @neuroman314
- @pramodk
- @alexsavulescu
- @ferdonline
- @olupton
- @vellamike
- @clbarnes
- @mattions
- @lebedov
- @baladkb
- @jefferis
- @mstimberg
- @arosh
- @unidesigner
- @lucasklmn

CHAPTER

THIRTYNINE

NEUROML REPOSITORIES

This page lists repositories related to NeuroML, listed in no particular order. It is generated periodically, most recently on 29/07/24. A complete list of contributors can be seen *here*.

For the status of tests on standardized NeuroML models on Open Source Brain, please see this page: https://github.com/OpenSourceBrain/.github/blob/main/testsheet/README.md.

- NeuroML/Cvapp-NeuroMorpho.org
- 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
- NeuroML/.github
- NeuroML/stochdiff

- NeuroML/xppy
- NeuroML/n2a
- NeuroML/xppaut
- LEMS/pylems
- LEMS/jLEMS
- LEMS/LEMS
- LEMS/org.lemsml.model
- LEMS/expr-parser
- LEMS/lems-domogen-maven-plugin
- OpenSourceBrain/CA1PyramidalCell
- OpenSourceBrain/OSB_API
- OpenSourceBrain/CerebellarNucleusNeuron
- OpenSourceBrain/GranCellLayer
- OpenSourceBrain/GranCellRothmanIf
- OpenSourceBrain/GranCellSolinasEtAl10
- OpenSourceBrain/GranuleCell
- OpenSourceBrain/GranuleCellVSCS
- OpenSourceBrain/IzhikevichModel
- OpenSourceBrain/MainenEtAl_PyramidalCell
- OpenSourceBrain/PurkinjeCell
- OpenSourceBrain/RothmanEtAl_KoleEtAl_PyrCell
- OpenSourceBrain/SolinasEtAl-GolgiCell
- OpenSourceBrain/VervaekeEtAl-GolgiCellNetwork
- OpenSourceBrain/Thalamocortical
- OpenSourceBrain/PyloricNetwork
- OpenSourceBrain/MorrisLecarModel
- OpenSourceBrain/StriatalSpinyProjectionNeuron
- OpenSourceBrain/CelegansNeuromechanicalGaitModulation
- OpenSourceBrain/PospischilEtAl2008
- OpenSourceBrain/Drosophila_Projection_Neuron
- OpenSourceBrain/NengoNeuroML
- OpenSourceBrain/NeuroElectroSciUnit
- OpenSourceBrain/NeuroMorpho
- OpenSourceBrain/redmine
- OpenSourceBrain/CSAShowcase
- OpenSourceBrain/L5bPyrCellHayEtAl2011

- OpenSourceBrain/neuroConstructShowcase
- OpenSourceBrain/PinskyRinzelModel
- OpenSourceBrain/Brunel2000
- OpenSourceBrain/NineMLShowcase
- OpenSourceBrain/BrianShowcase
- OpenSourceBrain/MUSICShowcase
- OpenSourceBrain/SBMLShowcase
- OpenSourceBrain/HindmarshRose1984
- OpenSourceBrain/FitzHugh-Nagumo
- OpenSourceBrain/BluehiveShowcase
- OpenSourceBrain/NIFShowcase
- OpenSourceBrain/CATMAIDShowcase
- OpenSourceBrain/VogelsSprekelerEtAl2011
- OpenSourceBrain/cereb_grc_mc
- OpenSourceBrain/NSGPortalShowcase
- OpenSourceBrain/ACnet2
- OpenSourceBrain/CNOShowcase
- OpenSourceBrain/DentateGyrus2005
- OpenSourceBrain/ghk-nernst
- OpenSourceBrain/korngreen-pyramidal
- OpenSourceBrain/neuroConstruct_to_be_deleted
- OpenSourceBrain/NEURONShowcase
- OpenSourceBrain/LarkumEtAl2009
- OpenSourceBrain/GPUShowcase
- OpenSourceBrain/FPGAShowcase
- OpenSourceBrain/FarinellaEtAl_NMDAspikes
- OpenSourceBrain/BlueBrainProjectShowcase
- OpenSourceBrain/OSB_Metadata
- OpenSourceBrain/OSB_Status
- OpenSourceBrain/osb-model-validation
- OpenSourceBrain/ModelDBShowcase
- OpenSourceBrain/OSB_Documentation
- OpenSourceBrain/V1NetworkModels
- OpenSourceBrain/PotjansDiesmann2014
- OpenSourceBrain/AllenInstituteNeuroML
- OpenSourceBrain/StochasticityShowcase

- OpenSourceBrain/VierlingClaassenEtAl2010
- OpenSourceBrain/OSB_Videos
- OpenSourceBrain/Contribute
- OpenSourceBrain/SmithEtAl2013-L23DendriticSpikes
- OpenSourceBrain/MiglioreEtAl14_OlfactoryBulb3D
- OpenSourceBrain/WeilerEtAl08-LaminarCortex
- OpenSourceBrain/Cerebellum3DDemo
- OpenSourceBrain/dLGNinterneuronHalnesEtAl2011
- OpenSourceBrain/GranularLayerSolinasNieusDAngelo2010
- OpenSourceBrain/CommunityModellingCA1
- OpenSourceBrain/FergusonEtAl2014-CA1PyrCell
- OpenSourceBrain/VERTEXShowcase
- OpenSourceBrain/MOOSEShowcase
- OpenSourceBrain/recaptcha
- OpenSourceBrain/OpenCortex
- OpenSourceBrain/OlfactoryTest
- OpenSourceBrain/FergusonEtAl2013-PVFastFiringCell
- OpenSourceBrain/M1NetworkModel
- OpenSourceBrain/NESTShowcase
- OpenSourceBrain/Hippocampus3DDemo
- OpenSourceBrain/OSB_Samples
- OpenSourceBrain/org.geppetto.persistence
- OpenSourceBrain/redmine_github_hook
- OpenSourceBrain/PyNNShowcase
- OpenSourceBrain/neuralensemble-docker
- OpenSourceBrain/WangBuzsaki1996
- OpenSourceBrain/GolgiCellDendGapJunctions
- OpenSourceBrain/NetPyNEShowcase
- OpenSourceBrain/Poirazi2003-CA1PyramidalCell
- OpenSourceBrain/Ferrante2009-DentateGyrusGranuleCell
- OpenSourceBrain/Hemond2008-CA3PyramidalCell
- OpenSourceBrain/geppetto-osb
- OpenSourceBrain/SynapticIntegration
- OpenSourceBrain/NorenbergEtAl2010_DGBasketCell
- OpenSourceBrain/TheVirtualBrainShowcase
- OpenSourceBrain/tutorials

- OpenSourceBrain/BartosEtAl2002
- OpenSourceBrain/destexhe_jcns_2009
- OpenSourceBrain/MouseLightShowcase
- OpenSourceBrain/BonoClopath2017
- OpenSourceBrain/SpineShowcase
- OpenSourceBrain/SadehEtAl2017-InhibitionStabilizedNetworks
- OpenSourceBrain/WilsonCowan
- OpenSourceBrain/MultiscaleISN
- OpenSourceBrain/PINGnets
- OpenSourceBrain/DynaSimShowcase
- OpenSourceBrain/Amacrine
- OpenSourceBrain/del-Molino2017
- OpenSourceBrain/MejiasEtAl2016
- OpenSourceBrain/NWBShowcase
- OpenSourceBrain/L23PyramidalCellTutorial
- OpenSourceBrain/ConnectivityShowcase
- OpenSourceBrain/IonChannelGenealogyShowcase
- OpenSourceBrain/TobinEtAl2017
- OpenSourceBrain/CalciumImagingDriftingGrating
- OpenSourceBrain/PsyNeuLinkShowcase
- OpenSourceBrain/EbnerEtAl2019
- OpenSourceBrain/OSBv2
- OpenSourceBrain/JoglekarEtAl18
- OpenSourceBrain/BahlEtAl2012_ReducedL5PyrCell
- OpenSourceBrain/SussilloAndAbbott2009
- OpenSourceBrain/DemirtasEtAl19
- OpenSourceBrain/SinusoidalVoltageProtocols
- OpenSourceBrain/ONNXShowcase
- OpenSourceBrain/ServiceStatus
- OpenSourceBrain/ArborShowcase
- OpenSourceBrain/hh-testing
- OpenSourceBrain/HNN
- OpenSourceBrain/Documentation
- OpenSourceBrain/GSoC_2021_OSB_NWB
- OpenSourceBrain/EDENShowcase
- OpenSourceBrain/BindsNETShowcase

- OpenSourceBrain/moose-core
- OpenSourceBrain/.github
- OpenSourceBrain/OSBv2_Showcase
- OpenSourceBrain/DANDIArchiveShowcase
- OpenSourceBrain/NeuroDataShare
- OpenSourceBrain/OSB_homepage
- OpenSourceBrain/pynsgr
- OpenSourceBrain/BarrelCortexInSilicoShowcase
- OpenSourceBrain/Maki-MarttunenEtAl2020
- NeuralEnsemble/libNeuroML
- scrook/neuroml-db
- NeuralEnsemble/neurotune
- NeuralEnsemble/pyelectro

CHAPTER

FORTY

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

CHAPTER

FORTYONE

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*.

The following project Kanban boards are used to consolidate issues:

- NeuroML: for all repositories under the NeuroML GitHub organization
- LEMS: for all repositories under the LEMS GitHub organization
- NeuralEnsemble: for all NeuroML related repositories in the Neural Ensemble GitHub organization

41.1 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.

41.1.1 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.

41.1.2 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.

41.1.3 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.

41.1.4 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`.

41.1.5 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).

41.1.6 Code style: Java

TODO

41.1.7 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.

41.1.8 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.

41.1.9 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.

41.2 Release Process

41.2.1 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.

41.2.2 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
Commit final stable work in development branches	v2.3
Make releases (not just tag - generates DOI) previous development versions of individual repos	v2.3
Increment all version numbers - to distinguish release from previous development version	v2.3
Test all development branches - rerun GitHub Actions at least once	v2.3
Recheck all READMEs & docs	v2.3
Run & check test.py in NeuroML2 repo	v2.3
Check through issues for closed & easily closable ones	v2.3
Update version in *documentation pages*	v2.3
Update HISTORY.md in NeuroML2	v2.3
pylems: Update README; Merge to master; Tag release; Release to pip	v2.3
libNeuroML: Update README; Retest; Merge to master; Tag release; Release to pip; Check installation docs	v2.3
pyNeuroML: Update Readme; Tag release; Release to pip	v2.3
NeuroMLlite: Update Readme; Tag release; Release to pip	v2.3
Java repositories (*jNeuroML*, org.neuroml.* etc.): Merge development to master; Tag releases	v2.3
Rebuild jNeuroML & commit to jNeuroMLJar and use latest for jNeuroML for OMV	v2.3
Add new binary release on https://github.com/NeuroML/jNeuroML/releases	v2.3
Update version used in neuroConstruct	v2.3
Update docs on http://docs.neuroml.org	v2.3
Update version on COMBINE website	v2.2
ANNOUNCE (mailing list, Twitter)	v2.2
Increment version numbers in all development branches	v2.3
DOI on Zenodo	v2.3
Update NeuroML milestones	v2.2
New release of neuroConstruct	v2.3
Test toolchain on Windows…	v2.0

41.3 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 the 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.

41.3.1 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.

41.3.2 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.

41.3.3 Updating the Java API: org.neuroml.model

TODO: Document what needs to be done for https://github.com/NeuroML/org.neuroml.model

41.3.4 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

41.3.5 Updating the C++ API

TODO: Document what needs to be done for https://github.com/NeuroML/NeuroML_API/

CHAPTER

FORTYTWO

INTERACTION WITH OTHER LANGUAGES AND STANDARDS

Needs work

TODO: Add more information to each of these

42.1 PyNN

https://github.com/NeuroML/NeuroML2/issues/73

42.2 SBML

https://github.com/OpenSourceBrain/SBMLShowcase

42.3 Sonata

https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1007696

42.4 NineML & SpineML

https://github.com/OpenSourceBrain/NineMLShowcase

42.5 ModECI MDF

http://www.modeci.org/

42.6 SWC

http://www.neuronland.org/NLMorphologyConverter/MorphologyFormats/SWC/Spec.html http://www.neuromorpho.org/myfaq.jsp

Part V

Reference

CHAPTER

FORTYTHREE

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

FORTYFOUR

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 Rumpe, Marcel Stimberg, Alan B. Stokes, Adam Tomkins, Guido Trensch, Marmaduke Woodman, and Jochen Martin Eppler. 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.

[RAS20] Subhasis Ray, Zane N. Aldworth, and Mark A. Stopfer. Feedback inhibition and its control in an insect olfactory circuit. *eLife*, 9:e53281, 2020. URL: https://doi.org/10.7554/eLife.53281, doi:10.7554/eLife.53281.

[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.

[SGM+25] Ankur Sinha, Padraig Gleeson, Bóris Marin, Salvador Dura-Bernal, Sotirios Panagiotou, Sharon Crook, Matteo Cantarelli, Robert C. Cannon, Andrew P. Davison, Harsha Gurnani, and Robin Angus Silver. The neuroml ecosystem for standardized multi-scale modeling in neuroscience. *eLife*, 13:RP95135, 2025. URL: https://doi.org/10.7554/eLife.95135, doi:10.7554/eLife.95135.

[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.