EDEN user's guide

Release 0.2.4a4

EDEN authors and contributors

Jul 03, 2026

EDEN (*E*xtensible *D*ynamics *E*ngine for *N*etworks) is a simulation program for spiking neural networks[1] that takes models described in NeuroML[2] and generates the simulated behaviour of these networks.

It is best used as part of a neural modelling workflow, between the stage of *generating* the model to simulate, and *analysis* of the simulation's results.

To learn how to use EDEN, check out the Quickstart (page 9) with a classic Hodgkin-Huxley neuron, and browse the user's guide for more about modelling and usage (starting with NeuroML basics (page 13)).

[1] https://en.wikipedia.org/wiki/Spiking_neural_network
[2] https://docs.neuroml.org/

CONTENTS

Installing 1

Contact us 3

A Introduction to NeuroML 5

1 Quickstart 9
- 1.1 Get EDEN . . . 9
- 1.2 Get a NeuroML model . . . 9
- 1.3 Run the NeuroML model with EDEN . . . 10
- 1.4 Show the results . . . 10
- 1.5 Done! . . . 11
  - 1.5.1 Appendix . . . 11

2 NeuroML basics 13
- 2.1 Creating a simulation . . . 13
- 2.2 Adding a neuron and recording output . . . 14
- 2.3 Adding stimuli . . . 17
- 2.4 Adding more neurons and synapses . . . 19
- 2.5 Adding more cell types and spike sources . . . 21
  - 2.5.1 Adding a new cell type and population . . . 21
  - 2.5.2 Adding abstract spike sources and applying them to the network . . . 24
- 2.6 Adding graded interactions . . . 27
- 2.7 NeuroML file structure conventions . . . 30

3 NeuroML and spatially detailed cells 33
- 3.1 Putting together a cell type . . . 33
- 3.2 Peeking at the neuron model's structure . . . 34
- 3.3 The morphology of the cell . . . 35
  - 3.3.1 Segments . . . 36
  - 3.3.2 Segment groups . . . 39
  - 3.3.3 Simulation aspect: Discretisation into compartments . . . 40
  - 3.3.4 Visualisation example . . . 41
- 3.4 The biophysics of the cell . . . 42
  - 3.4.1 Electrotonic properties . . . 43
  - 3.4.2 Adding ion channels and other leaks to the membrane . . . 43
  - 3.4.3 Adding concentration models near the membrane . . . 51
  - 3.4.4 Attaching probes and synapses on a spatially-modelled cell . . . 52
  - 3.4.5 Recap . . . 52
- 3.5 Example: Modelling and simulating a stretch of neural cable . . . 52
  - 3.5.1 Modelling the cell's morphology . . . 53
  - 3.5.2 Modelling the membrane mechanisms . . . 53
  - 3.5.3 Modelling the biophysics distributions . . . 54
  - 3.5.4 Modelling the experimental rig . . . 55

3.5.5 Recording over the cell's full extent . . . 55
3.5.6 Running the simulation and displaying results . . . 56
3.6 More resources on spatially-detailed modelling . . . 58
3.6.1 Multi-compartment modelling in other simulators . . . 58
3.7 Next steps . . . 59

4 Extending NeuroML with LEMS components 61
4.1 What is a LEMS component? . . . 61
4.2 What is a LEMS `ComponentType`? . . . 62
4.2.1 The "components" of a ComponentType . . . 62
4.3 How can LEMS components be used in NeuroML models? . . . 65
4.3.1 Overview of LEMS mechanisms . . . 66
4.3.2 Commonly available quantities . . . 66
4.3.3 Custom point neurons . . . 66
4.3.4 Custom synapses . . . 69
4.3.5 Custom input sources . . . 76
4.3.6 Custom ion channels, gates and rates . . . 78
4.3.7 Custom ion concentration models . . . 81

B Beyond NeuroML: EDEN-specific extensions 83

5 Extended LEMS paths 87
5.1 LEMS paths for cell locations . . . 87
5.2 LEMS paths for input list elements . . . 88
5.3 LEMS paths for synaptic projection elements . . . 88
5.4 LEMS paths for input streams . . . 88

6 Custom per-element variability 89
6.1 The `EdenCustomSetup` file specification . . . 89
6.1.1 General structure . . . 90
6.1.2 `set cell` statement . . . 90
6.1.3 `set synapse` statement . . . 92
6.1.4 `set input` statement . . . 92
6.1.5 Comment lines and inline comments . . . 92
6.1.6 Setting `VariableRequirements` . . . 93
6.2 Example: A network with variability over neurons, synapses and probes . . . 93
6.2.1 More examples . . . 97

7 Arbitrary parameters on spatially detailed cells 99
7.1 Sampling the cell's morphology . . . 99
7.2 Variability as distance from a plane . . . 101
7.3 Variability as distance from an axle . . . 102
7.4 Variability as by distance from a point in space . . . 103
7.5 Randomised parameters . . . 104
7.6 Aside for NEURON users . . . 105

8 LEMS extension: `<VariableRequirement>` 107
8.1 Usage . . . 107
8.2 Examples . . . 108

9 Extended input-output options 109
9.1 General concepts . . . 109
9.1.1 Data URL's . . . 110
9.1.2 Data format . . . 110
9.2 Extended output . . . 110
9.2.1 Time series with `EdenOutputFile` . . . 110
9.2.2 Event series with `EdenEventOutputFile` . . . 114
9.3 Extended input . . . 115

9.3.1 Time series with `EdenTimeSeriesReader` . . . 116
9.3.2 Event series with `EdenEventSetReader` . . . 118
9.3.3 On shorting readers and writers . . . 120

10 NeuroML extension: Multiple flows per neuron **121**
10.1 The `<DerivedVariable>` tag revisited . . . 121

11 LEMS extension: `<WritableRequirement>` **123**
11.1 The `<WritableRequirement>` tag . . . 123

C Usage examples 125

12 Tutorial: A network of detailed cells **129**
12.1 Constructing the network . . . 129
12.1.1 Placing neurons . . . 129
12.1.2 Adding synapses . . . 131
12.1.3 Adding stimuli . . . 133
12.2 Generating NeuroML for the model . . . 133
12.3 Displaying per-soma activity . . . 137
12.4 Displaying whole-neuron activity . . . 140

13 Example: Local field potential **147**
13.1 Get the model . . . 147
13.2 Prepare and run the simulation . . . 148
13.3 Post-process simulation data to get I_m . . . 150
13.3.1 Calculating axial current . . . 151
13.3.2 Calculating injected current . . . 152
13.3.3 Calculating membrane current . . . 152
13.3.4 Visualisation . . . 154
13.4 Post-process I_m to get the LFP . . . 154
13.5 Visualisation . . . 155
13.5.1 In 4-D . . . 155
13.5.2 Another popular way . . . 156
13.5.3 More ideas . . . 158

14 Example: Imposed electric and magnetic fields **159**
14.1 Get a cell model . . . 159
14.2 Prepare to run simulations . . . 160
14.3 Physics . . . 161
14.3.1 Classic cable equation . . . 163
14.4 Steady field shape . . . 164
14.4.1 Uniform electric field . . . 164
14.4.2 Disc electrode . . . 166
14.5 Magnetic field by a coil . . . 169
14.5.1 More physics . . . 169
14.6 General time-varying field . . . 169
14.6.1 Magnetic field by a moving coil . . . 170
14.7 Further considerations . . . 170
14.7.1 Feature size of the neurons and the fields . . . 170
14.7.2 Detail of the surroundings . . . 170
14.7.3 Ephaptic coupling . . . 170
14.8 Related resources . . . 171

15 Example: Playing Pong **173**
15.1 Modelling the play environment . . . 173
15.1.1 The field . . . 173
15.1.2 The paddles . . . 173
15.2 Showing the playing field . . . 174

15.2.1 Showing the ball's trace through the field . . . 175
15.3 Game Dynamics . . . 176
15.3.1 Walls . . . 176
15.3.2 Paddle movement . . . 176
15.3.3 Paddle hits . . . 176
15.3.4 Serving . . . 177
15.4 Implementing the playing field . . . 177
15.5 Implementing the players . . . 181
15.6 Putting it all together . . . 182
15.7 Displaying results . . . 183
15.8 Next steps . . . 185

16 Tutorial: A balanced excitatory-inhibitory network with delta synapses **187**

17 Example: Tsodyks, Uziel, Markram (2000) **193**

18 Example: Reward-modulated STDP in a virtual environment - Brzosko, Zannone et al. (2017) **201**
18.1 Architecture of the model and experimental rig . . . 201
18.2 Implementing the model . . . 202

D Building on top of the EDEN platform 219

19 Software Maintenance **223**
19.1 Testing process . . . 223
19.2 Build process . . . 223
19.2.1 Build environment . . . 224
19.2.2 Release binaries . . . 224

References 225

Examples gallery **227**

Frequently asked questions **229**
Terminology dictionary . . . 230
Using the simulator . . . 231
NeuroML resources . . . 232
Setting up and running models . . . 233
Writing LEMS equations . . . 234
Troubleshooting . . . 234
What is Eden capable of? . . . 235
What does Eden *not* do, and what are the workarounds? . . . 235

Python API **237**
`eden-simulator` . . . 237
Display API . . . 238
`eden_simulator.display.animation` . . . 238
`eden_simulator.display.spatial` . . . 239
`eden_simulator.display.spatial.k3d` . . . 239
`eden_simulator.display.spatial.pyvista` . . . 241
Experimental API . . . 241

Python Module Index **245**

Index **247**

INSTALLING

EDEN is most easily installed via `pip`:

```
pip install eden-simulator
```

It can then be run from:

- Python, as `import eden_simulator as eden; eden.runEden('LEMS_<sim file>.xml');`
- the command line, as `eden nml LEMS_<sim file>.xml`.

For more installation options, refer to the README here[3].

[3] https://gitlab.com/c7859/neurocomputing-lab/Inferior_OliveEMC/eden/#installing

If you are interested in EDEN, we'll be delighted to hear from you:

General discussion

Get in touch with our community on our chatroom and mailing list[4] - or just email the developers (page 3).

Support tickets

If you'd like to ask for a new feature, or report a problem, please file a support ticket on GitLab[5].

To subscribe to the mailing list, send an email to eden-simulator-general+subscribe@googlegroups.com.

Email us

And more generally, feel free to send e-mail to the developers anytime:

- Primary developer: Sotirios Panagiotou, s.panagiotou@erasmusmc.nl
- Principal investigator: Dr.Christos Strydis, c.strydis@erasmusmc.nl

> **Tip**
>
> For questions and suggestions regarding NeuroML, contact also the wider NeuroML community[6].

Join us

Do you want to add something to EDEN, like a feature, a modified engine or an improved user guide? Join us!

(See also: We have open student thesis topics (page 4) on sim tech research!)

Guide contributions

Do you have a nice usage example, or method of experimenting, with EDEN? Or perhaps you'd like to see some part of this guide explained in more detail? Contact us (page 3) for suggestions.

> **Hint**
>
> Refer to Examples gallery (page 227) to see if something is missing.

[4] https://groups.google.com/g/eden-simulator-general
[5] https://gitlab.com/c7859/neurocomputing-lab/Inferior_OliveEMC/eden/-/issues
[6] https://docs.neuroml.org/NeuroMLOrg/CommunicationChannels.html

Code contributions

Would you like to add a new simulation feature, customize the existing codebase to your needs, or contribute some useful API extensions or language bindings?

See the Hacker's guide (TBA), contact us (page 3) for support, and make a pull request[7].

Research topics

Neural simulation is a topic of active scientific research. More than a product, EDEN is also designed as a platform to test new ideas on the technology of numerical simulation. This ideas can range from new simulation algorithms, to adaptive performance tuning, to specialized computational hardware.

We do have several cutting-edge research questions on simulation technology; if a prospective student (or intern) has some relevant knowledge, and support from an academic institution or an intense personal interest, we can provide specific research topics to work on, and additional guidance.

Here are a few challenges, to explain our kind of work to prospective students:

- Get the sim code for a neural mass model from GitHub[8], and convert it so that the matrix is laid out in the ELLPACK[9] format, for more than 1000 neurons and connection probabilities of 1%, 10%, 90%. Compare simulation speed vs. the original code, and and try to explain the results.
- Get a neural network model in the ModelDB[10] with more than, say, a thousand neurons. Extract the set of synaptic connections between neurons. Apply clustering algorithms to the neurons of the network, so that for N clusters the number of x * (max neurons per cluster) + y * (sum connections between clusters) is minimized. Explore for varying x, y, N.

If you have pulled off one of these, or have another research project to propose, contact us (page 3); see also our Msc thesis vacancies[11] (theme "BrainFrame").

[7] https://gitlab.com/c7859/neurocomputing-lab/Inferior_OliveEMC/eden/-/merge_requests
[8] https://github.com/AmirrezaMov/tvb-algo-c/blob/master/tvb-algo.cpp
[9] https://faculty.cc.gatech.edu/~echow/ipcc/hpc-course/sparsemat.pdf
[10] https://modeldb.science/search?modeltype=Connectionist+Network%3B+Realistic+Network%3B+
[11] https://neurocomputinglab.com/jobs/?job__type_spec=student-thesis

Part A

Introduction to NeuroML

NeuroML is (as of this writing) the most extensive data format to describe models of spiking neural networks (SNN's) (and generally dynamic neural networks), outside the context of a specific simulation system. NeuroML version 2 (i.e. the latest one) relies on the LEMS language for defining new (ODE and event-based) dynamics for the various parts that constitute a neural network. EDEN runs models that are described in this NeuroML-v.2 (plus LEMS) data format.

This part introduces the basic concepts of NeuroML (in its second version) and LEMS, and how these can be applied to construct full-featured models of dynamic neural networks.

CHAPTER

ONE

QUICKSTART

This is a rapid introduction on how to immediately install and use EDEN, fetch a simple model, and get results. More details about each step can be found in the user's guide (page 7).

1.1 Get EDEN

The easiest way to install EDEN is as a Python package[12], which contains EDEN along with helpful routines to use it with Python.

Note that EDEN can also be used fine with other programming languages such as MATLAB or Octave, R and even LabVIEW (examples are coming soon); in fact any language can be used, since EDEN is a as a stand-alone program communicating through files and data streams.

```
%pip install eden_simulator
import eden_simulator
```

```
Requirement already satisfied: eden_simulator in /home/docs/checkouts/readthedocs.
↪org/user_builds/eden-simulator/envs/latest/lib/python3.10/site-packages (0.2.4a4)
Requirement already satisfied: numpy in /home/docs/checkouts/readthedocs.org/user_
↪builds/eden-simulator/envs/latest/lib/python3.10/site-packages (from eden_
↪simulator) (2.2.6)
Requirement already satisfied: lxml in /home/docs/checkouts/readthedocs.org/user_
↪builds/eden-simulator/envs/latest/lib/python3.10/site-packages (from eden_
↪simulator) (6.1.1)
Note: you may need to restart the kernel to use updated packages.
```

1.2 Get a NeuroML model

EDEN runs neural models described in the NeuroML 2[13] format. Users can acquire such models from many online sources (page 232), customise them and even build their own (page 227) for their experiments.

To begin, let's simulate a single neuron modelled classic Hodgkin-Huxley equations. We can get this from the OpenSourceBrain repository[14].

```
# Download the zip file with the model
import urllib.request
urllib.request.urlretrieve('https://codeload.github.com/OpenSourceBrain/hh-testing/
↪zip/refs/heads/master', 'hh.zip')
# and unpack it
```

(continues on next page)

[12] https://pypi.org/project/eden-simulator/
[13] https://neuroml.org
[14] https://github.com/OpenSourceBrain/hh-testing

(continued from previous page)

```
from zipfile import ZipFile
with ZipFile('hh.zip', 'r') as zipp: zipp.extractall()
# Check which files are present now
from os import listdir; # listdir() <-- unhide me !
```

If all went right, there should be a new folder `hh-testing-master` next to this notebook.
Inside it there should be a `NeuroML2` folder containing the description of our model. The reevant files are of types `.nml` and `.xml`:

```
nml_folder = 'hh-testing-master/NeuroML2/'
listdir(nml_folder)
```

```
['LEMS_hh_nostim.xml',
 'hh.lems.nostim.omt',
 'hh_nostim.nml',
 'hh.nml',
 'LEMS_hh.xml',
 'hh.lems.omt']
```

1.3 Run the NeuroML model with EDEN

There is the following distinction between files:

- `.nml` files describe the modelled network, neurons and their parts;
- `LEMS_<something>.xml` files describe meta-parameters about the simulation, such as how long to run it for, and which curves (or rasters) we want to keep from this simulation.

Let's now run the `LEMS_hh.xml` file that specifies a simulation:

```
sim_filename = nml_folder+"LEMS_hh.xml"
sim_data = eden_simulator.runEden(sim_filename)
```

1.4 Show the results

From the last code cell that ran the simulation, we got a variable `sim_data`. What is it, what does it contain?

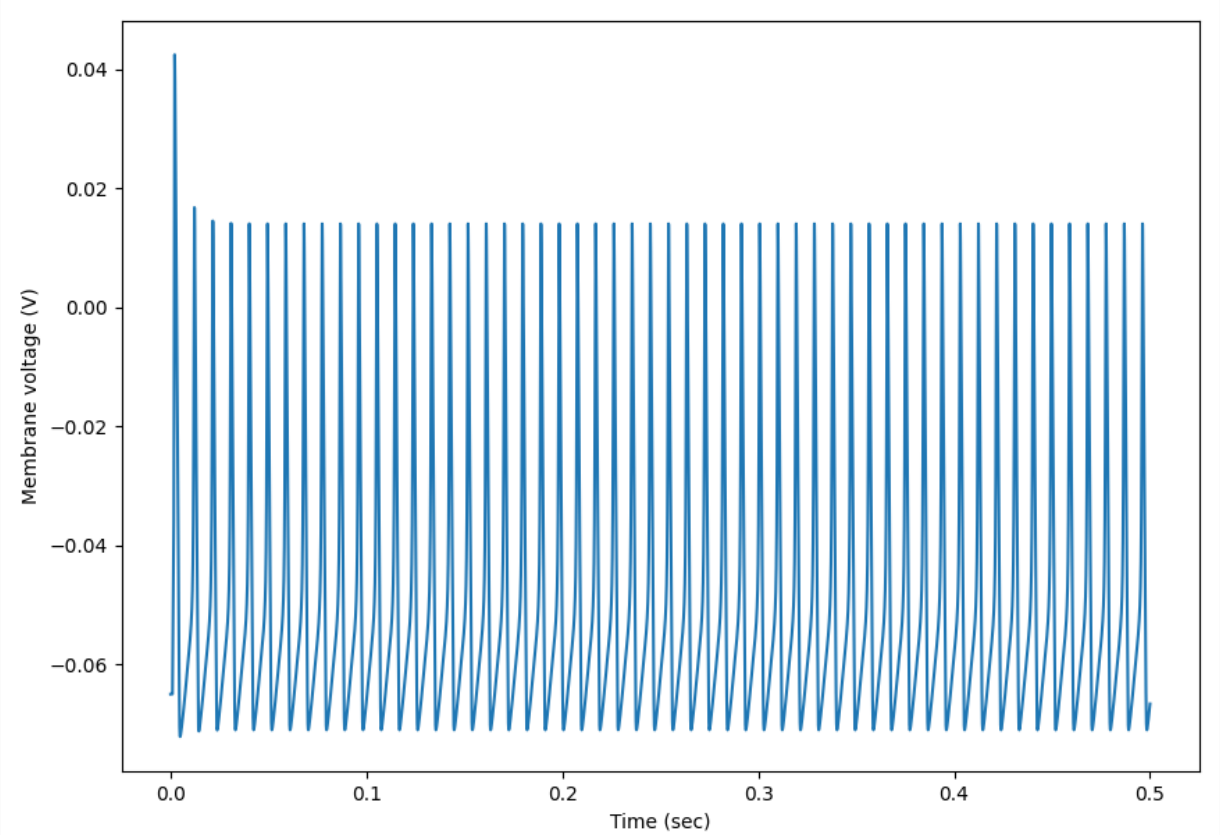
```
print(sim_data)
```

```
{'pop/0/hh/v': array([-0.065     , -0.06499997, -0.06499994, …, -0.06663233,
       -0.06662899, -0.06662564], shape=(500001,)), 't': array([0.00000e+00, 1.
↪00000e-06, 2.00000e-06, …, 4.99998e-01,
       4.99999e-01, 5.00000e-01], shape=(500001,))}
```

It looks like it returned a `dict` with two entries, `pop/0/hh/v` and `t`. The latter represents time throughout the simulation, and the former seems to be a voltage of a HH cell – most likely the one we just simulated.

So then, we can plot membrane voltage versus time and get the classic waveform of the HH model:

```
from matplotlib import pyplot as plt
plt.figure(figsize=[10,7]); plt.plot(sim_data['t'], sim_data['pop/0/hh/v']);
plt.ylabel("Membrane voltage (V)"); plt.xlabel("Time (sec)"); plt.show()
```



Note that voltage is produced in `volts` and time in `seconds`. Unless specified otherwise (page 109), EDEN outputs results in SI units[15] for compatibility with other NeuroML tools (page 233).

1.5 Done!

Following this tutorial, you may continue with a more in-depth introduction to NeuroML (page 13), and see more tutorials (page 127) or the full list of examples (page 227).

1.5.1 Appendix

In the interest of curiosity, we can inspect these NeuroML files that gave us this nice electrical waveform of the classic HH cell. These files are made up of machine-readable text, in the XML[16] format. They are a bit on the verbose side, but there is good reason:

1. The model's description is *explicit*, *complete* and *unambiguous*; all equations, parameters, and interactions that define the model are included in the `.nml` file.
2. Descriptions being *explicit* allows for very diverse range of models to be expressed in NeuroML – from abstract integrate-and-fire neurons, to anatomically detailed Purkinje cells at the electrophysiological level.

The user guide explains how to construct NeuroML networks (page 13) and custom mechanisms (page 61) in further detail. More information about the NeuroML format can be found in NeuroML resources (page 232).

[15] `https://en.wikipedia.org/wiki/SI_derived_unit`
[16] `https://www.tutorialspoint.com/xml/xml_overview.htm`

CHAPTER

TWO

NEUROML BASICS

The program that this user guide is about, EDEN, simulates the activity of spiking neural networks[17]. The models of such networks and their neurons are described in NeuroML[18], which is a neural modelling[19] language in the XML[20] format. EDEN then works by reading NeuroML files and crunching the numbers the model described in these files.

This article will explain how models are written in NeuroML, and how to use EDEN to simulate them. The various parts of a NeuroML simulation are introduced one after the other, in the following sections.

2.1 Creating a simulation

First of all, to set up a NeuroML[21] simulation and "run" it, it takes:

- a NeuroML `network` that contains all the things to simulate;
- a `simulation` that contains the parameters of how to simulate said things (i.e. for how long, what to observe and such).

These are created by writing down their descriptions as text-based, XML files, for various[22] programs[23] to read back.

Let's now see how to write the smallest possible `network` and `simulation`.

In the following, the `%%writefile` line is a Jupyter magic[24] so that when this code cell[25] is run, the specified *file* will be (over)written with the contents as follows in the cell.

> **Tip**
>
> We'll be changing the model files through the tutorial to show different features. When running the notebook, if you want to re-run a previous section, make sure to overwrite the `%%writefile` cells to that point before re-running the simulation.

```
%%writefile EmptyModel.nml
<!-- These properties on the top level <neuroml> tag are just to positively␣
↪identify the file as NeuroML.  They are not mandatory -->
<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_v2.1.xsd">
<!-- Here comes the network -->
```

(continues on next page)

[17] https://en.wikipedia.org/wiki/Spiking_neural_network
[18] https://neuroml.org
[19] https://en.wikipedia.org/wiki/Computational_neuroscience
[20] https://en.wikibooks.org/wiki/XML_-_Managing_Data_Exchange
[21] https://neuroml.org
[22] https://docs.neuroml.org/Userdocs/Software/Software.html
[23] https://docs.neuroml.org/Userdocs/Software/SupportingTools.html
[24] https://ipython.readthedocs.io/en/stable/interactive/magics.html#cellmagic-writefile
[25] https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/Running%20Code.html

(continued from previous page)

```
<!-- set the temperature to that you think your model is running under (depends on␣
↪the species, whether it's cold blooded) -->
<network id="MyNetwork" type="networkWithTemperature" temperature="37degC" >
<!-- Nothing here yet -->
</network>
</neuroml>
```

```
Writing EmptyModel.nml
```

And the simulation file (see also the NeuroML guide[26]) pointing to the `EmptyModel`, by convention there is one file for the `<Simulation>` and one for the `<network>`:

```
%%writefile LEMS_EmptySim.xml
<?xml version="1.0" encoding ="UTF-8" ?>
<Lems>
<!-- we will be <include>ing the nml file with MyNetwork -->
<include file="EmptyModel.nml" />
<!-- the Simulation is the main content, it needs a duration and a timestep.  It␣
↪involves a certain "target" <network> -->
<Simulation id="MySim" length="10 s" step="10 us" target="MyNetwork" >
<!-- Nothing here yet -->
</Simulation>
<!-- Just in case there are multiple <Simulation>s described, specify the one to␣
↪run here -->
<Target component="MySim"/>
</Lems>
```

```
Writing LEMS_EmptySim.xml
```

Now let's run this completely minimal simulation.

EDEN can be used from Python with the `eden_simulator` package[27], just `pip install` it if you haven't already in Quickstart (page 9):

```
from eden_simulator import runEden
%time runEden('LEMS_EmptySim.xml');
```

```
CPU times: user 55.8 ms, sys: 4.81 ms, total: 60.6 ms
Wall time: 661 ms
```

Now let's see how we can make it a tad more interesting.

2.2 Adding a neuron and recording output

The first step to define a neural model is to define a neuron. Let's add a neuron to the model and see how it behaves.

This is done by defining the neuron model as a cell type, and adding to the `<network>` a `<population>` with cells of said cell type (called the `component` for this `<population>`).

Instead of specifying the cell type from scratch, we'll use the common linear integrate-and fire model[28] as a base, and supply our own dynamical parameters for it. This cell type can be found in the standard NeuroML library as <iafCell>[29] and has parameters `C`(apacitance of cell membrane), `leakConductance` (of membrane), `leakReversal` (potential), `reset` (potential right after firing) and `thresh`(old for firing).

We'll set them as follows: `C = 200 pF, leakConductance = 10 nmho, leakReversal = -70 mV, reset = -70 mV, thresh = -50 mV`.

[26] https://docs.neuroml.org/Userdocs/LEMSSimulation.html
[27] https://pypi.org/project/eden-simulator
[28] https://en.wikipedia.org/wiki/Biological_neuron_model#Leaky_integrate-and-fire
[29] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#iafcell

As we'll see in an in-depth tutorial (page 129), the population can also be a `populationList` with specific 3-D positions for each cell, but this is not necessary for simulating the population.

```
%%writefile SingleCell_Model.nml
<!-- These properties on the top level <neuroml> tag are just to positively␣
↪identify the file as NeuroML.  They are not mandatory -->
<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_v2.1.xsd">
<!-- Here comes the cell type ! -->
<iafCell id="MyFirstCellType" C="200 pF" leakConductance="10 nS" leakReversal="-70␣
↪mV" reset="-70mV" thresh="-50mV" />
<!-- Here comes the network -->
<network id="MyNetwork" type="networkWithTemperature" temperature="37degC" >
    <!-- Add a population with a single cell ! -->
    <population id="MyFirstPopulation" component="MyFirstCellType" size="1"/>
</network>
</neuroml>
```

```
Writing SingleCell_Model.nml
```

There are more point neuron types defined in base NeuroML (see for example, adExIaFCell[30] and izhikevich2007Cell[31]), and custom ones can be built as LEMS components in a following section (page 61).

More than point neurons, NeuroML and EDEN also support spatially modelled neurons with a detailed morphology and spatial compartments. Due to the added details and modelling options, these are covered in a following section (page 33).

In order to observe the cell, we'll have to record one of its attributes over time.

This is done in the *simulation* file by adding an `<OutputFile>` with `<OutputColumn>`s to the `<Simulation>`.

- Each `OutputColumn` points to a `quantity`, which in the case of simplified neurons is described as `population name[cell number in population starting from 0]/variable`. (Refer to the path documentation (page 87) for more possibilities.)

We see (click on "Dynamics" here[32]) that `v` stands for the membrane voltage, let's record that.
Hence we want to record the quantity `MyFirstPopulation[0]/v`.

Let's write the new simulation file:

```
%%writefile LEMS_SingleCell_Sim.xml
<?xml version="1.0" encoding ="UTF-8" ?>
<Lems>
<!-- we will be <include>ing the nml file with the SingleCellModel -->
<include file="SingleCell_Model.nml" />
<!-- the Simulation is the main content, it needs a duration and a timestep.  It␣
↪involves a certain "target" <network> -->
<Simulation id="MySim" length="1 s" step="10 us" target="MyNetwork" >
    <!-- Add an OutputFile ! -->
    <OutputFile id="MyFirstOutputFile" fileName="results.gen.txt">
        <OutputColumn id="my_first_vm" quantity="MyFirstPopulation[0]/v"/>
    </OutputFile>
</Simulation>
<!-- Just in case there are multiple <Simulation>s described, specify the one to␣
↪run here -->
```

(continues on next page)

[30] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#adexiafcell
[31] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#izhikevich2007cell
[32] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#iafcell

(continued from previous page)

```
<Target component="MySim"/>
</Lems>
```

```
Writing LEMS_SingleCell_Sim.xml
```

And run the new simulation:

```
results = runEden('LEMS_SingleCell_Sim.xml')
results
```

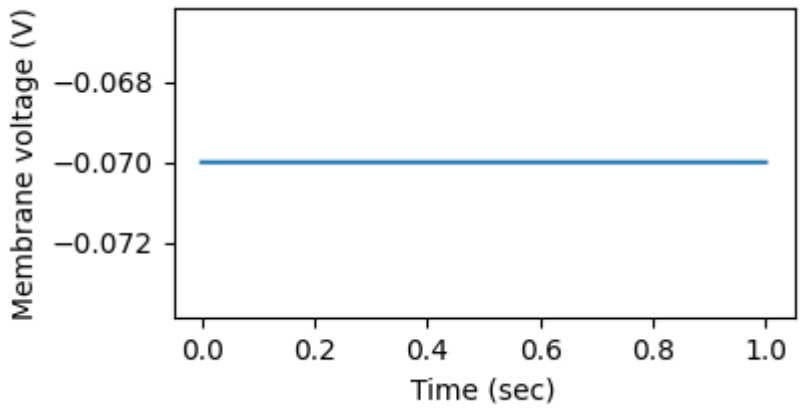
```
{'MyFirstPopulation[0]/v': array([-0.07, -0.07, -0.07, …, -0.07, -0.07, -0.07],␣
↪shape=(100001,)),
 't': array([0.0000e+00, 1.0000e-05, 2.0000e-05, …, 9.9998e-01, 9.9999e-01,
        1.0000e+00], shape=(100001,))}
```

The output of Eden is in the form of a `dict`. Its entries are `t` and the specified `quantity` of each `OutputColumn`, same as when using the general tool PyNeuroML[33].

The value of `results['t']` stands for the list of recorded points in time for the time series, and every other entry value stands for the trajectory of the respective variable over the sampling points in time.
All values are are in SI units[34], that is `seconds` for `t` `volts` for `v`.

Let's plot `MyFirstPopulation[0]/v` against `t` then:

```
from matplotlib import pyplot as plt; plt.figure(figsize=(4,2))
plt.plot(results['t'],results['MyFirstPopulation[0]/v'])
plt.xlabel('Time (sec)'); plt.ylabel('Membrane voltage (V)');
```



Apparently this type of neuron doesn't fire intrinsically; it just stays polarised and quiet, producing this electrical flat line.

Other types of neurons fire or oscillate intrinsically, but we'll need to apply some physiological *stimulus* on this one to see its response.

[33] https://github.com/NeuroML/pyNeuroML/
[34] https://en.wikipedia.org/wiki/SI_derived_unit

2.3 Adding stimuli

We typically study neurons in isolation, or at least as part of a small piece of tissue. But neurons normally react to electro-chemical stimuli supplied by other neurons, thus we'll have to provide an artificial "equivalent" to study their behaviour. (We can also study the natural response of neurons to their natural stimuli in vivo, but this is much harder to understand and reason about.)

A basic way to play with cells is the current clamp[35], and in this virtual world we don't even have to worry about injuring the cells. By injecting "positive" current, we can depolarise a cell and cause action potentials.

This is done by defining the input's model as an *input source* type, and adding to the `<network>` an `<inputList>` with probes of said source type (called the `component` for this `<inputList>`).

Instead of specifying the input type from scratch (as we'll be doing in a [following section]), we'll use the common *DC pulse* current clamp (flat current injected over a time window) as a base, and supply our own dynamical parameters for it. This input type can be found in the standard NeuroML library as <pulseGenerator>[36] and has parameters `delay` (before the pulse starts), `duration` (of the pulse), and `amplitude`.
We'll set them as follows: `delay = 100 msec, duration = 500 msec, amplitude = 210 pA`.

Next, we'll add an `<inputList>` to the `<network>`. Each `<inputList>` contains one or more copies (called `<input>` instances) of the same input source `component`, and is applied on a certain cell `population`.

Each `<input>` has a cell `target` which can be, for example, written as `<population name>[<cell instance number>]`. (See more about LEMS paths (page 88).) It also specifies a `destination` which is at the moment reserved to be the value `synapses` (as in, the input injects current to the cell just like synapses (page 19) do). Finally, the alternative name `inputW` can also be used to apply a `weight` property on the amplitude of the specific `<input>` instance. (See how it's done below (page 21)).

As we'll see in the following chapter on spatially detailed cells (page 33), the specific `segment` and `fractionAlong` (the `segment`) to apply the `<input>` on can also be optionally specified, otherwise the `input` will be placed on the "soma" of the cell.

Let's see then, how to instantiate a current clamp and apply it on the cell in the following.

```
%%writefile SingleCell_Model.nml
<!-- These properties on the top level <neuroml> tag are just to positively␣
↪identify the file as NeuroML.  They are not mandatory -->
<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_v2.1.xsd">
<!-- Here comes the cell type -->
<iafCell id="MyFirstCellType" C="200 pF" leakConductance="10 nS" leakReversal="-70␣
↪mV" reset="-70mV" thresh="-50mV" />
<!-- Here comes the input type ! -->
<pulseGenerator id="MyFirstInputSource" delay="100ms" duration="500ms" amplitude=
↪"0.21nA"/>
<!-- Here comes the network -->
<network id="MyNetwork" type="networkWithTemperature" temperature="37degC" >
    <!-- Add a population with a single cell -->
    <population id="MyFirstPopulation" component="MyFirstCellType" size="1"/>
    <!-- Apply an inputList on the population ! -->
    <inputList id="MyFirstInputList" population="MyFirstPopulation" component=
↪"MyFirstInputSource">
        <!-- With one stim placement on it -->
        <input id="0" target="MyFirstPopulation[0]" destination="synapses"/>
    </inputList>
```

(continues on next page)

[35] https://en.wikipedia.org/wiki/Electrophysiology#Current_clamp
[36] https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#pulsegenerator

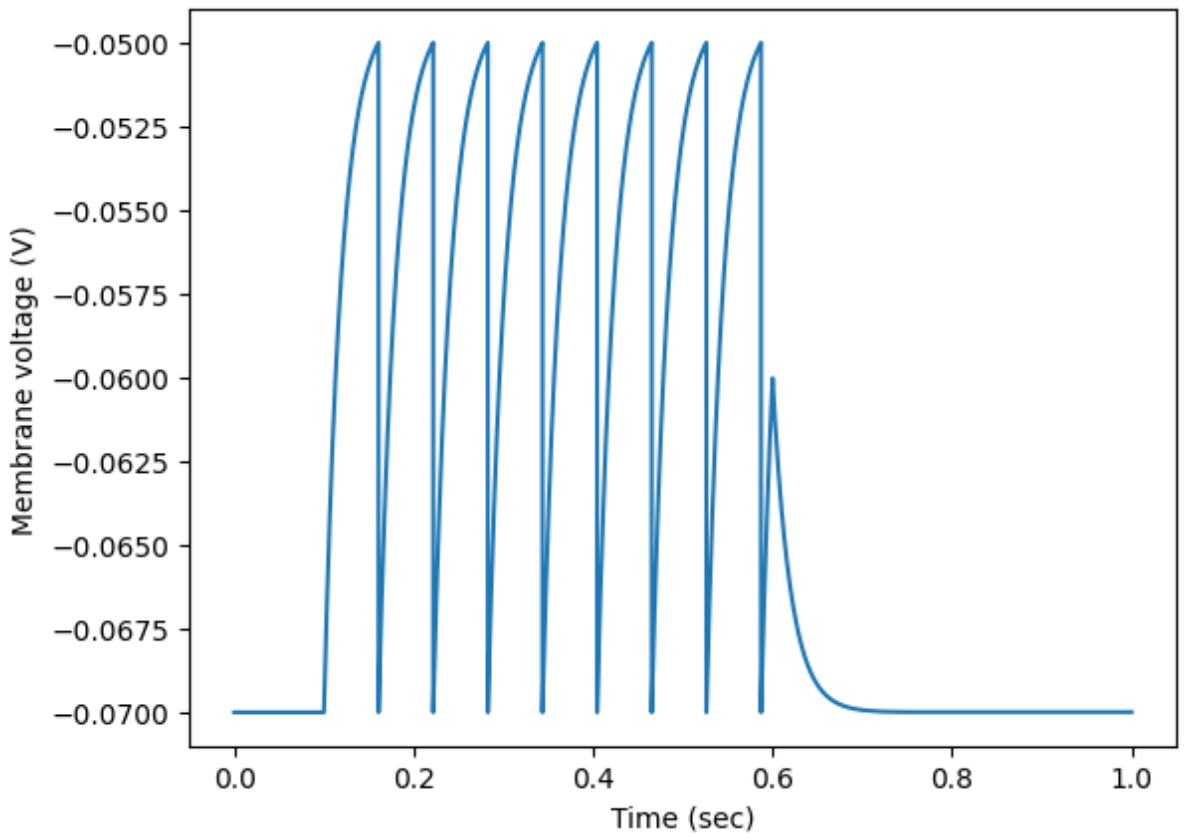
(continued from previous page)

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

```
Overwriting SingleCell_Model.nml
```

And run the simulation with the updated model file:

```
results = runEden('LEMS_SingleCell_Sim.xml')
plt.plot(results['t'],results['MyFirstPopulation[0]/v'])
plt.xlabel('Time (sec)'); plt.ylabel('Membrane voltage (V)');
```



That's more like it.

Change the `amplitude` of the input source (or equivalently weight (page 21) with `inputW`), repeat with the new model files and see how the firing rate changes.

There are more input types defined in base NeuroML (see for example, ramp[37] and sine[38] current sources and voltage clamps[39]), and custom ones can be built as LEMS components in a following section (page 61).

[37] `https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#rampgenerator`
[38] `https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#sinegenerator`
[39] `https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#voltageclamp`

2.4 Adding more neurons and synapses

This section shows how to make models with *more than one neuron*. This way, one can explore in detail the mechanism of interaction between a pair of neurons, or set up whole *populations* of neurons in an inter-connected *network* and explore the group dynamics.

First of all, let's extend the first population from out previous model to contain *two* neurons this time. This is quite easy to do, we can just set the `size` of the `population` to what we want.

Then, we'll connect these two cells with a *synapse*.

This is done by defining the synapse's model as an *synapse type*, and adding to the `<network>` a synaptic `<projection>` with synapses of said type (called the `component` for this `<projection>`).

Instead of specifying the synapse type from scratch (as we'll be doing in a following section (page 61)), we'll use the common exponential-decay conductance[40] model as a base, and supply our own dynamical parameters for it. This synapse type can be found in the standard NeuroML library as <expOneSynapse>[41] and has parameters `erev`(ersal potential of the active synapse), `gbase`(peak post-synaptic conductance per spike), and `decay` (time constant). We'll set them as follows: `erev = 0 mV, gbase = 10 nmho, decay = 5 ms` for educational purposes.

Next, we'll add a `<projection>` to the `<network>`. Each `<projection>` contains one or more copies (called `<connection>`s) of the same `synapse` type, and is applied from a certain cell `presynapticPopulation` to a certain `postsynapticPopulation`.

Each `<connection>` has a distinct serial `id` and its `pre` and `post` site. Each site is located using the `CellId` (of the `pre or post` population). The alternative names `connectionW` and `connectionWD` can also be used to apply a `weight` property on the amplitude of the specific `<connection>` for the former, and `weight` as well as propagation `delay` for the latter.

As we'll see in the following section on spatially detailed cells (page 33), the specific `SegmentId` and `Fraction-Along` (the `segment`) to apply the `<synapse>` between can also be optionally specified for the `pre` and `post` sites, otherwise the sites will be located on the "soma" of each corresponding cell.

Fun fact: One can even model a self-synapse that way.

Let's see then, how to instantiate a synapse and connect our *two cells* with it in the following. We'll use a huge delay of `10 msec` to make its effect visible.

```
%%writefile MultiCell_Model.nml
<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_v2.1.xsd">
<iafCell id="MyFirstCellType" C="200 pF" leakConductance="10 nS" leakReversal="-70␣
↪mV" reset="-70mV" thresh="-50mV" />
<pulseGenerator id="MyFirstInputSource" delay="100ms" duration="500ms" amplitude=
↪"0.21nA"/>

<expOneSynapse id="MyFirstSynapseType" gbase="10nS" erev="0V" tauDecay="5ms"/>
<network id="MyNetwork" type="networkWithTemperature" temperature="37degC" >

    <!-- Add a population with TWO cells ! -->
    <population id="MyFirstPopulation" component="MyFirstCellType" size="2"/>

    <!-- Apply an inputList on the population -->
    <inputList id="MyFirstInputList" population="MyFirstPopulation" component=
↪"MyFirstInputSource">
```

(continues on next page)

[40] https://neuronaldynamics.epfl.ch/online/Ch3.S1.html#Ch3.E2
[41] https://docs.neuroml.org/Userdocs/Schemas/Synapses.html#exponesynapse

(continued from previous page)

```
        <!-- With one stim placement on it -->
        <input id="0" target="MyFirstPopulation[0]" destination="synapses"/>
    </inputList>

    <!-- Apply a synaptic projection ! -->
    <projection id="MyFirstApProjection" presynapticPopulation="MyFirstPopulation"␣
↪postsynapticPopulation="MyFirstPopulation" synapse="MyFirstSynapseType">
        <!-- With one connection on it -->
        <connectionWD id="0" preCellId="0" postCellId="1" weight="1" delay="10 ms"/
↪>
    </projection>

</network>
</neuroml>
```

```
Writing MultiCell_Model.nml
```

There are more synapse types defined in base NeuroML, (see for example, alphaCurrentSynapse[42] and expThreeSynapse[43]), and custom ones can be built as LEMS components in a following section (page 61).

In order to observe the *two cells*, we'll have to record the new cell as well. This is done by adding one more `<OutputColumn>`s to the `<Simulation>`, as follows:

```
%%writefile LEMS_MultiCell_Sim.xml
<?xml version="1.0" encoding ="UTF-8" ?>
<Lems>
<include file="MultiCell_Model.nml" />
<Simulation id="MySim" length="1 s" step="100 us" target="MyNetwork" >
    <OutputFile id="MyFirstOutputFile" fileName="results.gen.txt">
        <!-- Add TWO outputColumns, one for each cell ! -->
        <OutputColumn id="my_first_vm"  quantity="MyFirstPopulation[0]/v"/>
        <OutputColumn id="my_second_vm" quantity="MyFirstPopulation[1]/v"/>
    </OutputFile>
</Simulation>
<Target component="MySim"/>
</Lems>
```

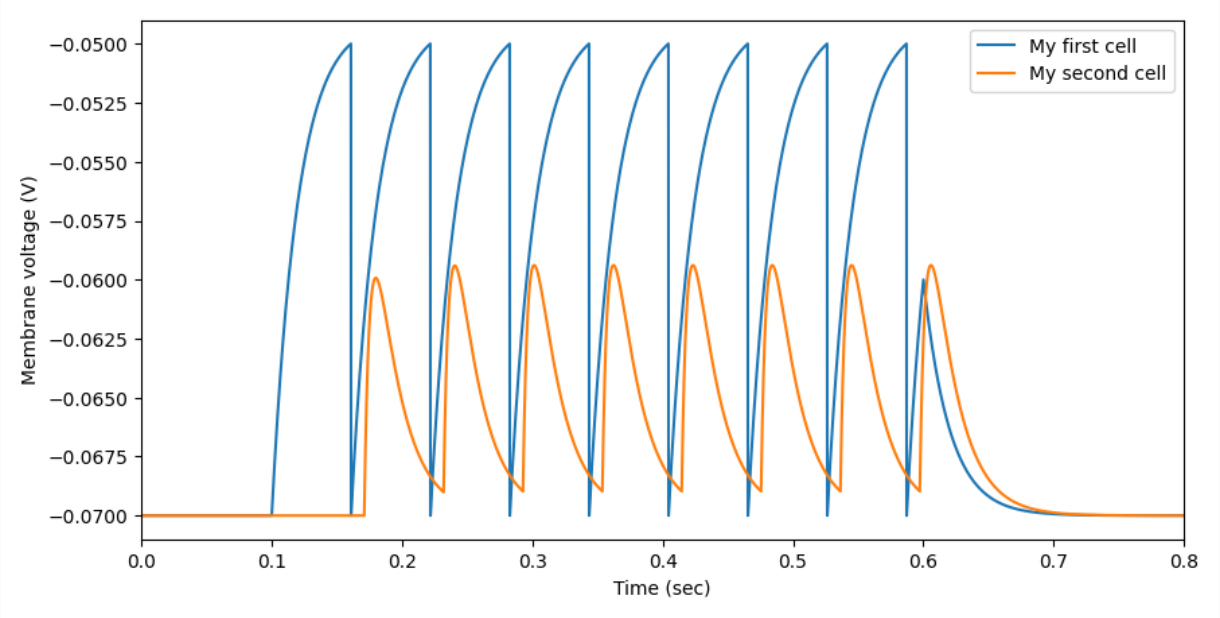
```
Writing LEMS_MultiCell_Sim.xml
```

Let's run this simulation:

```
results = runEden('LEMS_MultiCell_Sim.xml')
plt.figure(figsize=(10,5))
plt.plot(results['t'],results['MyFirstPopulation[0]/v'], label='My first cell')
plt.plot(results['t'],results['MyFirstPopulation[1]/v'], label='My second cell')
plt.xlabel('Time (sec)'); plt.ylabel('Membrane voltage (V)'); plt.legend(); plt.
↪xlim((0, .8))
```

```
(0.0, 0.8)
```

[42] https://docs.neuroml.org/Userdocs/Schemas/Synapses.html#alphacurrentsynapse
[43] https://docs.neuroml.org/Userdocs/Schemas/Synapses.html#expthreesynapse



2.5 Adding more cell types and spike sources

There are many SNN models that are formed by one population of similar cells, but there also exist *heterogeneous* networks comprising *multiple populations* of different cells, connected by various *synaptic projections*. Moreover, some of these populations may receive action potentials from extra-model *spike sources*.

In this section, we'll introduce another point-neuron type and its corresponding population, see how to access different cell populations, and add some external spike input to our network as well.

2.5.1 Adding a new cell type and population

Adding another point-neuron type and population is straightforward: We'll introduce the new *neuron type* with a tag outside the `<network>` definition, and add a second population inside the `<network>`, just like we did with our first cell type and population.

This time, we'll use the common and very expressive Izhikevich cell model[44] (a simple non-linear system that captures many waveform types) as a base, and supply our own dynamical parameters for it.

This input type can be found in the standard NeuroML library as <izhikevich2007Cell>[45] (an update of the original <izhikevichCell>[46] model that used mostly dimensionless variables) and has parameters `C`(apacitance), `a` (rate of adaptation), `b` (coupling of adaptation to voltage), `c` (after-spike reset voltage), `d` (after-spike adaptation increment), `k`(onstant of voltage growth rate), `v0` (initial membrane voltage), `vpeak` (max voltage before reset), `vr` (resting membrane potential), and `vt` (threshold potential).

After consulting the Izhikevich model gallery repo[47] on OpenSourceBrain that shows different activity types with their corresponding parameters, we select the "Regular Spiking" activity type and modify it a bit to show interaction between neurons. We'll set them as follows:

```
C = 100 pF, a = 0.03 Hz^-1, b = -2 nS, c = -50 mV, d = 100 pA, k = 0.7 µS/V,
vr = -60 mV, vpeak = 35 mV, vt = -40 mV, v0 = vr
```

[44] https://www.izhikevich.org/publications/spikes.pdf
[45] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#izhikevich2007cell
[46] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#izhikevichcell
[47] https://github.com/OpenSourceBrain/IzhikevichModel/blob/master/NeuroML2/Izh2007Cells.net.nml

```
%%writefile MultiCell_Model.nml
<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_v2.1.xsd">
<iafCell id="MyFirstCellType" C="200 pF" leakConductance="10 nS" leakReversal="-70␣
↪mV" reset="-70mV" thresh="-50mV" />

<!-- Add a new a new cell type ! -->
<izhikevich2007Cell id="MySecondCellType" 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">
    <notes>Regular spiking cell</notes>
</izhikevich2007Cell>

<pulseGenerator id="MyFirstInputSource" delay="100ms" duration="500ms" amplitude=
↪"0.21nA"/>
<expOneSynapse id="MyFirstSynapseType" gbase="10nS" erev="0V" tauDecay="5ms"/>

<network id="MyNetwork" type="networkWithTemperature" temperature="37degC" >
    <population id="MyFirstPopulation"  component="MyFirstCellType"  size="2"/>

    <!-- Add another population with two cells ! -->
    <population id="MySecondPopulation" component="MySecondCellType" size="2"/>

    <!-- Apply an inputList on the first population -->
    <inputList id="MyFirstInputList" population="MyFirstPopulation" component=
↪"MyFirstInputSource">
        <!-- With one stim placement on it -->
        <input id="0" target="MyFirstPopulation[0]" destination="synapses"/>
    </inputList>

     <!-- Apply an inputList on the SECOND population ! -->
    <inputList id="MySecondInputList" population="MySecondPopulation" component=
↪"MyFirstInputSource">
        <!-- With two stim placements on it, one for each cell -->
        <inputW id="0" target="0" destination="synapses" weight="0.4"/>
        <inputW id="1" target="1" destination="synapses" weight="0.3"/>
    </inputList>

    <!-- Apply the AP synaptic projection -->
    <projection id="MyFirstApProjection" presynapticPopulation="MyFirstPopulation"␣
↪postsynapticPopulation="MyFirstPopulation" synapse="MyFirstSynapseType">
        <!-- With one connection on it -->
        <connectionWD id="0" preCellId="0" postCellId="1" weight="1" delay="10 ms"/
↪>
    </projection>

</network>
</neuroml>
```

```
Overwriting MultiCell_Model.nml
```

And let's add more one more <OutputColumn>s to the <Simulation> to keep track of the new cells.

Setting the randomisation seed

Also, because we'll be using simulated random input in the following, we can set another parameter of the `<Simulation>` to "fix[48]" the random numbers that will be pulled out of the hat and hence get the same result for a given model, every time.

- This is the optional `seed` that should be an integer number. If `seed` is not specified, then EDEN will pick the time and date as the seed number, making all the random numbers change every time.
- **Important**: Due to multiple technological reasons, different computing setups come with different *numerical perturbations*. Being (usually) chaotic systems, neural networks may show different activity from the same initial state, because of these perturbations; though a good model's qualitative *function* should not be affected by this. Read more about the limitations of numerical reproducibility, in a frequently answered question (page 231).

> **Caution**
>
> Please do not rely on the `seed` to make your model work, random variables should be random. A working model should be working for at least most of the `seeds`, always check before publishing.

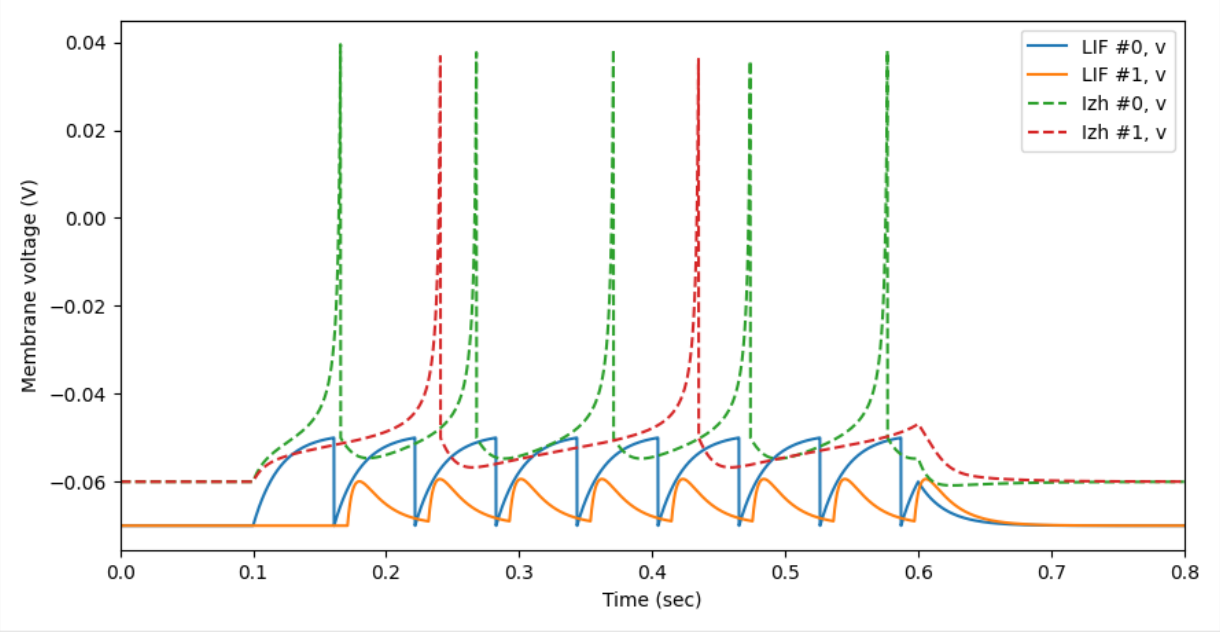
```
%%writefile LEMS_MultiCell_Sim.xml
<?xml version="1.0" encoding ="UTF-8" ?>
<Lems>
<include file="MultiCell_Model.nml" />
<!-- Set the SEED to freeze the random numbers ! -->
<Simulation id="MySim" length="1 s" step="100 us" seed="20190109" target="MyNetwork
↪" >
    <OutputFile id="MyFirstOutputFile" fileName="results.gen.txt">
        <OutputColumn id="my_first_vm"  quantity= "MyFirstPopulation[0]/v"/>
        <OutputColumn id="my_second_vm" quantity= "MyFirstPopulation[1]/v"/>
        <!-- Add TWO more outputColumns, one for each new cell ! -->
        <OutputColumn id="my_third_vm"  quantity="MySecondPopulation[0]/v"/>
        <OutputColumn id="my_fourth_vm" quantity="MySecondPopulation[1]/v"/>
    </OutputFile>
</Simulation>
<Target component="MySim"/>
</Lems>
```

```
Overwriting LEMS_MultiCell_Sim.xml
```

And see our new cells behave. They exhibit an actual AP shape, whereas the LIF cells reset right on the threshold-to-spike instead of the spike's peak.

```
def show_multicell_results(results):
    from matplotlib import pyplot as plt;
    plt.figure(figsize=(10,5))
    plt.plot(results['t'],results[ 'MyFirstPopulation[0]/v'], label='LIF #0, v')
    plt.plot(results['t'],results[ 'MyFirstPopulation[1]/v'], label='LIF #1, v')
    plt.plot(results['t'],results['MySecondPopulation[0]/v'], label='Izh #0, v',␣
↪linestyle='--')
    plt.plot(results['t'],results['MySecondPopulation[1]/v'], label='Izh #1, v',␣
↪linestyle='--')
    plt.xlabel('Time (sec)'); plt.ylabel('Membrane voltage (V)'); plt.legend();␣
↪plt.xlim((0, .8))

results = runEden('LEMS_MultiCell_Sim.xml')
show_multicell_results(results)
```

[48] https://en.wikipedia.org/wiki/Random_seed



2.5.2 Adding abstract spike sources and applying them to the network

As we noted previously, our virtual neurons may need to be stimulated with realistic input; this way that we can watch and play with them at a "natural-like" state, instead of crushing loneliness on a glass dish. Of course, these stimuli are normally provided by *adjacent neurons*, which motivates us to model those neurons as well as well as the stimuli that these neurons receive… At some point we end up out of scope and/or technical resources, and will have to replace external neuron input with a rougher *abstraction*. One such abstraction is entities that emit "firing events" independently of what the network does, according to a mathematical model. These firing events can then activate synapses on target cells in our neural network.

Let's see how this works in NeuroML, by defining two different *spike source types*, adding `populations` of said types in the `network`, and delivering their potentials through `projections`.

We will add two *spike source types* from the NeuroML library: a deterministic <spikeArray>[49] and a stochastic <SpikeSourcePoisson>[50]:

- For the former, we will specify the exact occurrence time of every single spike by adding `<spike>` *child tags* inside it.
- For the latter, we'll supply parameters as usual in the previous parts, namely: `start = 200 ms, duration = 300 ms, rate = 20 Hz`.

Next, we'll instantiate them with `<population>` tags as if they were cells. (It so happens.)

Finally, we'll add `<projection>`s with `synapse` `connections` *from* those not-cell populations, *to* real cells.

Note: we cannot specify spike sources as `post`-synaptic targets since they are not neurons, and hence we can't just attach biophysical synapses on them. Neither can we connect them with graded synapses (page 27) for that matter.

```
%%writefile MultiCell_Model.nml
<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_v2.1.xsd">
<iafCell id="MyFirstCellType" C="200 pF" leakConductance="10 nS" leakReversal="-70␣
↪mV" reset="-70mV" thresh="-50mV" />
```

(continues on next page)

[49] https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#spikearray
[50] https://docs.neuroml.org/Userdocs/Schemas/PyNN.html#spikesourcepoisson

(continued from previous page)

```
<izhikevich2007Cell id="MySecondCellType" 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">
    <notes>Regular spiking cell</notes>
</izhikevich2007Cell>
<pulseGenerator id="MyFirstInputSource" delay="100ms" duration="500ms" amplitude=
↪"0.21nA"/>

<!-- Add new spike sources ! -->
<spikeArray id="MySpikeList">
    <spike id="0" time="210 ms"/>
    <spike id="1" time="350 ms"/>
    <spike id="2" time="450 ms"/>
</spikeArray>
<SpikeSourcePoisson id="MyRandomSpikeSource" start="200ms" duration="300ms" rate=
↪"15Hz"/>

<expOneSynapse id="MyFirstSynapseType" gbase="10nS" erev="0V" tauDecay="5ms"/>

<network id="MyNetwork" type="networkWithTemperature" temperature="37degC" >
    <population id="MyFirstPopulation"  component="MyFirstCellType"  size="2"/>
    <population id="MySecondPopulation" component="MySecondCellType" size="2"/>

    <!-- Add "populations" of SPIKE SOURCES beside the cells ! -->
    <population id="Spikes_Counted" component="MySpikeList"         size="1" />
    <population id="Spikes_Random"  component="MyRandomSpikeSource" size="1" />

    <!-- Apply an inputList on the first population -->
    <inputList id="MyFirstInputList" population="MyFirstPopulation" component=
↪"MyFirstInputSource">
        <!-- With one stim placement on it -->
        <input id="0" target="MyFirstPopulation[0]" destination="synapses"/>
    </inputList>
     <!-- Apply an inputList on the second population -->
    <inputList id="MySecondInputList" population="MySecondPopulation" component=
↪"MyFirstInputSource">
        <!-- With two stim placements on it, one for each cell -->
        <inputW id="0" target="0" destination="synapses" weight="0.4"/>
        <inputW id="1" target="1" destination="synapses" weight="0.3"/>
    </inputList>

    <!-- Apply the AP synaptic projection -->
    <projection id="MyFirstApProjection" presynapticPopulation="MyFirstPopulation"␣
↪postsynapticPopulation="MyFirstPopulation" synapse="MyFirstSynapseType">
        <!-- With one connection on it -->
        <connectionWD id="0" preCellId="0" postCellId="1" weight="1" delay="10 ms"/
↪>
    </projection>

    <!-- And, finally, project synapses from spike sources to cells ! -->
    <projection id="MyFirstSourceProjection" presynapticPopulation="Spikes_Counted
↪" postsynapticPopulation="MySecondPopulation" synapse="MyFirstSynapseType">
        <connection id="0" preCellId="0" postCellId="0"/>
    </projection>
    <projection id="MySecondSourceProjection" presynapticPopulation="Spikes_Random
↪" postsynapticPopulation="MyFirstPopulation" synapse="MyFirstSynapseType">
        <connection id="0" preCellId="0" postCellId="1"/>
    </projection>
</network>
</neuroml>
```

```
Overwriting MultiCell_Model.nml
```

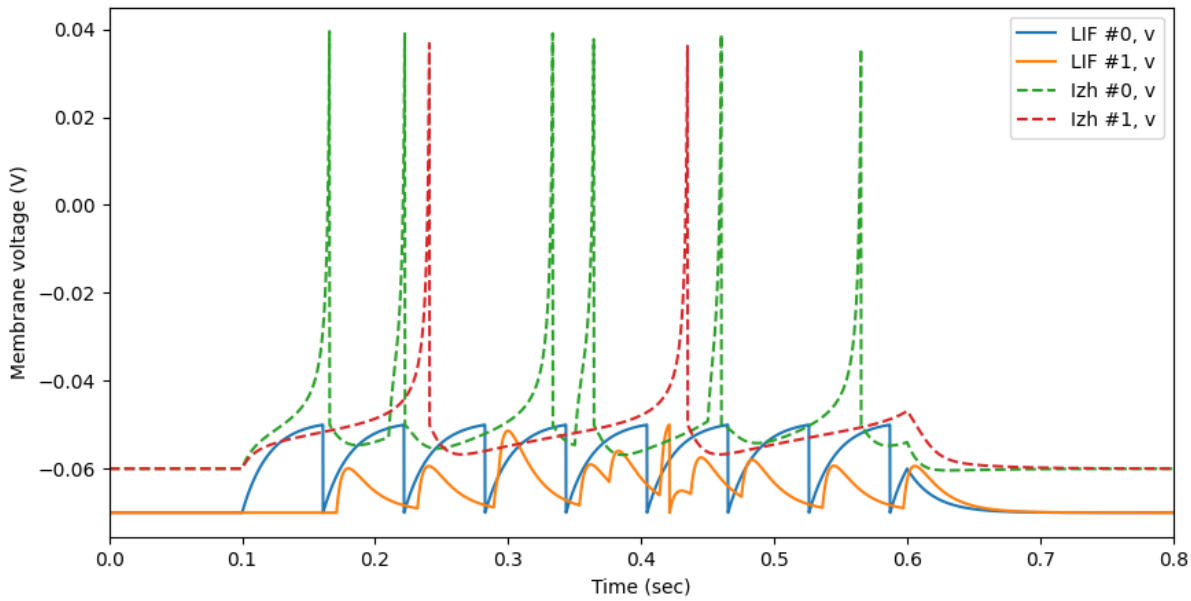
There is this interesting detail about spike sources: although functionally they are *inputs* and are classed as such[51] in the NeuroML specification, model-wise they act like a lesser form of *neurons* and in fact they are instantiated in the `network` as such.

There are more spike sources defined in base NeuroML, (see for example, the regular spikeGenerator[52] and the uniformly-distributed spikeGeneratorRandom[53]). Custom ones can be built as LEMS components in a following section (page 61).

If the desired spike series are too complicated to describe with a base NeuroML tag pr with a LEMS spike source (page 76), they can also be generated *outside* EDEN and streamed *into* the simulation, using the IO extension (page 118). The same applies for other input, in the form of biophysical time series (page 116).

Let's see what happens.

```
results = runEden('LEMS_MultiCell_Sim.xml')
show_multicell_results(results)
```



The spikes are applied to one cell of each type cells, and the exact times of occurrence are seen as very abrupt increases (i.e. depolarisations) in membrane potential.

Observe that the events from the `<spikeArray>` trigger the synapses at the precise times specified in the `<spike>`s, whereas the spikes from the `<SpikeSourcePoisson>` are stochastically distributed.

Remove the `seed` mentioned above (page 23) and re-run the cell, to see how the `<spikeArray>` will remain the same and the `<SpikeSourcePoisson>` will vary.

[51] https://docs.neuroml.org/Userdocs/Schemas/Inputs.html
[52] https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#spikegenerator
[53] https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#spikegeneratorrandom

2.6 Adding graded interactions

In the previous sections, we added a synaptic connection that follows the traditional "action potential" model: if the pre-synaptic neuron passes a threshold, then an invariant quantity of neurotransmitters is released, the neurotransmitters slowly cross the synaptic gap and the post-synaptic potential is applied to the post-neuron. Information travels one way (`pre → post`) only, exclusively at the instants that the pre-neuron fires.
(See also Figure 3.1[54] from the *Neuronal Dynamics* book.)

This is not the only way that neurons are seen to interact in the wild. Transmitter release may happen gradually with membrane potential (it's not all or nothing), and neurons often also interact through *graded* potentials. These are ways that neurons engage in *continuous interaction*, and we can also model that in NeuroML.

In this section, we will add *graded interactions* between cells in the network, and look closely at their effects.

> **Caution**
>
> Use graded synapses only with neurons that have realistic action potential shapes (*not* LIF, for example), lest you end up with wrong conclusions.

We will add two different *graded synaptic component* types from the NeuroML library: an ohmic[55] <gapJunction>[56] model, and a dummy <silentSynapse>[57] model that applies no effect (for one-way interactions).

As usual, we'll define the synapse types inside the `<neuroml>` and outside the `<network>`, but this time we'll different `projection` tags to state that unlike the previous 'classical' synapses, these are *bi-directional continuous connections*. The new tags used are `<electricalProjection>` for symmetric connections and `<continuousProjection>` for not necessarily symmetric connections. These contain different tags, `<electricalConnectionInstance>` and `<continuousConnectionInstance>` respectively.

The difference with `<projection>` and `<connection>` is that the `synapse` is not an attribute of the `projection` but individually specified for each connection instance:

- `<electricalConnectionInstance>`s have a `synapse` type parameter. This `synapse` gets instantiated twice, once for *each* side of the connection, and the two halves are symmetrically attached to the (only nominally) `pre` and `post` synaptic sites.
- `<continuousConnectionInstance>`s have two different `synapse` type parameters, `preComponent` and `postComponent`. In this case, the `pre` component may be different from the `post` component, each is instantiated and attached on its respective side of the synapse.

So then, the tags are slightly different from the classic-type `<connection>` and the `{pre, post}synapticComponents` are specified individually for each synapse, that's because the style changed during the development of NeuroML. EDEN may be rather permissive on mixing and matching from either style. Just remember that you may have to use a slightly differently style for action and graded potentials when making your NeuroML.

In order to specify weight, `<electricalConnectionInstance>` and `<continuousConnectionInstance>` can be replaced with the similar `<electricalConnectionInstanceW>` and `<continuousConnectionInstanceW>` tags.

- There also are two equivalent but a little bit shorter tags `<electricalConnection>` and `<continuousConnection>`; but tags like `<electricalConnectionW>` and `<continuousConnectionW>` are *not* in the standard and will be just *ignored* by EDEN, watch out.

[54] https://neuronaldynamics.epfl.ch/online/Ch3.S1.html#Ch3.F1
[55] http://www.scholarpedia.org/article/Neuronal_cable_theory#Electrical_Synapses
[56] https://docs.neuroml.org/Userdocs/Schemas/Synapses.html#gapjunction
[57] https://docs.neuroml.org/Userdocs/Schemas/Synapses.html#silentsynapse

Note that these *continuous* connections can't have a `delay` associated with them, though that would help make neural mass models[58]. If you are interested in continuous interactions *with* pure lag, do state your requirements in a feature request for NeuroML[59].

```
%%writefile MultiCell_Model.nml
<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_v2.1.xsd">
<iafCell id="MyFirstCellType" C="200 pF" leakConductance="10 nS" leakReversal="-70␣
↪mV" reset="-70mV" thresh="-50mV" />

<!-- Add a new a new cell type -->
<izhikevich2007Cell id="MySecondCellType" 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">
    <notes>Regular spiking cell</notes>
</izhikevich2007Cell>

<pulseGenerator id="MyFirstInputSource" delay="100ms" duration="500ms" amplitude=
↪"0.21nA"/>
<expOneSynapse id="MyFirstSynapseType" gbase="10nS" erev="0V" tauDecay="5ms"/>

<!-- Add a new synapse type, a gap junction ! -->
<gapJunction id="Syn_gapJunction" conductance="2000pS"/> <!-- With a pretty huge␣
↪conductance, to demonstrate the effect -->

<!-- Add a new synapse type, a dummy graded synapse ! -->
<silentSynapse id="Syn_SilentSynapse"/> <!-- No parameters, no effect, no worries -
↪->

<network id="MyNetwork" type="networkWithTemperature" temperature="37degC" >
    <population id="MyFirstPopulation"  component="MyFirstCellType"  size="2"/>
    <population id="MySecondPopulation" component="MySecondCellType" size="2"/>

    <!-- Apply an inputList on the first population -->
    <inputList id="MyFirstInputList" population="MyFirstPopulation" component=
↪"MyFirstInputSource">
        <!-- With one stim placement on it -->
        <input id="0" target="MyFirstPopulation[0]" destination="synapses"/>
    </inputList>
     <!-- Apply an inputList on the second population -->
    <inputList id="MySecondInputList" population="MySecondPopulation" component=
↪"MyFirstInputSource">
        <!-- With two stim placements on it, one for each cell -->
        <inputW id="0" target="0" destination="synapses" weight="0.4"/>
        <inputW id="1" target="1" destination="synapses" weight="0.3"/>
    </inputList>

    <!-- Apply the AP synaptic projection -->
    <projection id="MyFirstApProjection" presynapticPopulation="MyFirstPopulation"␣
↪postsynapticPopulation="MyFirstPopulation" synapse="MyFirstSynapseType">
        <!-- With one connection on it -->
        <connectionWD id="0" preCellId="0" postCellId="1" weight="1" delay="10 ms"/
↪>
    </projection>

    <!-- Apply the new graded synaptic projections ! -->
```

(continues on next page)

[58] http://www.scholarpedia.org/article/TheVirtualBrain#Modeling_Approach
[59] https://github.com/NeuroML/NeuroML2/issues/new?assignees=&labels=enhancement&projects=&template=feature_request.md&title=

(continued from previous page)

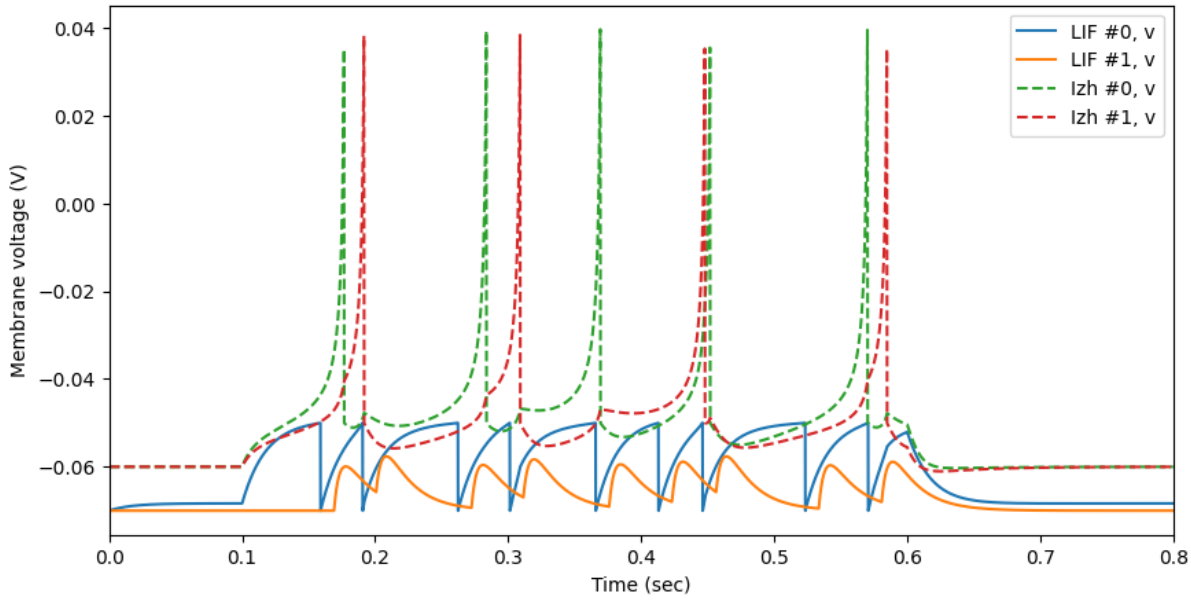
```
    <electricalProjection id="MyGapProjection" presynapticPopulation=
↪"MySecondPopulation" postsynapticPopulation="MySecondPopulation">
        <electricalConnectionInstanceW id="0" preCell="0" postCell="1" preSegment=
↪"0" postSegment="0" synapse="Syn_gapJunction" weight="1"/> <!-- seg 0 is the␣
↪only one for point neurons -->
    </electricalProjection>
    <continuousProjection id="MyUnsymmetricGap" presynapticPopulation=
↪"MySecondPopulation" postsynapticPopulation="MyFirstPopulation">
        <continuousConnection id="0" preCell="1" postCell="0" preComponent="Syn_
↪SilentSynapse" postComponent="Syn_gapJunction"/>
    </continuousProjection>

</network>
</neuroml>
```

```
Overwriting MultiCell_Model.nml
```

The standard NeuroML library is a bit sparse in continuous synapses: more than the component types we just used, it includes one detailed chemical receptor model[60] that has been used for modelling[61] the pyloric rhythm of the crustacean stomatogastric ganglion. But this is only because there are no other typical, commonly agreed interaction models yet. To implement your preferred interaction model, see the LEMS tutorial (page 61) and the implementation[62] of said STG model, and EDEN's multi-flux capability (page 121) for more options.

Let's run this simulation:

```
results = runEden('LEMS_MultiCell_Sim.xml')
show_multicell_results(results)
```



Observe how the membrane voltage each Izhikevich cell is sharply affected whenever the other one fires; the one-way graded potential applied to the first LIF cell has an even more pronounced effect. The steady leak due to different resting potentials for the two cells is also visible at the start (before the current clamps activate), feel free to change `vr` to same in both cells if it's a concern.

[60] https://docs.neuroml.org/Userdocs/Schemas/Synapses.html#gradedsynapse

[61] https://www.researchgate.net/profile/Eve-Marder/publication/2267079_Modeling_Small_Networks/links/54482da50cf2d62c3052a237/Modeling-Small-Networks.pdf

[62] https://github.com/NeuroML/NeuroML2/blob/NMLv2.1/NeuroML2CoreTypes/Synapses.xml#L574

2.7 NeuroML file structure conventions

In the previous examples, we used *two* files per model. This is not the only option; we could also have split the parts of the simulation differently, as long as the file for each part is `<Include>`d by the file we pointed the simulator to in `runEden`, or by a another file `<Include>`d by a file that's ultimately `<Include>`d by the simulation file, and so forth. Or (when using EDEN) we could just roll all model parts side by side with the `<Simulation>` in a *single file* if that's more convenient. (By the way, if the same file is `<Include>`d by different files, that's no problem.)

As a neural modelling project gets more detailed, and its parts get re-mixed in different ways (also between projects), there are some NeuroML file *naming*conventions[63] that may help manage the model contents:

- each `<cell>` type is defined in its own file with extension `.cell.nml`;
- each `synapse` type used is defined in its own file with extension `.synapse.nml`;
- the `<network>` to be simulated is placed in its own file with extension `.net.nml`;
- the `<simulation>` information are placed in their own file with a name like `.xml`.
- There is no common convention for `<input>`s; they are usually added to the `.net.nml` files, but they could also be added to NeuroML files in some other way that makes sense.

Of course, if there are overriding reasons the strcuture can be different in practice, as mentioned previously.
For example, if a set of different model parts is also meaningful as a named `collection`, there could be an additional `.nml` file that `<Includes>` the files for said parts.

The following code cells show how the parts comprising the last model can also be laid out in a more modular file structure.

```
%%writefile MultiCell.net.nml
<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_v2.1.xsd">

<!-- Include all the re-usable components ! -->
<include file="MultiCell_Iaf.cell.nml" />
<include file="MultiCell_Izh.cell.nml" />
<include file="SimpleChemical.synapse.nml" />
<include file="SimpleGapJunction.synapse.nml" />
<include file="Graded_Dummy.synapse.nml" />

<!-- Leave the input sources here i guess, or not ... -->
<pulseGenerator id="MyFirstInputSource" delay="100ms" duration="500ms" amplitude=
↪"0.21nA" />

<!-- And leave here just the network ! (and auxiliaries) -->
<network id="MyNetwork" type="networkWithTemperature" temperature="37degC" >
    <population id="MyFirstPopulation"  component="MyFirstCellType"  size="2"/>
    <population id="MySecondPopulation" component="MySecondCellType" size="2"/>

    <inputList id="MyFirstInputList" population="MyFirstPopulation" component=
↪"MyFirstInputSource">
        <input id="0" target="MyFirstPopulation[0]" destination="synapses"/>
    </inputList>
    <inputList id="MySecondInputList" population="MySecondPopulation" component=
↪"MyFirstInputSource">
        <inputW id="0" target="0" destination="synapses" weight="0.4"/>
        <inputW id="1" target="1" destination="synapses" weight="0.3"/>
```

(continues on next page)

[63] https://docs.neuroml.org/Userdocs/Conventions.html#file-naming

(continued from previous page)

```
    </inputList>

    <projection id="MyFirstApProjection" presynapticPopulation="MyFirstPopulation"␣
↪postsynapticPopulation="MyFirstPopulation" synapse="MyFirstSynapseType">
        <connectionWD id="0" preCellId="0" postCellId="1" weight="1" delay="10 ms"/
↪>
    </projection>

    <electricalProjection id="MyGapProjection" presynapticPopulation=
↪"MySecondPopulation" postsynapticPopulation="MySecondPopulation">
        <electricalConnectionInstanceW id="0" preCell="0" postCell="1" preSegment=
↪"0" postSegment="0" synapse="Syn_gapJunction" weight="1"/> <!-- seg 0 is the␣
↪only one for point neurons -->
    </electricalProjection>
    <continuousProjection id="MyUnsymmetricGap" presynapticPopulation=
↪"MySecondPopulation" postsynapticPopulation="MyFirstPopulation">
        <continuousConnection id="0" preCell="1" postCell="0" preComponent="Syn_
↪SilentSynapse" postComponent="Syn_gapJunction"/>
    </continuousProjection>
</network>
</neuroml>
```

```
Writing MultiCell.net.nml
```

```
%%writefile MultiCell_Iaf.cell.nml
<neuroml>
<iafCell id="MyFirstCellType" C="200 pF" leakConductance="10 nS" leakReversal="-70␣
↪mV" reset="-70mV" thresh="-50mV" />
</neuroml>
```

```
Writing MultiCell_Iaf.cell.nml
```

```
%%writefile MultiCell_Izh.cell.nml
<neuroml>
<izhikevich2007Cell id="MySecondCellType" 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">
    <notes>Regular spiking cell</notes>
</izhikevich2007Cell>
</neuroml>
```

```
Writing MultiCell_Izh.cell.nml
```

```
%%writefile SimpleChemical.synapse.nml
<neuroml>
<expOneSynapse id="MyFirstSynapseType" gbase="10nS" erev="0V" tauDecay="5ms"/>
</neuroml>
```

```
Writing SimpleChemical.synapse.nml
```

```
%%writefile SimpleGapJunction.synapse.nml
<neuroml>
<gapJunction id="Syn_gapJunction" conductance="2000pS"/> <!-- With a pretty huge␣
↪conductance, to demonstrate the effect -->
</neuroml>
```

```
Writing SimpleGapJunction.synapse.nml
```

```
%%writefile Graded_Dummy.synapse.nml
<neuroml>
```

(continues on next page)

(continued from previous page)

```
<silentSynapse id="Syn_SilentSynapse"/> <!-- No parameters, no effect, no worries -
↪->
</neuroml>
```

```
Writing Graded_Dummy.synapse.nml
```

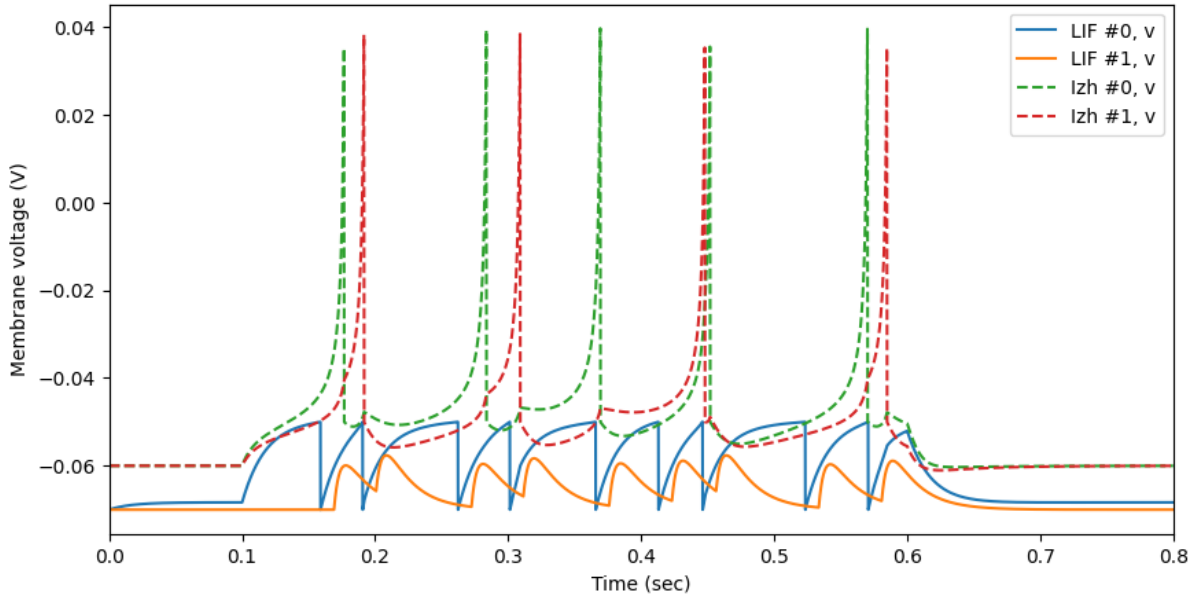
Make another LEMS file that points to the new `.net.nml`:

```
%%writefile LEMS_MultiCell_Modular_Sim.xml
<?xml version="1.0" encoding ="UTF-8" ?>
<Lems>
<include file="MultiCell.net.nml" />
<!-- Set the seed to freeze the random numbers -->
<Simulation id="MySim" length="1 s" step="100 us" seed="20190109" target="MyNetwork
↪" >
    <OutputFile id="MyFirstOutputFile" fileName="results.gen.txt">
        <OutputColumn id="my_first_vm"  quantity= "MyFirstPopulation[0]/v"/>
        <OutputColumn id="my_second_vm" quantity= "MyFirstPopulation[1]/v"/>
        <OutputColumn id="my_third_vm"  quantity="MySecondPopulation[0]/v"/>
        <OutputColumn id="my_fourth_vm" quantity="MySecondPopulation[1]/v"/>
    </OutputFile>
</Simulation>
<Target component="MySim"/>
</Lems>
```

```
Writing LEMS_MultiCell_Modular_Sim.xml
```

And re-run the modular version of the model.

```
results = runEden('LEMS_MultiCell_Modular_Sim.xml')
show_multicell_results(results)
```



This concludes the introduction to the basics of NeuroML. You may proceed to lean more about spatially detailed cells (page 33) and making your own mechanisms with LEMS (page 61).

CHAPTER

THREE

NEUROML AND SPATIALLY DETAILED CELLS

The "NeuroML basics (page 13)" article introduced the NeuroML framework and data format for expressing spiking neural networks. In the process, it explained how simplified neuron models, such as LIF and Izhikevich types, can be specified, parameterised and used to construct networks.

These simplified models are also often called *abstract cells* or *point neurons*, because they reduce the detailed anatomy and biophysical specificity found in nature, into just a few *scalar values* that produce a (roughly) equivalent response.

There is also another, more "direct" approach to simulate biophysics, by taking into account the spatial extent of the neuron and the immediate electro-chemical reactions that are supposed to take place along it. The "spatial" or "physically-based" formulation is thus composed of the *shape* (called *morphology* in neuro-anatomy) of the neuron, and the biophysical mechanisms that are present across the morphology. Under this formulation, most *cell-internal* biophysics properties (such as ion channels, concentration models, and physical membrane and cytosol properties) are modelled as being *finely distributed* over spans of neurite; whereas points of *interaction with* the neuron such as synapses and input probes are modelled as non-spatial ("point") processes that are located at a specific spot on the neuron.

Aside: Once the biophysical effects are captured and quantified well enough through in-vitro and/or in-silico experiments, these effects can be added back to simplified neuron models: the aim is to preserve the effect while simplifying the dynamics equations and their computational requirements. These *improved* simplified neurons can be expressed in NeuroML with the help of LEMS, as shown in the relevant article (page 61).

This article introduces the additional attributes that spatially-detailed cell models have compared to point neuron models, outlines how these attributes can be stated through NeuroML, and shows the usual steps to construct spatially detailed neuron models, run them, and display and assess the simulation's results.

3.1 Putting together a cell type

In NeuroML, spatially-modelled neurons are described through the generic-sounding `<cell>` tag, one for each *neuron type*. This tag must "contain" two other tags: a `<morphology>` tag that describes the *shape* of the neuron type, and a `<biophysicalProperties>` tag that describes the biophysics going on throughout the shape of the cell. Each of these tags will be explained in *adequate* depth in the following sections.

- **Tip**: If each neuron has a different shape, each neuron will need to be in its own `<cell>` type.

When different cell types share a morphology or set of biophysics, or it's just more convenient that way, the `<morphology>` and `<biophysicalProperties>` tags may have unique `id` attributes and be defined *independently* in the top-level `<neuroml>` tag. In this case, the `<cell>` tag may, instead of containing a `<morphology>` or `<biophysicalProperties>` tag inside it, have a `morphology` or `biophysicalProperties` *attribute* that names the stand-alone tag with associated `id`.

- **Note**: If different `<cell>` types point to the same `<biophysicalProperties>` this way, the contents of that tag will be duly *interpreted* according to each `<cell>` type's `<morphology>` (including `<segmentGroup>`s that may differ in each case).

Let's see how the attributes of a spatially-modelled neuron are described with NeuroML; first the shape (page 35) of the neuron, and then the biophysics (page 42) that are present on each part of the morphology.

3.2 Peeking at the neuron model's structure

Before diving into the details, to provide some context on what we're about to discuss, let's visualise a spatially detailed neuron model first. For that we'll use explain_cell (page 241) that will be further explained in the following (page 41).

A typical neuron model with high spatial detail is NMLCL000625: A Neocortical L6 basket cell[64] from the NeuroML-DB. Feel free to search for and upload other models as well; a simpler stick-figure model is also suggested at the end of this section.

```
nmldb_cell_name = 'NMLCL000625'; zip_file_name = f'{nmldb_cell_name}.zip'
# Download the zip file with the model
import urllib.request
# Because NMLDB is having some issues with HTTPS, don't demand HTTPS verification
import ssl; ssl._create_default_https_context = ssl._create_unverified_context
urllib.request.urlretrieve(f'http://neuroml-db.org/GetModelZip?modelID={nmldb_cell_
↪name}&version=NeuroML', zip_file_name)
# and unpack it
from zipfile import ZipFile
with ZipFile(zip_file_name, 'r') as zipp: zipp.extractall(nmldb_cell_name+'/')
# Find the cell's .nml file
import os
cell_filenames = [ nmldb_cell_name+'/'+name for name in os.listdir(nmldb_cell_
↪name) if name.endswith('.cell.nml') ]
assert len(cell_filenames) == 1, "Didn't find a lonely cell nml file!"
```

An important thing to understand and remember is that when simulating across the body of a neuron, this is not done in perfect detail, for practical reasons. The shape of the neuron is *discretised* (sort of "chopped") into chunks called *compartments* (or other terms, as seen on the terminology table (page 230).

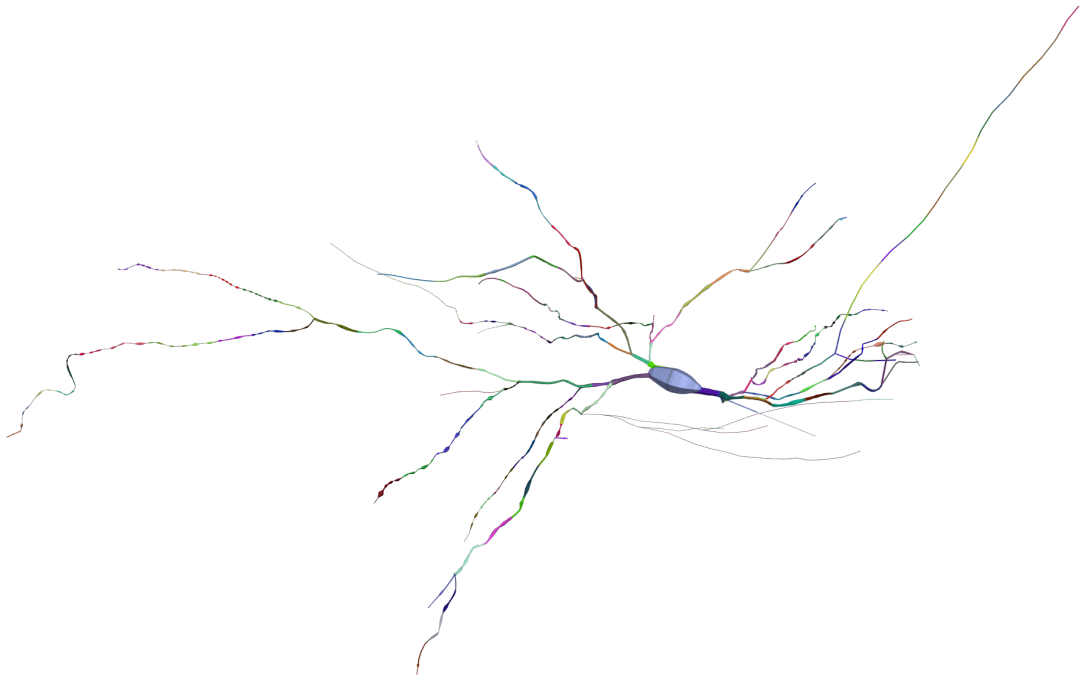
To illustrate, we'll paint each *compartment* differently and show both the shape of the modelled neuron, as well as how it is *discretised* for simulation. Right after, we'll see how this neuron was built in NeuroML.

```
# Get some information about the cell, like shape and no. of compartments
import eden_simulator
cells_info = eden_simulator.experimental.explain_cell(cell_filenames[0])
# one cell in the model file, get it
cell_info = list(cells_info.values())[0]
# count the per-compartment information that explain_cell provides
nCellComps = len(cell_info['comp_midpoint'])

# Create some different colours to show the different compartments
import numpy as np
rng = np.random.default_rng(seed=20240901)
comp_colors = rng.random([nCellComps, 3])
```

```
# Show a neuron, painted by compartment
from eden_simulator.display.spatial import k3d as ek3d; import k3d
plot = Comps_plot = ek3d.Plot(grid_visible=False, menu_visibility=False, camera_
↪auto_fit=True);
plot += ek3d.plot_neuron(cell_info, comp_colors)
# TODO zoom a bit closer
plot.show_html()
```

[64] https://neuroml-db.org/model_info?model_id=NMLCL000625

```
<IPython.core.display.HTML object>
```



Try to read the NeuroML description[65] of the cell (check for code-folding arrows at the start of nested lines, on your web browser or code editor). Scroll to near the end of the file and find that these endless lines of `<segment>`s and `<segmentGroup>`s have given way to a `<biophysicalProperties>` tag with only a few `<channelDensity>`s and other properties inside it. Don't push yourself to understand all of the file just yet; all of its intricacies will be explained in depth by this article.

In case the above file is unwieldy to use (or out of interest), see also a simpler `<cell>` model with a *stylised* (or "stick figure") shape, for granule cells of the cerebellar cortex[66]; scroll down to `Granule_98_3D.cell.nml` and click on `( view NeuroML )` to read the NML description. As you'll see by studying its `<morphology>` in the following (and also by clicking on `Morphology` on that page), this cell has a small spherical soma linked to a very long axon, bifurcated into a `T` shape. This axon is called a parallel fiber[67] and it goes in the same direction relative to the Cb[68] cortex, for all granule cells. Likewise, other types of neuron in the brain have different distinctive anatomical features and shapes[69].

Let's see what NeuroML `<cell>`s are made of then.

3.3 The morphology of the cell

In neuro-anatomy, what's called the *morphology* of a cell is all the *spatial information* about it. This is described in NeuroML with a `<morphology>` tag, which contains `<segment>`s that the cell is built of and `<segmentGroups>` which define areas of particular interest on the cell.

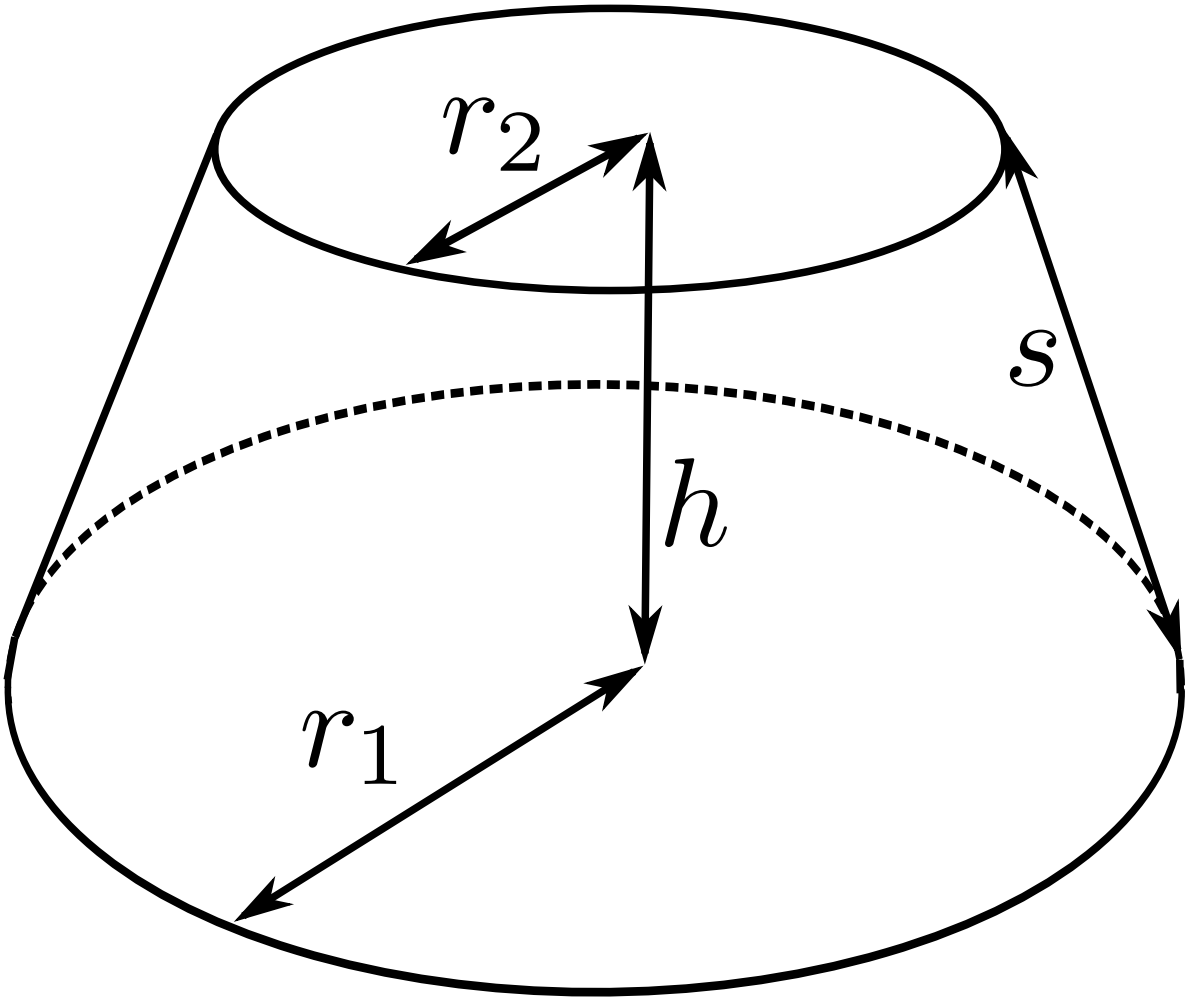
See also the official NeuroML guide[70] on the following concepts and moving data to/from other technologies.

[65] `https://neuroml-db.org/render_xml_file?modelID=NMLCL000625`
[66] `https://neuroml-db.org/model_info?model_id=NMLCL000002`
[67] `https://en.wikipedia.org/wiki/Parallel_fiber`
[68] `http://www.thehumanbrain.info/database/nomenclature.php#C`
[69] `https://en.wikipedia.org/wiki/Neuron#Structural_classification`
[70] `https://docs.neuroml.org/Userdocs/ImportingMorphologyFiles.html`

3.3.1 Segments

The morphology of a neuron, in NeuroML and virtually all simulators at the modelling level of networks with multi-compartment neurons, is formulated as a tree structure[71] of branching neurite tubes. The tubes can have a varying diameter over their length; they can be broken down into *truncated cones* (specifically non-oblique frusta[72]) that are joined together to form the neuron's neurite tree[73].

Here is a sketch of a NeuroML-style frustum for reference, from Wikipedia[74]:



A `<segment>`'s geometry can be defined as a pair of *ends*, where each end is a circular disc with a *centre* in space, and a *diameter* (or equivalently radius). The planes on which the discs lie are perpendicular to the line segment joining the two centres, for non-oblique frusta.

Within a `<morphology>`, the `<segment>`s are arranged in a *directed tree* structure, which means that all except for one are connected to a "parent" segment. The one that remains is called the "root" and is the "ancestor" of all other segments. By convention, the root is considered to be the soma (from which all processes emerge, after all), so that each "parent" segment is in turn located a little closer to the soma. Hence, for each segment we name its end that's near the parent as *proximal* and the one that's further from the parent as *distal*. (For the *root segment*, we can name the two ends either way.)

Of course, some questions arise when it's time to model the soma as a tree of frusta. The usual practices are:

- When the soma can be modelled as one single compartment and looks bulbous, it is allowed to be modelled as a *sphere* instead:

[71] `https://en.wikipedia.org/wiki/Tree_(data_structure)`
[72] `https://mathworld.wolfram.com/ConicalFrustum.html`
[73] `https://en.wikipedia.org/wiki/Neurite`
[74] `https://en.wikipedia.org/wiki/File:CroppedCone.svg`

  - The segment's two ends have the same *diameter*, and the distance between them is either *none* (as a convention that's accepted to represent a sphere) or equal to the diameter (in which case, the resulting cylinder has the same (membrane) surface area as a sphere with the same diameter).

- Alternatively, for an e.g. elongated soma, a modeller could *slice it like a tuber* along the longest axis, to better preserve the original shape.
- When modelling a bipolar cell, modellers may opt to have neurites grow from the "proximal" end of a segment that represents the soma. More commonly for any type of cell, neurites may sprout from a segment near the middle of the soma.

Note

It is worth noting that the soma is not that nicely captured in this formulation, which may impact simulation accuracy *within the soma*. To correct for this, the biophysical factors over the soma are usually empirically adjusted to yield the correct "effective" values.

The neuron's shape is then described inside the `<morphology>` as a list of `<segment>` tags like this:

```
<morphology>
    ...
    <segment id="0" name="Soma_0">
        <proximal x="0" y="0" z="0" diameter="20"/>
        <distal x="10" y="0" z="0" diameter="20"/>
    </segment>
    <segment id="10" name="Axon_1">
        <parent segment="0"/>
        <proximal x="0" y="0" z="0" diameter="15.6"/>
        <distal x="20" y="7" z="0" diameter="5"/>
    </segment>
    <segment id="11" name="Axon_2">
        <parent segment="10" fractionAlong="0.6"/>
        <distal x="12" y="9" z="0" diameter="5"/>
    </segment>
    ...
```

And the meaning of the attributes and the enclosed `<proximal>` and `<distal>` tags is as follows:

- `id`: A distinct integer to name give the segment a name.
- `name`: A more human-readable name to describe the segment with, but it is not actually used as part of the model; the `id` number is, instead.
- Within the segment there are more tags:
  - Two tags called `proximal` and `distal` to define the extent of each frustum. Each has the following attributes:
    - `x`, `y`, `z`: The coordinates of the end's centre in 3-d space, in microns.
    - `diameter`:The diameter of the frustum on that end, in microns.
  - For non-root segments, a `<parent>` tag specifying a the `id` of each's parent `segment`. Said parent must have been declared previously in the `<segment>` sequence.
    - For more precision, the exact *point on the parent segment* to which this segment is attached, can also be specified. The point is given as the `fractionAlong` the parent segment's length, from `0` (`proximal` "start" point) to `1` (`distal` "end" point).
      - The default value is `1` (attached to the parent's `distal` end).
  - If the segment has a `parent`, defining its `<proximal>` end is optional; by default, it will be same as the point interpolated by `fractionAlong`, on the `parent` segment. the `x`, `y`, `z` as well as the `diameter` are interpolated between the two ends of the parent.

Standards in morphology coordinates

As expected, the 3-D coordinates for the `proximal` and distal ends of each segment must have an *origin* $(0, 0, 0)$; but what this is, is up to who traced the neurons. A common convention to place the origin near the middle of the soma.
Likewise, what the $\vec{x}$, $\vec{y}$, $\vec{z}$ directions represent is, unfortunately, up to the person who traced the neuron; there is no commonly agreed convention. Hence, you may have to tweak the coordinates to your favourite coordinate system:

- Observe a discernible feature e.g. the axon;
- then see which direction it extends to in the modeller's coordinates;
- see which direction it should be following in your coordinates, and you'll understand how to map this axis.
- If it's important, repeat for the other two axes (e.g. with planarity, or asymmetrical shape) to find out how to transform the other axes.

Note: the origin used for the morphology is also used to (dis)place the cell `<instance>`[75]s within a `<network>`[76]. See how this is done for example, on the Network tutorial (page 129) and `EdenCustomSetup` example (page 93).

Note: The NeuroML formulation is similar to very common SWC file format[77] that describes morphologies. The additional features that NeuroML offers beyond SWC are:

- The `proximal` end of a `segment` can be different from the `parent`'s `distal`; i.e. it can have different diameter and displacement in space.
- The `<segmentGroup>` concept is much more powerful, in that the groups have names and descriptions (or metadata), and can *contain* and/or *overlap with* other groups.
- The preferable discretisation of the cell can be provided, which is important for accurate *and* efficient simulation.

Specifying a location on a cell

Given the tree of `<segment>`s that outlines the shape of the neuron, a site on its surface can be designated as a NeuroML `<segment>` `id`, and `fractionAlong` the axial direction of said `<segment>`. This pair of `segment` and `fractionAlong` is used to:

1. precisely attach (page 52) synapses and stimulation probes to cells;
2. record local action potentials with `<EventOutputFile>`s, and trajectories of quantities in `<OutputFile>`s — to specify `fractionAlong` together with the `segment`, use EDEN's extended LEMS paths (page 87).

In all these cases:

- If the `segmentId` is not specified, it is assumed `0` : typically the soma (but look out), or the entirety of a point neuron.
- If the `fractionAlong` is not specified, it is assumed `0.5`: the middle of the segment, doesn't matter for point neurons. (or spatially detailed ones that comprise just one compartment, for that matter)

Aside: One could also ask for a specific *depth* into the cytosol and *azimuth* on the plane with the same `fractionAlong`, but NeuroML (and SNN simulators in general) hasn't yet reached this level of spatial detail in the space of representable models (or there would be a way to model the soma in detail already).

[75] `https://docs.neuroml.org/Userdocs/Schemas/Networks.html#instance`
[76] `https://docs.neuroml.org/Userdocs/VisualisingNeuroMLModels.html#view-the-3d-structure-of-your-model`
[77] `https://doi.org/10.1016/S0165-0270(98)00091-0`

3.3.2 Segment groups

The *segments* that build up the neuritic tree can be organised into *groups* that represent parts of the neuron. Groups are like mathematical sets[78] in that they one can completely include another, or they can overlap at an *intersection subset* of segments. This also means that segments don't have to be part of one group exclusively.

Segment groups thus demarcate interesting *regions* within a neuron, for looking at, or also relevant to our model.

In NeuroML, segment groups are defined within a `<morphology>`, as `<segmentGroup>` tags that include member segments, and/or other segment groups.

The `<segmentGroup>` tag has the following properties:

- `id`: unique name for the group. **Note**: Unlike the `id`s of `<segment>`s, this can be any alphanumeral.
- `neuroLexId`: A machine-readable NeuroLex[79] keyword that identifies the *type* of the neuron's region. For example, GO:0043025[80] stands for "soma", GO:0043024[81] stands for "axon", and GO:0097447[82] stands for "all the dendrites".
  - The identifier `sao864921383` has a special meaning for the implicit discretisation (page 40) of the neuron, explained in a following section.

Each `<segmentGroup>` tag may also contain other tags, that list its properties:

- `<member segment="<seg id>"/>`: Adds a single segment to this group.
- `<include segmentGroup="group id"/>`: Adds a previously defined group to this group.
- There also exist `<path>`[83] and `<subTree>`[84] tags to include segment spans in the standard, but EDEN doesn't support them yet; they have not been used in practice. (**Note**: Segment groups can always be constructed programmatically to exactly fit the modeller's intention.)
- `<property tag="<name>" value="<content>">` tags can also be used to add more information about the segment group. The `tag="numberInternalDivisions"` may be used to help with discretisation (page 40), *when* `neuroLexId=sao864921383` "compartment[85]".
- Finally, the `<inhomogeneousParameter>` tag can be used to associate a spatial scalar function with the group. This function can then be used to specify non-uniformity in ion channel distributions; hence the tag is explained in the relevant section (page 50).

The group `"all"`

If it is not otherwise defined by the modeller, the group `"all"` is defined as a shorthand that captures the entirety of the cell. It is, for example, useful to place mechanisms *all over* the cell, such as the membrane's intrinsic electrical leak, or possibly an ion channel distribution that's present throughout the cell's membrane.

[78] https://en.wikipedia.org/wiki/Set_theory
[79] https://doi.org/10.3389/fninf.2013.00018
[80] https://amigo.geneontology.org/amigo/term/GO:0043025
[81] https://amigo.geneontology.org/amigo/term/GO:0030424
[82] https://amigo.geneontology.org/amigo/term/GO:0097447
[83] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#path
[84] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#subtree
[85] https://scicrunch.org/scicrunch/interlex/view/ilx_0109858

3.3.3 Simulation aspect: Discretisation into compartments

So far, we saw how the anatomical shape of the neuron is expressed in NeuroML. But as we saw at the start (page 34) of the article, this shape is *split* (in mathematical language, *discretised*) into little bits, called *compartments*. The neuron is approximated by these compartments in the same way that an image can be approximated by pixels. There often are good reasons for the modeller to *control* how this discretisation happens; and NeuroML allows such control, albeit through more implicit than explicit rules.

Note: Although the exact discretisation method shouldn't matter in theory, it is often important in practice. Naturally, a too big (or less often, too *small*) compartment may provide a poor reproduction of the ideal "real" thing, as we'll see in a following exercise (page 52). *But* there are models where compartments (often containing certain sensitive biophysical mechanisms) have to have an *exact* size to reproduce the phenotype!

The implicit rules of NeuroML for discretisation are as follows:

- `<segmentGroup>`s tagged with `neuroLexId="sao864921383"` follow "cable" rules (page 40);
- Each `<segment>` that doesn't belong in such a corresponds to one individual compartment.
  - This is how discretisation is always done, for example, in the GENESIS simulator.

Tip: If your modelling pipeline already offers its own discretisation (and it's not as easy to convert as with NEURON), consider *splitting* your `<segment>`s (or equivalent) at compartment boundaries, and then joining the segment parts of each compartment into "cable" `<segmentGroup>`s (see below). If your modelled compartments include branching points *within* them, consult also the `comp_area`, `comp_capacitance` and `comp_conductance_to_parent` information that `explain cell` provides (page 241), to make equivalent NeuroML-supported compartments; look out for the influence of altered `<segment>`s on their surface area and electrical tortuosity.

The *unbranched section* directive

NeuroML offers an implicit rule for discretising spans of unbranched neurite *cable*. The rule applies when the span is gathered into a `<segmentGroup>` with the attribute `neuroLexId="sao864921383"`. In that case, the length of all contained segments is added up and divided among a number of compartments, so that each compartment takes over an equal fraction of the total cable length. The number of compartments can be specified by adding inside the same `<segmentGroup>` the tag `<property tag="numberInternalDivisions" value="<number>">`.

- If `neuroLexId="sao864921383"` and this tag is *not* specified, the whole group is assumed to map to *one* compartment (i.e. `numberInternalDivisions = 1`).

See also these tickets:

- https://github.com/NeuroML/pyNeuroML/issues/77
- https://github.com/NeuroML/NeuroML2/issues/156

> **Tip**
>
> `neuroLexId="sao864921383"` corresponds to the `<cable>`s of the previous incarnation of NeuroML, NMLv1[86]. It facilitates conversion of models that were originally written for the NEURON simulator, which shares the concept of continuous *sections* of neurite tube making up the neurons.

To prevent ambiguity, "cable" `<segmentGroup>`s are not allowed to overlap.

Refer also to NEURON's user guide and forum on wise choices[87] for `numberInternalDivisions`.

[86] https://github.com/NeuroML/org.neuroml1.model/blob/v1.9.1/src/main/resources/NeuroML1Schemas/Level1/MorphML_v1.8.1.xsd#L327

[87] https://nrn.readthedocs.io/en/latest/guide/faq.html#what-s-a-good-stretegy-for-specifying-nseg

But also keep in mind that uniform spacing could be a suboptimal discretisation method if the neurite diameter, or parameters of active mechanisms, vary wildly. Take care when building models, and consider fine-tuning the discretisation around the important points.

As a final note: For some models and experiments, the discretisation can also be *reduced* to a smaller number of compartments (see this early paper[88] and other advances since). This will affect the neuron's dynamics (by more or by less) and *drastically* cut down the time it takes to run a simulation. Check if it's applicable for your use case; but also be aware of the trade-offs, and the points where the decision will have to change.

Retrieving and using the discretisation

After *specifying* the discretisation, *getting it back* can also prove useful. Typical applications are:

- displaying a changing quantity across the neuron;
- visually inspecting (page 34) the discretisation's fineness while modelling;
- controlling non-uniform biophysical properties (page 50) at the finest level (page 99);
- better reconciling the equations between domains of the simulation, when the *surroundings* of the neuron matter; see also the LFP (page 147) and imposed field (page 159) examples.

EDEN can help with extracting the discretisation of neurons, through the auxiliary `explain cell` feature (page 241), as seen above (page 34). This feature produces a breakdown of cell types into the compartments that make them (which may have a many-to-many relationship with `<segment>`s in general), and the geometrical and physical properties of each compartment such as area and electrical capacitance of the membrane, and distance from the soma along the neurite path. It also produces additional compartment-related information, such as the list of compartments that belong to each segment group (page 39) and a 3-D polygon mesh that represents the neuron's morphology, and can be coloured per individual compartment.

3.3.4 Visualisation example

Now that we know what `<segment>`s and `<segmentGroup>`s are, let's see the neuron that was painted per compartment before, painted here separately for each main region. Note that all dendrites happen to be classified as "basal" in this cell model.

```
# For each group: NML name, color, description
group_info = [
    ('somatic', (0.8, 0.7, 0.4), 'Soma'),
    ('axonal' , (0.8, 0.2, 0.2), 'Axon'     ),
    ('basal'  , (0.5, 0.1, 0.5), 'Dendrites'), ]

# First, set a color for comps that are in none of these groups.
comp_colors[:,:] = [[0.5,0.5,0.5]]
# Then, paint the compartments for each group.
for group_name, color, _ in group_info:
    comps_in_group = cell_info['segment_groups'][group_name]['comps']
    comp_colors[comps_in_group] = color

# Show the neuron:
plot = Groups_plot = ek3d.Plot(grid_visible=False, menu_visibility=False);
plot += ek3d.plot_neuron(cell_info, comp_colors)

# And add some annotations.
for i, (_, color, description) in enumerate(group_info):
    plot += k3d.text2d(description, (0,0.9-0.15*i), color=ek3d.RgbToInt(color),
```

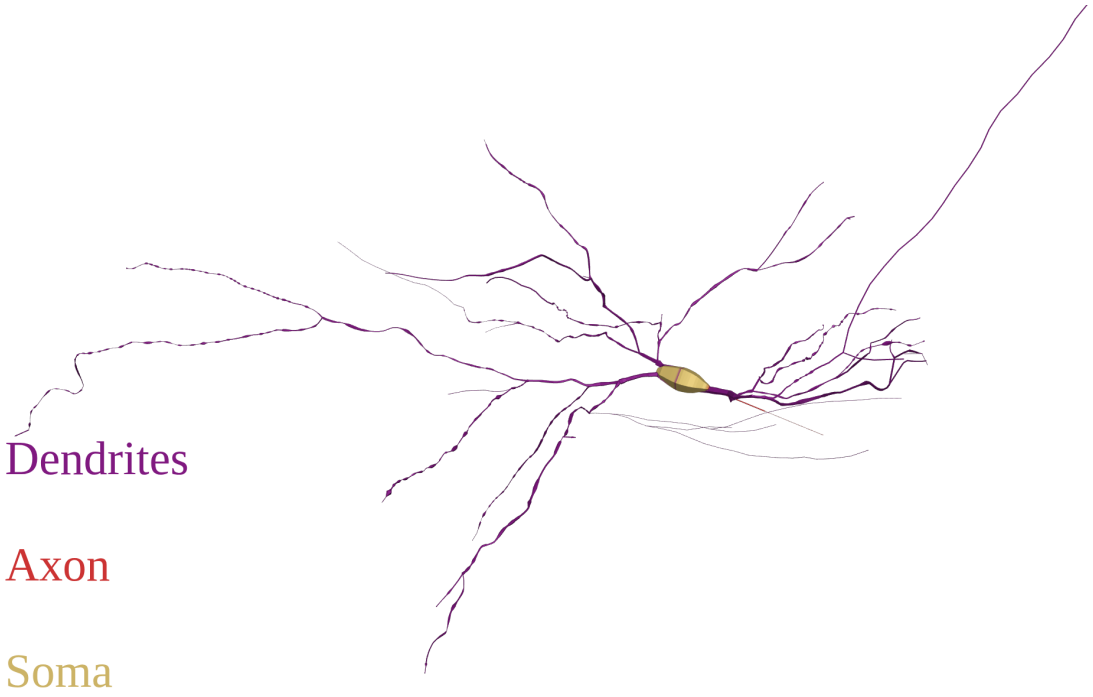
(continues on next page)

[88] https://doi.org/10.1016/0165-0270(93)90151-G

(continued from previous page)

```
                        is_html=True, label_box=False, size=2.5)

plot.show_html()
```

```
<IPython.core.display.HTML object>
```



3.4 The biophysics of the cell

After specifying everything about the shape and discretisation of a neuron, it's time to specify the *biophysical properties* that are present in (on, and around) it in our model. In NeuroML, these can be classified as:

- Electrical properties (page 43) of the neuritic cable, such as surface capacitance, axial resistance, and voltage threshold to register an action potential (to trigger post-synapses);
- Distributions of trans-membrane mechanisms, such as ion channels, passive leaks (page 43), and concentration models (page 51) (and pumps) of substances (such as ions) around the membrane;
- Individual (usually trans-membrane) instances of mechanisms (page 52), such as synapses and stimulation probes.

In NeuroML, all these are defined within a `<biophysicalProperties>` tag. This contains the `<membraneProperties>`, `<intracellularProperties>` and `<extracellularProperties>` tags where each in turn contains some of the properties (as more appropriate location-wise).

Until a later revision of NeuroML includes them, other properties may still be model-able today as LEMS components (page 61) that are designed to apply the desired effect.

All these are described for a cell type as *distributions* of parameters and mechanisms. The distributions apply all over the cell, or are bounded by a `<segment>` or `<segmentGroup>`, referring to the `<morphology>` that's already defined for the `<cell>` type.

- Each of the tags discussed in the following may have a respective `segment` or `segmentGroup` attribute; if neither is specified, it applies for the group (page 39) `all`.

Caution

Make sure you don't miss parts of the intended area when distributing over disparate segment groups; if you want a visual check, use the code above (page 41) to verify.

Fun fact: The user guide of the simulator Arbor calls[89] the process of placing mechanisms on neurons, "decoration" and specifically distributing attributes over the cell's morphology, "painting"; this author approves of those metaphors.

3.4.1 Electrotonic properties

Some general *electrical* properties of the neuritic cable making up the neuron can be provided with the following tags:

- `<specificCapacitance value="(capacitance per area)"/>`: The local *specific* capacitance of the membrane.
- `<resistivity value="(resistance times length)"/>`: The local resist*ivity* of the cytoplasm. (assuming current flows through the whole cross-section equally)
- `<initMembPotential value="(voltage value)" />`: The *initial value* for the cross-membrane voltage V_m when the simulation starts.
- `<spikeThresh value="(voltage value)"/>`: The threshold that V_m need to exceed, for an action potential to be *registered* in the model. This is also the point in time (plus possible delay) when *event-based synapses* are *activated*. (See also the comparison (page 27) between "classic" and graded synapses.)
  - **Note**: `<spikeThresh>` is optional, but spikes will *not* be registered from the areas where it isn't specified!

Except for `<resistivity>` which is located within `<intracellularProperties>`, the other properties are located within `<membraneProperties>`.

- **Important**: These properties may be *redefined* by tags with overlapping areas. In this case, each spot is assigned the value of the last tag that applies to it, in sequence within the `<biophysicalProprties>` tag. This is similar to how EDEN's `<EdenCustomSetup>` (page 89) extension works.

Note: *Passive* electrical leakage of the membrane is specified in NeuroML as a trans-membrane mechanism (page 43), typically a `<channelDensity>` of and `<ionChannel>` in the degenerate case (lacking any "`<gate>`"s).

Note

The dynamics of a spatially detailed cell are that of the passive neural cable[90], combined with the dynamics of the various *mechanisms* present on different parts of the cell.

3.4.2 Adding ion channels and other leaks to the membrane

More than simple physical properties, dynamic *mechanisms* may also be present over the neuron's morphology. The most common type of trans-membrane mechanism is *ion channels*. These "open" and "close" for specific ion species (or sometimes, any form of current) and their state is influenced by electro-chemical factors, plus the state they were at right before. The constituent parts of the ion channel that participate in opening and closing often called "*gates*".

Since a single ion channel is a quite small entity (compared to compartments (page 40) of the neuron), populations of ion channels are usually modelled as *distributions* finely spread over areas.

For the same reason, the ion channels' dynamics are usually modelled at the macroscopic (or smooth) level. In this level there are fractional *gate variables* that represent the *average* state of a gate population within the channel population, and there are *continuous-time*ODE[91] *'s* that govern the *average* rates of change between the "open" and

[89] https://docs.arbor-sim.org/en/latest/concepts/decor.html#cablecell-decoration
[90] https://en.wikipedia.org/wiki/Cable_theory
[91] https://en.wikipedia.org/wiki/Ordinary_differential_equation

"closed" states. (More faithful, *stochastic* models may still be implemented, but at this point in NeuroML they'll be more practical to model as stochastic processes for the macroscopic gate variables.)

Designing ion channels

NeuroML describes ion channels through the `<ionChannel>` tag, and includes a multiplied-gate *formulation* of ion channel models for modellers to work with. (Do keep in mind that more formulations than those out of the box can be described, using LEMS (page 78)).

All ion channel tags may have the following optional attributes:

- `type`: Can be used in place of the tag's name itself, to identify the particular type of ion channel; EDEN uses that or the tag name to use LEMS `<ComponentType>`s as ion channels.
- `species`: The sort of substance (typically an ion) that passes through the channel. If missing, the ion channel is assumed to permit *non-specific current*; that is, current flows without really affecting the local concentrations of tracked ion species.
- `conductance`: The conductance of a *single* ion channel. Must be set for a population (page 50) of *individual ion channels* rather than a coarse density.
  - **Note**: The dynamics are the same either way, the "population" is still assumed to be lumped into one aggregate mechanism instance (per compartment) in NeuroML.

Unless specially modelled through LEMS, the `<ionChannel>` tag contains `<gate>`s of various types, each with its multiplicity and dynamics as described in the following:

- Classic Hodgkin-Huxley gate models (page 47): `<gateHHrates>`, `<gateHHtauInf>`, `<gateHHratesTau>`, `<gateHHratesInf>`, `<gateHHratesTauInf>`
- HH sub-gate models (page 47): `<gateFractional>`
- Multi-state gates with general kinetics (page 47): `<gateKS>`
- Stateless gates (page 49): `<gateInstantaneous>`

Note: Constant leaks are also modelled in NeuroML as "degenerate" `<ionChannel>`s with no gates at all (thus a fixed conductance).

In NeuroML, an ion channel's dynamics are modelled separately from the local *reversal potential* and the *density* of its instances that are spread over cells; said parameters are modelled as part of the particular channel distribution (page 50) applied on cells.

Note that in the wild there may be more specific ion channel tag names such as `<ionChannelPassive>`, `<ionChannelHH>` and `<ionChannelKS>`; EDEN will parse and handle all of the contents regardless of stated type, so that potential *hybrid* models mixing up the gate types are also supported.

Refer also to the official NeuroML guide[92] and the LEMS extension section (page 78) of this guide on how to make your own ion channels in NeuroML.

In the following, the transition rates that govern the kinetics of gate variables are introduced, and in turn the various gate formulations that use these transition rates.

[92] `https://docs.neuroml.org/Userdocs/ConvertingModels.html#b-convert-channels-to-neuroml`

Transition rates

As mentioned above, the transition between states of individual gates can (in a big enough population) be macroscopically modelled as the *fraction* of gates that are in each state, and the rates of transition (as fraction per time, or T^{-1}) between states. These rates can be expressed in NeuroML as either "transition rate per existing fraction" functions (also seen as Hodgkin-Huxley[93] α, β "rate constants"). In chemical language, transition between two states can be expressed as the reversible reaction scheme:

$$\text{Closed} \underset{\alpha}{\overset{\beta}{\rightleftharpoons}} \text{Open}$$

Another, equivalent way to define these rate functions is through the quantities "steady-state value to converge to" and "time-constant to converge to it", which may written as x_∞, τ. Often this is easier to get as a measure (say, from literature), or just more convenient to write down as a formula. These two are directly convertible to and from α, β functions through the relations:

$$\tau = \frac{1}{\alpha + \beta},\ x_\infty = \frac{\alpha}{\alpha + \beta} \iff \alpha = \frac{x_\infty}{\tau},\ \beta = \frac{1 - x_\infty}{\tau}$$

Keep in mind that α, β (or τ, x_∞ just the same) *will* (as pairs) vary against external factors, such as trans-membrane voltage. (Naturally, if the rates couldn't change then ion channels would arrive and stay at a resting equilibrium, without oscillating.)

Rate formulae

NeuroML includes a small set of pre-defined rate formulae `types` as functions of membrane voltage only, in terms of: *rate* (for α, β functions), *steady state* (for x_∞) and *time constant* (for τ). These are used as in `<forwardRate>`s, `<reverseRate>`s, `<timeCourse>`s and `<steadyState>`s of Hodgkin-Huxley-style gates (page 47), and `forward`, `reverse` and `tauinf` `Transitions` of Markov-style gates (page 47) rates as applicable.
These pre-defined function types are:

- `HHExpVariable`[94] and `HHExpRate`[95], which are parameterisable *exponential curves*. The general formula is:

  $$f(V_m) = rate \cdot e^{\frac{V_m - midpoint}{scale}}$$

  Their attributes are:

  - `rate`: The *dependent-variable* scaling factor for the quantity represented by the curve, from the dimensionless inner curve to the function's dimensions. Is the value of the function when membrane voltage is `midpoint`.
  - `midpoint`: The reference point to center the curve around. The function has the value `rate` for this value of membrane voltage.
  - `scale`: The *independent-variable* scaling unit to evolve the curve by. Values `scale` apart on the independent variable are in a proportion of $1 : e$.
    - Notice that `midpoint` and `rate` are redundant for the exponential curve, set them in the way that most makes sense.
- `HHExpLinearVariable`[96] and `HHExpLinearRate`[97], which are parameterisable *rectified linear curves with exponential taper to zero*. The general formula is:

[93] `https://en.wikipedia.org/wiki/Hodgkin-Huxley_model`
[94] `https://docs.neuroml.org/Userdocs/Schemas/Channels.html#hhexpvariable`
[95] `https://docs.neuroml.org/Userdocs/Schemas/Channels.html#hhexprate`
[96] `https://docs.neuroml.org/Userdocs/Schemas/Channels.html#hhexplinearvariable`
[97] `https://docs.neuroml.org/Userdocs/Schemas/Channels.html#hhexplinearrate`

$$f(V_m) = rate \cdot \frac{a}{1 - e^{-a}}, \quad a = \frac{V_m - midpoint}{scale}$$

Their attributes are:

- `rate`: The *dependent-variable* scaling factor for the quantity represented by the curve, from the dimensionless inner curve to the function's dimensions. Is *twice* the value of the function when membrane voltage is `midpoint`.
- `midpoint`: The reference point to center the curve around. The function has the value `rate/2` for this value of membrane voltage.
- `scale`: The *independent-variable* scaling unit to evolve the curve by. The extent of the "taper zone" from linear to zero is proportional to this value.

- `HHSigmoidVariable`[98] and `HHSigmoidRate`[99], which are parameterisable logistic[100] *curves*. The general formula is:

$$f(V_m) = rate \cdot \frac{1}{1 + e^{-a}}, \quad a = \frac{V_m - midpoint}{scale} \tag{3.1}$$

Their attributes are:

- `rate`: The *dependent-variable* scaling factor for the quantity represented by the curve, from the dimensionless inner curve to the function's dimensions. Is *twice* the value of the function when membrane voltage is `midpoint`.
- `midpoint`: The reference point to center the curve around. The function has the value `rate/2` for this value of membrane voltage.
- `scale`: The *independent-variable* scaling unit to evolve the curve by. The extent of the "taper zone" between zero and maximum is proportional to this value.

- `fixedTimeCourse`[101], which represents a *fixed time constant* to converge to the steady state by. The only attribute is `tau`, the value of the time constant (with units).

More then the pre-defined set, additional rate formulae can be supplied using LEMS (page 79).

Using these rate formulae, we can specify various types of ion channel *gate* dynamics.

General attributes of gates

In the pre-defined NeuroML formulation, `<ionChannel>` (page 44) models contain one or more gates. The fraction of ion channels that are open in an aggregate population is assumed to be the multiplicative product[102] of the fractions of gates that are open, where the fraction of each gate that is open is the fraction of *gate particles*, multiplied over by the number or particle *instances* it takes to actually open a gate.
Every sort of `<gate>` in an `<ionChannel>` must specify the following attributes:

- `id`entifying name for the gate and its activation variable;
- multiplicity as `instances`;
  - The gate's activation is multiplied by itself this many times over, to scale the activation of the ion channels it belongs to (getting multiplied with the other gates in it as well).
- and if the tag name is `<gate>`, an attribute that clarifies the specific dynamics `type` of the gate.

[98] `https://docs.neuroml.org/Userdocs/Schemas/Channels.html#hhsigmoidvariable`
[99] `https://docs.neuroml.org/Userdocs/Schemas/Channels.html#hhsigmoidrate`
[100] `https://en.wikipedia.org/wiki/Logistic_function`
[101] `https://docs.neuroml.org/Userdocs/Schemas/Channels.html#fixedtimecourse`
[102] `https://en.wikipedia.org/wiki/Product_(mathematics)`

Classic Hodgkin-Huxley gate dynamics

The most common model for a gate assigns two possible states for it: either `Open` or `Closed`. In that case, one can consider x, the fraction of gates of a certain type in an ion channel population that are open; it follows that $1 - x$ of all such gates are closed. Hence, the *state* of that gate population is captured in just one number `x`, commonly called the "activation" or "gate variable".

The transition rates (page 45) between these two states can be expressed as α, β functions, or alternatively τ, x_∞ (or a mix of them!). The tag names for a gate under this formalism can be `<gateHHrates>`[103], `<gateHHtauInf>`[104], `<gateHHratesTau>`[105], `<gateHHratesInf>`[106], and even `<gateHHratesTauInf>`[107] so that `rates` stands for α, β, `tau` stands for τ, `inf` stands for x_∞, and which functions are specified is contained in the tag name.

As explained above, each of the tags may contain the transition rate formulae as child tags:

- α as `<forwardRate>` and β as `<reverseRate>`, each with a `type` attribute referring to a *per time*-valued function;
- τ as `<timeCourse>`, with a `type` attribute referring to a *time*-valued function;
- x_∞ as `<steadyState>`, with a `type` attribute referring to a *dimensionless* function.

If either τ or x_∞ are specified, they are consulted to simulate the dynamics; otherwise, their values are derived from the α, β formulae.
This means that the α, β rates specified in the model may well be inconsistent with τ and/or x_∞! To prevent this, avoid the `<gateHHratesTauInf>` tag and prefer the `<gateHHrates>` and `<gateHHtauInf>` tags to describe the dynamics.

Fractional HH gate dynamics

Sometimes, gates are modelled as individual and independent *sub-populations*, where each "sub-gate" has Hodgkin-Huxley dynamics. Since the subpopulations are independent, they contribute *additively* (not multiplicatively!) to the total activation of the gate. The NeuroML tag to describe this is `<gateFractional>`[108]. It contains multiple `<subGate>`s, each with attributes `id`(entifier) and `fractionalConductance` (weight by which they contribute to the `<gateFractional>`'s activation). Each `<subGate>`is then modelled with the same attributes and semantics as a Hodgkin-Huxley-style `<gateHHtauInf>` (page 47).

Note: It may make more sense if the total of a `<gateFractional>`'s `<subgate>`s adds up to 1.

Multi-state Markov gate dynamics

Another way to model a gate population is to assume *more than two* possible (exclusive) states for each individual gate. In that case the state of a gate population can be described by the fractions of gates per state they're in — but same as with Hodgkin-Huxley modelling, there is one degree of freedom *less* since all the fractions per state must add up to 1. The various pathways for gates to (statistically) move between states are then laid out as a general *kinetic scheme* (or continuous-time Markov process[109]).
Tip: Non-Markov kinetic schemes can also be modelled just fine, by specifying *non-stationary* transition rates with LEMS (page 79).

[103] `https://docs.neuroml.org/Userdocs/Schemas/Channels.html#gatehhrates`
[104] `https://docs.neuroml.org/Userdocs/Schemas/Channels.html#gatehhtauinf`
[105] `https://docs.neuroml.org/Userdocs/Schemas/Channels.html#gatehhratestau`
[106] `https://docs.neuroml.org/Userdocs/Schemas/Channels.html#gatehhratesinf`
[107] `https://docs.neuroml.org/Userdocs/Schemas/Channels.html#gatehhratestauinf`
[108] `https://docs.neuroml.org/Userdocs/Schemas/Channels.html#gatefractional`
[109] `https://en.wikipedia.org/wiki/Continuous-time_Markov_process`

The NeuroML tag to describe this is <gateKS>[110]. It contains multiple <openState>s <closedState>s, each with an identifier name attribute. It also contains multiple transition paths between these states, as the tags:

- <forwardTransition>, which contains a <rate> tag that has same attributes as the <forwardRate> in HH-style gate dynamics (page 47).
- <reverseTransition>, which contains a <rate> tag with same attributes as <reverseRate>.
- <TauInfTransition>, which expresses the rates in a different way (that would really make sense for HH dynamics). It contains the <timeCourse> and <steadyState> tags with the same semantics as in <gateHHtauInf>, but the true transition time constant and dynamic equilibrium between from and to also depend on all the other (direct and indirect) transition pathways as well. Use the τ, x_∞ to α, β conversion formulae (page 45) to retrieve the forward and reverse rates that <TauInfTransition> stands for.

Each Transition must specify from which state it goes to which as attributes. For each <forwardTransition> between two states there should be a <reverseTransition> and vice versa, and multiple transitions between a pair of states are not allowed (sum then up instead).

- Even if a transition goes in one direction only, this should be made explicit by specifying a <reverseTransition> with a <rate> of constant 0.

Examples

Take a look at the NeuroML description[111] of the following gating kinetic scheme from Hirschberg et al. 1998[112], as shown in figure 10[113] of this paper:

$$\begin{array}{ccccc} C_1 & \rightleftharpoons C_2 & \rightleftharpoons C_3 & \rightleftharpoons C_4 \\ & & \downdownarrows & \downdownarrows \\ & & O_1 & O_2 \end{array}$$

Compare and contrast with its description[114] as a MOD file for NEURON.

Since the rate functions aren't part of the pre-defined rate formulae, they have been described with LEMS. Note that all parameters are repeatedly defined for all rates, even though each rate actually cares for only one specific rate constant.

- The interested reader may thus slim down the definitions of the LEMS rates, and even further, whittle the <ComponentType>s down to one calcium-dependent and one fixed transition rate, as seen with the second example below.

Another ion channel modelled[115] with a <gateKS> is from Carter et al. 2012[116] figure 7a[117], with 12 separate states that arise from 6 *mechanical* states that in sequence open the ion channel, and a potential *inactivation* sub-state for each mechanical state. Note that all these rates follow one of just two base formulae, with only different parameter values changing — that's *much neater!*

Exercise

Convert the Markov ion channel of from the MOD file description[118] (or other cases of the kinetic model[119] for NEURON, to a NeuroML description. Which description is easier to read and modify, in which use cases? Use the <q10Settings> tag (page 49) to model the identical effect of temperature on all transitions.

[110] https://docs.neuroml.org/Userdocs/Schemas/Channels.html#gateks
[111] https://github.com/OpenSourceBrain/SolinasEtAl-GolgiCell/blob/v0.1/neuroConstruct/generatedNeuroML2/KAHP_CML.channel.nml
[112] https://doi.org/10.1085/jgp.111.4.565
[113] https://pmc.ncbi.nlm.nih.gov/articles/PMC2217120/figure/F10/
[114] https://modeldb.science/112685?tab=2&file=Golgi_cell/Golgi_SK2.mod
[115] https://github.com/OpenSourceBrain/AllenInstituteNeuroML/blob/v0.1/CellTypesDatabase/models/NeuroML2/NaV.channel.nml
[116] https://doi.org/10.1016/j.neuron.2012.08.033
[117] https://pmc.ncbi.nlm.nih.gov/articles/PMC3460524/figure/F7/
[118] https://modeldb.science/230137?tab=2&file=SinKin_ModelDB/Nav13_a.mod
[119] https://modeldb.science/230137

Instantaneous gate dynamics

Finally, there are some ion channel gates that open or close *instantaneously* (or quick enough, in any case) in response to their surroundings. Hence they don't carry an intrinsic *state variable* that affects their evolution moving forward; they are *stateless*.

The behaviour of such gates is described in NeuroML as a `<gateHHInstantaneous>`[120] tag; this contains a `<steadyState>` tag with the `type` attribute that selects the dimensionless formula (page 45) for the instantaneous activation fraction, and its corresponding parameters as attributes.

Gate rate scaling

As chemical reactions often do, an ion channel gate's transitions may be accelerated or slowed down by the local temperature (or possibly other factors in the model). For convenience, NeuroML allows adding a `<q10Settings>` tag on any non-static type of `<gate>` (or `<subGate>`), which adds a temperature-dependence Q_{10} factor[121] to speed up the gate by (effectively scaling τ).
The attributes of this tag may be either:

- `q10Factor` (unitless, per 10°C) and `experimentalTemp` that the gate's recorded dynamics refer to (i.e. the temperature at which the gate behaves the same as without the `q10Settings` mentioned).
- or `fixedQ10` (unitless scaling factor) for a reaction speed that actually stays the same within the relevant temperature zone of the model (or was calculated outside the model's description, e.g. when generating a model file and having pre-calculated rates).

Conductance scaling

As we saw above (page 49), the open-close dynamics of ion channel gates may be affected by temperature and NeuroML offers a way to model that. Another effect of temperature that's often encountered is that the *conductance* of certain ion channels may be altered as well. To capture this effect, NeuroML offers the `<q10ConductanceScaling>` tag, which can be inserted to `<ionChannel>`s and has attributes `q10Factor` and `experimentalTemp`, with the same meaning as for temperature-affected gate transition rates.

Refer also to how custom temperature-dependence models that may be affected by more parameters can be modelled with LEMS (page 80).

More dynamics models

Finally, there may be ion channel models (especially ones proposed for the first time) that don't fully fall within one of the predefined formulations. If their dynamics allow, these can still be modelled as custom LEMS components (page 81), and also using EDEN's extensions to NeuroML (page 85) if necessary.

[120] `https://docs.neuroml.org/Userdocs/Schemas/Channels.html#gatehhinstantaneous`
[121] `https://en.wikipedia.org/wiki/Q10_(temperature_coefficient)`

Ion channel distributions

After designing an (aggregate) ion channel mechanism (page 44), it's time to place *instances* of this mechanism on cells. This is done in NeuroML through *distribution* tags that designate a *type* of ion channel, an *extent* over the cell, and *parameters* of the distribution, such as `condDensity` and `erev`(ersal). (Much like the tags for electrical biophysical properties (page 43) do.)

There are many different distribution types in NeuroML (refer to the NeuroML reference[122] and search the page for `channelDensity` and `channelPopulation`); what they all have in common is the following attributes:

- `ionChannel`, which points to the ideal ion channel mechanism to replicate all over the place (or simply scale by `conductanceDensity * compartment area` for the usual macroscopic models).
- `ion` (optionally) which designates the specific ion species to be pooled. May be `non_specific` when referring to an "`<ionChannel>`" lacking the `ion` property. Useful then there are e.g. two separate "pools" of calcium which activate certain mechanisms differently for some reason.
- `segment` or `segmentGroup` (either optionally) which designates the part of the cell to add the ion channels to; `all` if neither attribute is present.

Note: for `<channelPopulation>`s which involve a specific `number` of ion channels, the per-individual `conductance` must have been specified for the `<ionChannel>`.

Caution

If two distributions with the same `<ionChannel>` name are defined for overlapping `<segmentGroup>`s, it is ambiguous whether the distributions should *both* be instantiated, or one should replace the other, in the zone of overlap. Make sure that distributions of the "same" ion channel mechanism do not overlap, to to avoid ambiguity!

Non-uniform ion channel distributions in NeuroML

The previous way to define distributions of mechanisms specifies, for each parameter, the *same value* all over the distribution; but some times we need more that that. Of course, one could always use a fine-grained set of tiny `<segmentGroup>`s and define a different distribution for each one of them for each non-uniform distribution, but this is cumbersome to write, read, store and use. NeuroML offers some provisions for non-uniformity through the `<channelDensityNonUniform>`, `<channelDensityNonUniformNernst>` and `<channelDensityNonUniformGHK>` tags for ion channel distributions.

Tip

For a *fully general* (albeit EDEN-specific) way to control the variability of *all* mechanisms on spatially-modelled cells, refer to the relevant usage example (page 99) of the `<EdenCustomSetup>` extension.

Inhomogeneous value in the `channelDensity`

Instead of a `segment` or `segmentGroup` attribute, the aforementioned tags contain one or more `<variableParameter>` tags that define variability over `segmentGroups`, each through a relevant `<inhomogeneousParameter>` in the `<segmentGroup>` and `<inhomogeneousValue>` mathematical function of said parameter. The `<variableParameter>` tags have the following attributes:

- `segmentGroup`: to distribute ion channels over, and search for `<inhomogeneousParameter>`s within;

[122] `https://docs.neuroml.org/Userdocs/Schemas/Cells.html#channeldensity`

- `parameter`: The *biophysical attribute* to vary over the ion channel's distribution; this can be `condDensity` for `<channelDensityNonUniform>` and `<channelDensityNonUniformNernst>`, and `permeability` for `<channelDensityNonUniformGHK>`.

And inside each `<variableParameter>` tag, there is a `<inhomogeneousValue>` tag with attributes:

- `inhomogeneousParameter`: The `id` of the `<inhomogeneousParameter>` specified in the `<segmentGroup>`
- `value`: A *dimensionless* LEMS expression that involves the `<inhomogeneousParameter>` by its `variable` name. **Note:** The numerical value of the expression stands for SI units, such as *mhos per square meter*, S/m^2 for `condDensity` and *metres per second*, m/s for `permeability`.

Inhomogeneous parameters in the `morphology`

To support mechanism distributions with non-uniform parameters, non-uniform dimensionless *parameters* can be defined over a `<morphology>` by inserting `<inhomogeneousParameter>` tags in it. These have the following attributes:

- `id`: Must be unique within the `<segmentGroup>`.
- `variable`: The actual variable name to be used in LEMS expressions referencing the `<inhomogeneousParameter>`.
- `metric`: The name of the metric to give the variable values by, over the extent of the `<segmentGroup>`. The only permitted metric for now is `Path Length from root`. **Note**: Unlike most LEMS quantities, this variable is provided *dimensionless* and its value is in *microns* (if not made relative with `normalizationEnd`).

The following optional tags inside the `<inhomogeneousParameter>` apply transformations to the `metric` variable:

- `<proximal translationStart="<value>"/>`: *Offset* the metric so that the minimum is `value` (typically `0`).
- `<distal normalizationEnd="<value>"/>`: *Divide* the metric my its maximum and then multiply by `value` (typically `1`), so that the variable is now a relative, *dimensionless* measure.

3.4.3 Adding concentration models near the membrane

Another type of local "mechanism" that can be present on (parts of) the neuron is *concentrations of chemical species* (typically ions) but near the cell's membrane. The presence (and relevance) of such concentrations can be expressed in NeuroML through `<species>` tags inside `<intracellularProperties>` (i.e. along with axial `<resistivity>` tags).
Each `<species>` tag has the following attributes:

- `id`: Name of the "distribution" of the concentration model. Useful when different concentration models are used for the same species.
- `segment` or `segmentGroup` which designates where to add the concentration model to; `all` if neither attribute is present.
- `ion`: Name of the *substance* whose concentration is being tracked (in theory, it shouldn't be just ions though).
- `initialConcentration`: The concentration of the chemical species near the *inner* side of the membrane, when the simulation starts.
- `initialExtConcentration`: The concentration near the *outer* side of the membrane, when the simulation starts. This usually is kept steady, to model a well mixed buffer when neurons are isolated on a dish; but new `<concentrationModels>` can always be have more interesting dynamics.

The concentration model, i.e. relationship between incoming flux and local concentration over time for a chemical species, is described by NeuroML tags at the top level of the NeuroML files. NeuroML offers the following types of concentration models outside the box:

- `<decayingPoolConcentrationModel>`[123], for modelling a thin shell of a certain `shellThickness` near the compartment's membrane, and a `decayConstant` of time over which the concentration converges to `restingConc`;
- `<fixedFactorConcentrationModel>`[124], for similar dynamics, but with a fixed linear relationship between the *incoming current density* and the resulting *rate of change* of the local concentration (instead of considering the Faraday constant and `shellVolume`, as the previous model does).

Same as with most NeuroML mechanisms, novel concentration models can also be modelled with LEMS (page 81).

Note

The total *species influx* arriving to the local concentration model, is the sum of all `<ionChannel>` distributions transporting the same chemical species.

3.4.4 Attaching probes and synapses on a spatially-modelled cell

As we saw in the Introduction section (page 19), mechanisms can also be individually *attached* to cells to complete the models, through `<projection>`s and `<inputList>`s in the `<network>`.

The exact spot (page 38) to place these mechanisms on the cell is specified for:

- `<input>`s in the `<inputList>`s, as `<segment>` and `<fractionAlong>`
- `<connection>`s in the `<projection>`s, as `{pre,post}Segment` and `{pre,post}FractionAlong` attributes (one pair for each of the `pre`- and `post`-synaptic sites).

3.4.5 Recap

These are the things which make up a *spatially-detailed biophysical* neuron model: if you've managed to read this far, you now know enough to read models of neurons, as well as make your own.

Armed with this knowledge, look again at the `.cell.nml` files shown at the start (page 34). If you see anything that's still not clear now, ask about it on the NeuroML discussion channels[125] and tell the guide's authors (page 3) to include it in this article.

3.5 Example: Modelling and simulating a stretch of neural cable

As a first tutorial, we'll implement in NeuroML and play with the simplest spatially-detailed neuron model: an unbranched *section of neural cable* (that could be part of the e.g. axon), with a spike that propagates along it following the Hodgkin-Huxley type, Na and K ion channel dynamics.

[123] `https://docs.neuroml.org/Userdocs/Schemas/Cells.html#decayingpoolconcentrationmodel`
[124] `https://docs.neuroml.org/Userdocs/Schemas/Cells.html#fixedfactorconcentrationmodel`
[125] `https://docs.neuroml.org/NeuroMLOrg/CommunicationChannels.html`

3.5.1 Modelling the cell's morphology

The morphology of a straight cylinder is rather simple. We'll define only one cylindrical `<segment>` and make use of `neuroLexId="sao864921383"` and `numberInternalDivisions` to have it chopped into finer compartments.

- Note that, as explained above (page 40), the same morphology could be made by constructing individual segments and letting the default discretisation of 1 compartment per segment take hold.

```
cable_length_microns = 200
cable_diameter_microns = 1
cable_compartments = 50
cable_morph_file = f'''<neuroml>
<morphology id="Subdivided_Morph">
    <segment id ="0" name="Axon_0">
        <proximal x="0" y="0" z="0" diameter="1"/>
        <distal x="{cable_length_microns}" y="0" z="0" diameter="{cable_diameter_
↪microns}"/>
    </segment>
    <segmentGroup id="axon" neuroLexId="sao864921383">
        <property tag="numberInternalDivisions" value="{cable_compartments}"/>
        <member segment="0"/>
    </segmentGroup>
</morphology>
</neuroml>'''
# print(cable_morph_file)
with open('Subdivided_Morph.nml', 'wt') as f: f.write(cable_morph_file)
```

3.5.2 Modelling the membrane mechanisms

Next, we'll add the ion channels from the famous Hodgkin-Huxley model, as also seen on the official NeuroML guide[126]. Note that more than by typing, NeuroML files can also be created with `pynml` as shown in the linked example, and any other suitable method.

We'll make one `.channel.nml` file for each independent trans-membrane mechanism.

Sodium channel

```
%%writefile HH_Sodium.channel.nml
<neuroml>
<ionChannel id="na_chan" type="ionChannelHH" conductance="10pS" species="na">

    <gateHHrates id="m" instances="3">
        <q10Settings type="q10ExpTemp" q10Factor="3" experimentalTemp="6.3 degC"/>
        <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">
        <q10Settings type="q10ExpTemp" q10Factor="3" experimentalTemp="6.3 degC"/>
        <forwardRate type="HHExpRate" rate="0.07per_ms" midpoint="-65mV" scale="-
↪20mV"/>
        <reverseRate type="HHSigmoidRate" rate="1per_ms" midpoint="-35mV" scale=
↪"10mV"/>
    </gateHHrates>
```

(continues on next page)

[126] https://docs.neuroml.org/Userdocs/SingleCompartmentHHExample.html#declaring-ion-channels

(continued from previous page)

```
</ionChannel>
</neuroml>
```

```
Writing HH_Sodium.channel.nml
```

Potassium channel

```
%%writefile HH_Potassium.channel.nml
<neuroml>
<ionChannel id="k_chan" type="ionChannelHH" conductance="10pS" species="k">

    <gateHHrates id="n" instances="4">
        <q10Settings type="q10ExpTemp" q10Factor="3" experimentalTemp="6.3 degC"/>
        <forwardRate type="HHExpLinearRate" rate="0.1per_ms" midpoint="-55mV"␣
↪scale="10mV"/>
        <reverseRate type="HHExpRate" rate="0.125per_ms" midpoint="-65mV" scale="-
↪80mV"/>
    </gateHHrates>

</ionChannel>
</neuroml>
```

```
Writing HH_Potassium.channel.nml
```

Passive membrane leak

Note that we only need the `conductance` attribute defined if we're using distributions of a *counted number* of channels, such as `<channelPopulation>`[127].

```
%%writefile Passive_leak.channel.nml
<neuroml>
<ionChannelHH id="passiveChan">
            <notes>Leak conductance</notes>
</ionChannelHH>
</neuroml>
```

```
Writing Passive_leak.channel.nml
```

3.5.3 Modelling the biophysics distributions

Then, we'll apply the ion channels over the whole cable, with `erev`, `condDensity` and other biophysical parameters again as per the example in the NeuroML guide.

```
%%writefile Subdivided_Cable.cell.nml
<neuroml>
<!-- Get the previous definitions -->
<include href="Subdivided_Morph.nml"/>
<include href="HH_Sodium.channel.nml"/>
<include href="HH_Potassium.channel.nml"/>
<include href="Passive_leak.channel.nml"/>

<!-- Set the biophysics -->
<biophysicalProperties id="bioPhys_Passive">
    <membraneProperties>
        <channelDensity id="na_all" ionChannel="na_chan"     ion="na"          ␣
```

(continues on next page)

[127] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#channelpopulation

(continued from previous page)

```
↪condDensity="0.12   S_per_cm2" erev="+50.0 mV" />
        <channelDensity id="k_all"  ionChannel="k_chan"      ion="k"           ␣
↪condDensity="0.036  S_per_cm2" erev="-77.0 mV" />
        <channelDensity id="leak"   ionChannel="passiveChan" ion="non_specific"␣
↪condDensity="0.0003 S_per_cm2" erev="-54.3 mV" />
        <spikeThresh value="0 mV"/>
        <specificCapacitance value="1.0 uF_per_cm2"/>
        <initMembPotential value="-65mV" />
    </membraneProperties>
    <intracellularProperties>
        <resistivity value="1 kohm_cm"/>
    </intracellularProperties>
</biophysicalProperties>

<!-- And compose into a cell type -->
<cell id="Subdivided_Cable" morphology="Subdivided_Morph" biophysicalProperties=
↪"bioPhys_Passive"/>
</neuroml>
```

```
Writing Subdivided_Cable.cell.nml
```

3.5.4 Modelling the experimental rig

We'll put a current clamp on some $3/10$ along the cable, to see how the induced spike will propagate along the cable towards both directions.

```
cable_pop_name='SingleAxon'
model_file = f'''<neuroml>
    <include href="Subdivided_Cable.cell.nml"/>

    <pulseGenerator id="pulseGen1" delay="5ms" duration="200ms" amplitude="0.1nA"/>
    <network id="Net">

        <population id="{cable_pop_name}" component="Subdivided_Cable" size="1" />
        <inputList id="stimInput_Subdivided_1" component="pulseGen1" population="
↪{cable_pop_name}">
            <input id="0" target="../{cable_pop_name}[0]" segmentId="0"␣
↪fractionAlong="0.3" destination="synapses"/>
        </inputList>

    </network>
</neuroml>'''
# print(model_file)
with open('Model_LonelyAxon.nml', 'wt') as f: f.write(model_file)
```

3.5.5 Recording over the cell's full extent

We want to record the state of the cable throughout, over time. We'll use EDEN's `explain_cell` and `GetLemsLocatorsForCell` helper routines (page 241) to tell us which *spots on the neuron* to record, so that all *compartments* are covered.

> **Note**
>
> Because official NeuroML allows recording on `<segment>`s but only on the middle of each `<segment>` (when there's only one in our stylised morphology), we'll have to use extended LEMS path (page 87) syntax, which is supported by EDEN but not official NeuroML.

```
import eden_simulator
cells_info = eden_simulator.experimental.explain_cell('Model_LonelyAxon.nml')
cell_info = cells_info['Subdivided_Cable']
```

```
comp_locators = eden_simulator.experimental.GetLemsLocatorsForCell(cell_info)
cell_voltage_paths = [ f'{cable_pop_name}[0]/{loc}/v' for loc in comp_locators ]
tabline = '\n        '
sim_file = f'''<Lems>
<Include href="Model_LonelyAxon.nml"/>
<Simulation id="MySim" length="100 ms" step="50 us" target="Net">
    <OutputFile id="MyOutFile" fileName="results.gen.txt">
        {tabline.join([ f'<OutputColumn id="v_0_{loc}"  quantity= "{path}"/>' for␣
↪loc, path in zip(comp_locators, cell_voltage_paths)])}
    </OutputFile>
</Simulation>
<Target component="MySim"/></Lems>'''
# print(sim_file)
with open('Sim_LonelyAxon.xml', 'wt') as f: f.write(sim_file)
```

3.5.6 Running the simulation and displaying results

```
results = eden_simulator.runEden('Sim_LonelyAxon.xml')
```

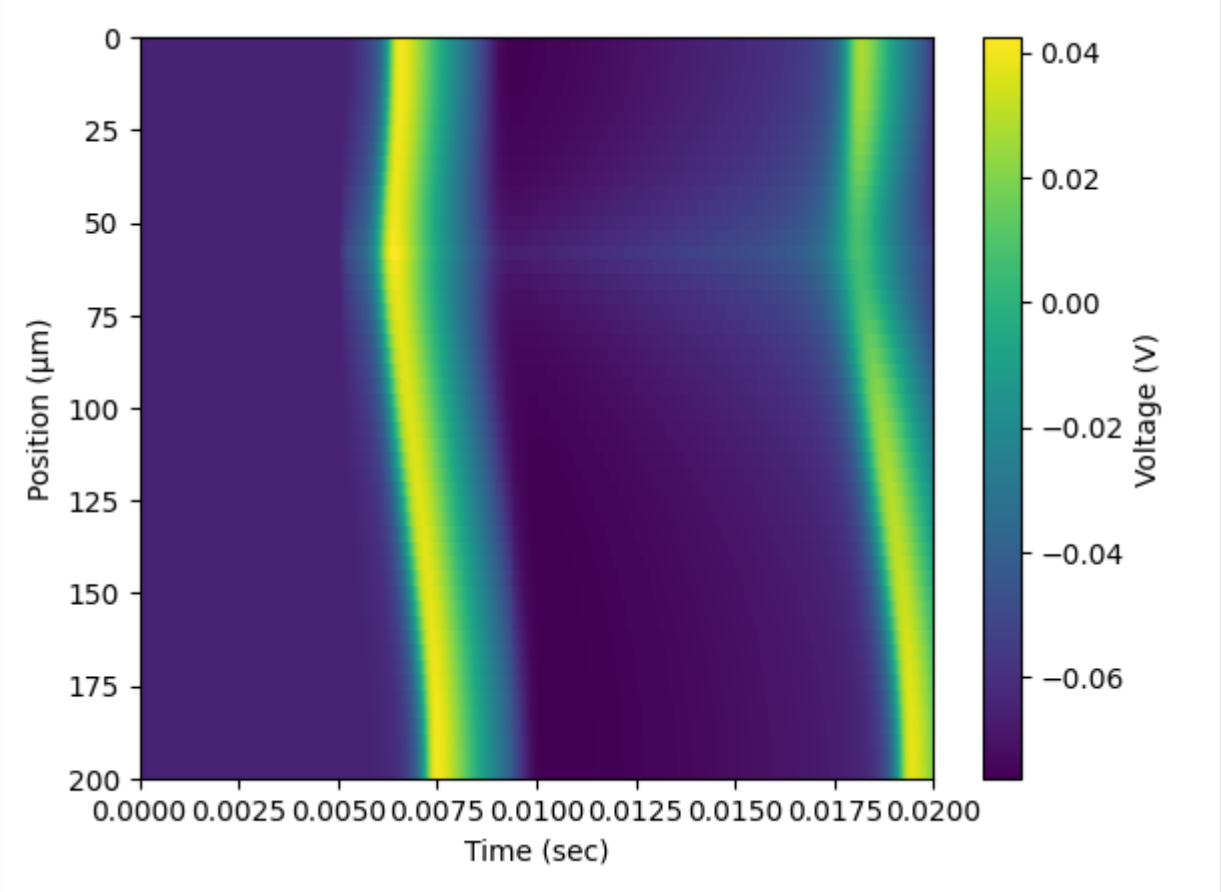
```
import numpy as np
rec_time_axis_sec = results['t']
neuron_waveforms = np.array([results[path] for path in cell_voltage_paths])
neuron_waveforms.shape
```

```
(50, 2001)
```

Since it's a straight unbranched cable, we can intuitively assume a single spatial axis `x` along it, and plot a 2-D function over `x` and `time`.

When a neuron is more like a sprawling *tree* (or bush), we can still record along a certain linear section, but otherwise we may have to use a more *skeuomorphic* display.

```
from matplotlib import pyplot as plt
# help(plt.imshow)
im = plt.imshow(neuron_waveforms,
    extent=[ *rec_time_axis_sec[[0,-1]], cable_length_microns, 0 ],
           aspect='auto',interpolation='None')
cbar = plt.colorbar(im); cbar.set_label('Voltage (V)')
plt.xlabel('Time (sec)');plt.ylabel('Position (µm)')
plt.xlim((0, 0.02));
```



Finally, let's visualise the "real" thing *in space*, painted with (of course) false colour that represents membrane voltage.
(And as a first step to that, reduce the amount of displayed data to a practical frame rate.)

```
from eden_simulator.display.animation import subsample_trajectories
_, anim_axis, sampled_time_axis_sec, (sampled_voltage,) = subsample_
↪trajectories(rec_time_axis_sec, [neuron_waveforms.T])
print("Animation duration: %d frames, %.3f sec" % (len(anim_axis), anim_axis[-1]))
anim_axis
```

```
Animation duration: 1001 frames, 33.333 sec
```

```
array([0.00000000e+00, 3.33333333e-02, 6.66666667e-02, …,
       3.32666667e+01, 3.33000000e+01, 3.33333333e+01], shape=(1001,))
```

Let's see the painted neuron that you might have already visualised in your mind, while modelling it. We'll use K3D[128] for an animated 3-D display.

```
import k3d; from eden_simulator.display.spatial.k3d import Plot, plot_neuron
plot = Cable_plot = Plot()
plot += plot_neuron(cell_info, sampled_voltage, time_axis_sec=anim_axis, color_map=
↪'jet',);
# Add a nice text box to inform us about the time elapsed in the simulation.
k3d_label_dict = { str(real_time): f't = {sim_time*1000:.1f} ~ ms' for (real_time,␣
↪sim_time) in zip(anim_axis, sampled_time_axis_sec)  }
plt_label = k3d.text2d(k3d_label_dict, (0.,0.)); plot += plt_label
```

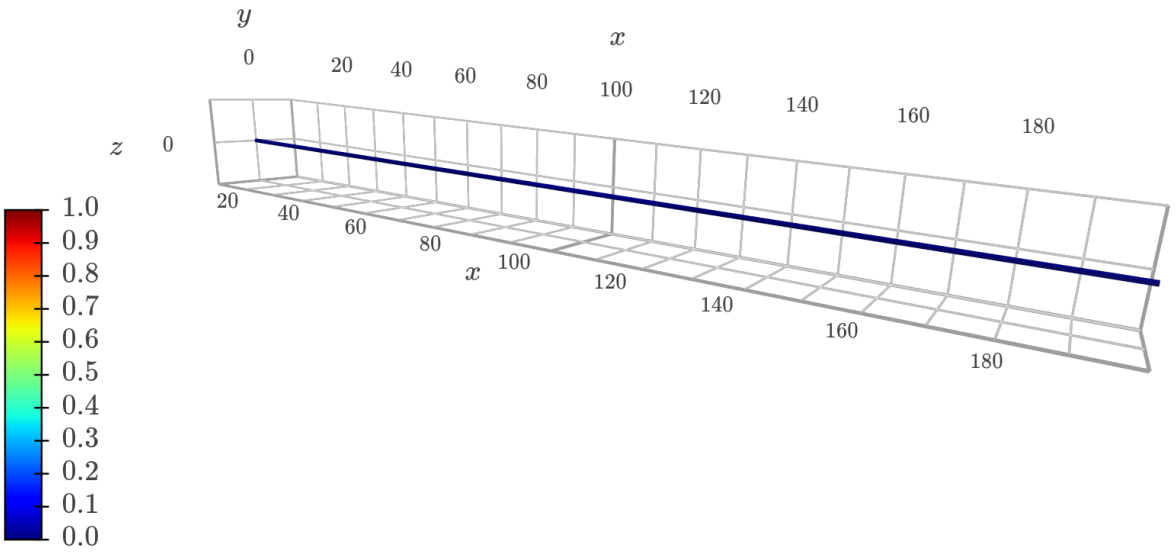
(continues on next page)

[128] http://k3d-jupyter.org

(continued from previous page)

```
# And display as a HTML inset.
from IPython.display import HTML, display
plot.snapshot_type = 'inline'; display(HTML(plot.get_snapshot()))
```

```
<IPython.core.display.HTML object>
```



Exercise: When the cable is 200 µm long, we observe that the action potential propagates slower near its starting point, than away of it. (Enlarge the time axis or focus on a small part of it to see this clearly.)

- Grow the cable to 1~2 mm (increase the `numberInternalDivisions` if necessary for an accurate result) and look again at membrane voltage over time and space. How does the picture change? Plot the subset, show it beside the original shorter cable so that the clamped points are aligned.
- Why is AP propagation so sped up near the ends of the cable? Consider the system's dynamics.
- If an endless cable is to be modelled (to study small-scale propagation dynamics and such), how would you alter the model, to minimise the effect of the cable's finite length? Look up how this is handled with other types of cables.

Exercise: Explore how the discretisation's fineness affects the result. What happens when there are too few (or too many) compartments? What happens as the spatial resolution *starts getting* too low or too high? Compare and contrast with inappropriate values for the simulation's *time* step.

- **Tip**: Wrap the model-to-2D-plot procedure in a routine instead of running all the cells separately.

3.6 More resources on spatially-detailed modelling

3.6.1 Multi-compartment modelling in other simulators

The concept of spatially-detailed neurons that are modelled as coupled compartments is also supported by other neuro-simulators, and their platform-specific modelling formats. Readers who have learned (or are learning) to use another simulator, or want to compare and contrast the implementations of concepts and see more usage examples, may refer to:

- Chapter 5 of the NEURON book (and the whole book and user's guide[129] for more details);

[129] `https://nrn.readthedocs.io/`

- Arbor's implementation of the concept, much in line with NEURON and NeuroML, as "cable" cells[130];
- Chapters 2 and 5 of the Book of Genesis[131];
- BRIAN's multi-compartment capability[132] — note that the compartment's dynamics equations are all written *explicitly* (i.e. same as with point neurons), rather than being composed of the various distributed mechanisms.

3.7 Next steps

After learning how to model spatially-detailed neurons and their geometry-based biophysics, it's time to model more neurons, with more mechanisms on them and more effects that control phenomena. Since each model and experiment typically introduces highly-specific effects, mechanisms that capture them can be written by the model authors, as LEMS extensions (page 61).

Following this tutorial, you may also try out the "Building a network" tutorial (page 129), which shows how to use and display a full-featured spatially-detailed cell. As mentioned previously, if a quantity needs to vary among neurons or across their extent for a desired model, EDEN permits full control over variability through <EdenCustomSetup> (page 99).

[130] https://docs.arbor-sim.org/en/latest/concepts/cable_cell.html
[131] http://www.genesis-sim.org/GENESIS/iBoG/iBoGpdf/index.html
[132] https://brian2.readthedocs.io/en/stable/user/multicompartmental.html

CHAPTER

FOUR

EXTENDING NEUROML WITH LEMS COMPONENTS

NeuroML[133] is a rich language for describing neural models. However, the mechanisms in its standard library are not that extensive and to be fair, it would not be possible to anticipate all the possible mechanisms that could be proposed in neuroscience. Hence, NeuroML works together with another modelling language, LEMS[134]. LEMS can describe arbitrary dynamical equations for mechanisms (arbitrary as in, any ODE[135]'s plus instantaneous changes). In a sense, it is similar to the block[136]diagram[137] models seen in, e.g. MathWorks Simulink™[138].

Within NeuroML, LEMS may be used to describe new types of cells, synapses and much more (see below for the full list (page 66)).

This article will introduce *LEMS components*, and how these can be used to create custom neural dynamics. The various NeuroML-related aspects of LEMS components are presented, and then an example is given on how to write custom dynamics for each sort of neural mechanism (neuron, synapse, &c.)

4.1 What is a LEMS component?

A LEMS component is a virtual physical entity (or *mechanism*) that interacts with its environment in ways described my mathematical expressions. Its internal *state* is captured by a number of scalar *variables*, and its interactions with the environment are in terms of *scalar* physical quantities or instantaneous *events* that it *is affected by* or *determines*. It may also include other physical quantities called *parameters* that remain fixed throughout a simulation, but still influence the state's dynamics.

Side note: The scalar quantities can be used to implement *continuous in space* (also known as "density-type") mechanisms, since all models are discretised to run on the computer; refer to the guide's article on spatially detailed cells (page 33), and also the examples (page 227) about modelling fields.

Within its *Requirements* (what comes *in*) and the *Exposures* (what comes *out*), a LEMS component behaves like a black box[139]; this means that it does not otherwise interact with its surroundings. This property is helpful to keep track of what goes where[140] in complex systems.

It is not necessary for a LEMS component to have state variables: a stateless component can determine and `expose` quantities that are a function of the component's `Requirements` at each point in time. This the case for instantaneous ion channel gates, gap junctions and other "static" mechanisms, where the e.g. electrical `conductance` is an important *parameter*.

LEMS components are added to NeuroML files just like NeuroML-standard mechanisms. A tag[141] named after their *type* is added inside the `<neuroml>` but not inside the `<network>`, and the tag's attributes[142] specify the

[133] https://neuroml.org
[134] https://lems.github.io/LEMS/
[135] https://en.wikipedia.org/wiki/Ordinary_differential_equation
[136] https://www.youtube.com/watch?v=IqxJpaKuQGo
[137] https://ocw.mit.edu/courses/15-988-system-dynamics-self-study-fall-1998-spring-1999/1ba2bda320cfc48c71d04cb2cf42409b_building.pdf
[138] https://www.mathworks.com/discovery/block-diagram.html
[139] https://en.wikipedia.org/wiki/Black_box
[140] https://en.wikipedia.org/wiki/Separation_of_concerns
[141] https://en.wikipedia.org/wiki/XML#Tag
[142] https://en.wikipedia.org/wiki/XML#Attribute

parameters that determine its behaviour. Indeed, the NeuroML-standard mechanisms we've used in the previous (page 61) articles (page 33) also have LEMS descriptions[143].

What remains then, is to describe the *types* of the components.

4.2 What is a LEMS `ComponentType`?

Besides `<Component>`, the other core concept of LEMS is `<ComponentType>`. This describes the *external interface* and the *internal structure* of components, and the *mathematical expressions* that govern their behaviour.

Their usefulness comes from the frequent occasion where many types of mechanism can be captured by the same mathematical equations. A famous case is the Izhikevich neuron model[144] where the same equations capture many and much different neural activity patterns: all that changes is the values of some parameters. By describing a `ComponentType` we can thus specify the essential dynamics once, instead of re-defining all the properties and behaviour for each *component instance*. `ComponentTypes` are also very useful when archiving or exchanging NeuroML models, since every user (or use case) may require a different set of parameters for the same general model type.

Thus, the ComponentType is defined *once* and then specialised, *concrete* components are defined as follows:

```
<ComponentType name="MyNewSynapseType">
    ...
    <Dynamics>
        ...
    </Dynamics>
    ...
</ComponentType>

...

<MyNewSynapseType this_parameter="60 ms" that_parameter="37 degC">

    or, equivalently:

<Component type="MyNewSynapseType"
    this_parameter="60 ms" that_parameter="37 degC">
```

Note that the components can indeed be further replicated, as in the case of multiple `<connection>`s in a `<projection>`, or `<input>`s in an `<inputList>`. In case that individual instances need to have different parameters within the same `<inputList>` or `<projection>`, this type of variability can be specified through EDEN's <CustomSetup> (page 89) extension.

Now let's see how to write a new `ComponentType`.

4.2.1 The "components" of a ComponentType

Here is a reference of the LEMS properties that can be used to make NeuroML mechanisms (as far as EDEN is concerned).

- Attribute `name`; that's used to refer to the `ComponentType` and hence must be unique for each.
- Attribute `extends`; This specifies the `ComponentType` as a 'derived'[145] type that *inherits* properties from another, more abstract `ComponentType`. This way it's not necessary to re-write the whole `ComponentType` from scratch. Refer to the NeuroML reference[146] (check sidebar) for the standard LEMS components, and the NML component list below (page 66) for a per-type analysis.

[143] https://github.com/NeuroML/NeuroML2/tree/master/NeuroML2CoreTypes
[144] https://www.izhikevich.org/publications/spikes.pdf
[145] https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)
[146] https://docs.neuroml.org/Userdocs/Schemas/Cells.html

Note: Components should `extend` a relevant base type of neuron, synapse, etc. to be considered by EDEN for said roles.

- Attribute `description`; that's optional and up to taste. It is better than XML comments for explaining what the `ComponentType` does or represents. Note that the other LEMS tags may also accept a `description` attribute as well.
- *Interface* tags:
  - `<Requirement>`: This is a *quantity* that the component *needs* from its surroundings, to determine its own behaviour. For example, an ion channel may open or close depending on membrane potential.
    - Attributes are the `name` identifier (which mush match with an available `<Exposure>`), and the `dimension` (in the sense used in Physics[147]) of said quantity.
  - `<Exposure>`: This is a *quantity* that the component *determines* based on its requirements and internal state. Information about this quantity may be thus *exposed* to related parts of the model. For example, an ion channel may *determine* and *expose* the electrical current flux through it.
    - Attributes are the `name` identifier (which must match with the `<Requirement>`s that dependent components have) and the `dimension` of the exposed quantity.
  - `<EventPort>`: This is a named "port" (or "letter box" for lack of a better word) that components can receive the *occurrence* of discrete events through, or alternatively *emit* discrete events In a sense, it is similar to a `<Requirement>` or `<Exposure>` for discrete events rather than quantities over time. For example, a post-synapse waits for the firing of the pre-synaptic neuron that's a discrete event, and a (site of a) neuron may announce when a "spike" has happened on it. Attributes are the `name` identifier, and the `direction` which can be either `in` (the port *receives* events) or `out` (the port *sends* events).
- *Static attributes* tags:
  - `<Parameter>`: This is a quantity about the component, that remains the same for the *component* throughout the simulation. Its value must be specified as a XML attribute[148] in every declared `<Component>` of this `<ComponentType>`. For example, different neurons may have different `<Parameter>` values, but the values stay the same for each cell. Attributes are `name` and `dimension`.
  - `<Property>`: This is same as the `<Parameter>` tag, *except* there can also be a `defaultValue` attribute that is used *if* the value is not specified in a declared `<Component>`. For example, the `weight` of synapses and input sources is `1` unless otherwise specified.
  - `<Constant>`: This is a quantity that's just immutable. Its value is specific to the `<ComponentType>`; it may not be defined for each instance separately. They are typically used for natural constants such as π, or units that are used in math expressions. But they can also be used to reduce the *memory footprint* of the component, so that more of them can fit in the computer during simulation. Attributes are `name`, `dimension`, and mandatory `value`.
- The `<Dynamics>` tag (or tags), that describes the actual behaviour of the `<Component>`. Its structure will be described in the following sub-section.

The components of `<Dynamics>`

These LEMS properties are used inside the `<Dynamics>` tag of a `<ComponentType>` to define its behaviour:

[147] https://en.wikipedia.org/wiki/Dimension_(physics)
[148] https://en.wikipedia.org/wiki/XML#Attribute

"Dynamic" variables

- `<StateVariable>`: This is a *self-standing quantity* inside the component that may change during the simulation. Its value must be assigned `<OnStart>`, see below.
  - Required attributes are `name` and `dimension`. Optionally the `exposure` attribute can be used to have it be a named `<Exposure>`.
- `<DerivedVariable>`: This is a *dependent quantity* whose value is a function of other quantities. It can't depend directly on indirectly on itself, of course.
  - Required attributes are `name` and either:
    - `value`, which is the mathematical expression that yields the `<DerivedVariable>`'s value;
    - or `select`, which may be the sum of the `<Exposure>`s on the mechanism's `<Children>`. This will be explained further through EDEN's Multiflux (page 121) extension.
- `<ConditionalDerivedVariable>`: This is a `<DerivedVariable>` which may be evaluate as different expressions[149], depending on whether certain mathematical conditions apply.

  Note: It is notably used for ion channel gate equations[150] when the formula can evaluate as `0/0` for certain `Vm`. In these cases, the formula is fixed by instead taking the limit[151] and linearisation, or just visually substituting with the curve's value, near the problematic point.
  - Required attributes are `name` and `dimension`.
  - It also must have one or more `<Case>` tags, that each specify as attributes the applicable `condition` and the associated math expression yielding the `value`.
  - The last case should not have a `condition`; it is the *default* `<Case>`, and its `value` will apply when all other `<Case>`s don't.
  - **Note:** The character < is not allowed[152] in XML attributes (though EDEN's parser may still take it). Consider instead of <, >, <=, >=, the alternative FORTRAN[153] spellings `.lt.`, `.gt.`, `.le.`, `.ge.`.
  - The conditions are evaluated in the provided order: in case multiple `conditions` apply, the first stated `<Case>` takes precedence. But the default `<Case>` comes last (and *should* be stated last). The algorithm goes like: `if <first case> ... else if <second case> ... else <default case>`.
  - Due to how the original LEMS interpreter was made, absent a default `<Case>`, a ConditionalVariable slipping all cases will be `0` *but please don't rely on this!*
- `<DerivedParameter>`: This is a `<DerivedVariable>` which depends only on fixed `<Parameter>`s, `<Property>`s and `<Constants>`, and hence its value remains the same for each component.
  - Required attributes are `name`, `dimension` and `value`.

[149] https://en.wikipedia.org/wiki/Piecewise_function
[150] https://en.wikipedia.org/wiki/Hodgkin-Huxley_model#Voltage-gated_ion_channels
[151] https://en.wikipedia.org/wiki/L'Hôpital's_rule
[152] https://www.w3.org/TR/xml/#syntax
[153] https://fortran-lang.org/learn/quickstart/operators_control_flow/#logical-operators

Dynamic behaviour

- `<TimeDerivative>`: This is the rate of change over time[154] that applies for a `<StateVariable>`.
  - Required attributes are the `variable` the rate is for, and the `value` expression for the rate (its `dimension` should be `(the variable's dimension)/time`).
  - It can be specified once per `<StateVariable>`. If you want the rate to be `Conditional`, then make a `<ConditionalDerivedVariable>` for the rate and use that as the `value`. **Note:** It is not necessary for every StateVariable to have a rate; some variables may be updated only in specific occasions through `<OnCondition>` or `<OnEvent>` tags.
- `<OnStart>`: This tag contains the necessary actions to take when the `<Component>` first starts to exist. That is, assign an initial value for each of its `<StateVariables>`.
  - It contains one or more `<StateAssignment>` tags:
    - Required attributes for each are the `variable` and the `value` expression to be assigned to it.
  - Try to not depend on the sequence these are evaluated in.
- `<OnCondition>`: This tag contains the necessary actions to take when a *mathematical condition* is met. For example, when a LIF[155] <iafCell>[156]'s `v` reaches the firing `threshold`, then the cell *emits* a `spike` and `v`'s value is `reset`.
  - The necessary attribute is the condition to `test`. It may contain the following tags:
  - `<StateAssignment>`: Set a state `variable` to the `value` of a math expression, same as with `<OnStart>`.
  - `<EventOut>`: Emit a spike on the specified output `port`.
- `<OnEvent>`: This tag is similar to `<OnCondition>`, but instead of the `test` expression it waits on an `<EventPort>` (with direction `in`) named by the `port` attribute. When a spike arrives on that `port`, the enclosed actions are taken as with `OnCondition`. For example, when a spike reaches a synapse's `in` port, its neurotransmitter cascade is started or stimulated further.

4.3 How can LEMS components be used in NeuroML models?

As mentioned at the start, NeuroML has some models for all types of mechanisms, but that's hardly ever enough. For our modelling, we may prefer a slightly different model; it could be a model from a recent paper, or a brand new proposed set of dynamics equations. We can make such models in NeuroML by describing them in the LEMS dynamics language.

In practice, this means describing individual mechanisms as LEMS components, and using these components as options for model parts in our NeuroML model; the NeuroML formulation remains in charge of the high-level structure of the model.

This section will present the needed LEMS context, and the method to model in LEMS each type of mechanism that EDEN supports.

For *even more* `<Dynamics>` options and other capabilities to model custom interactions, refer to the "EDEN extensions" section of this user guide (page 85).

[154] https://en.wikipedia.org/wiki/Time_derivative
[155] https://en.wikipedia.org/wiki/Biological_neuron_model#Leaky_integrate-and-fire
[156] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#iafcell

4.3.1 Overview of LEMS mechanisms

EDEN allows LEMS components for all these types of mechanisms:

- "Point" neurons (page 66) without detailed anatomy[157]
- Synapses (page 69), on their pre- and post-synaptic parts
  - AP based (page 70), graded (page 72), or mixed interaction
- Input sources (page 76) of the experimental rig such as current, voltage, and other clamps and electrodes
- Ion channels and their properties:
  - Custom whole-gate (page 80) dynamics
    - Or gate transition rates (page 79) (expressed as α, β rates or, τ, ∞ time to stabilise and steady state) for HH, Markov and instantaneous gates
  - Custom scaling factor (page 80) for ion channel conductance
  - Entire ion channels (page 81) with fully-custom dynamics
- Ion pool (page 81) models

Let's see how to work for each one of them.

4.3.2 Commonly available quantities

Beside the quantities that are provided for each type, there are some common, *intrinsically exposed* quantities that are always available:

- `time`: The virtual time that the simulation has been running for, starting at `0`. Also aliased as `t`.
- `temperature`: It's the same across the simulated `<network>`.

Neuron-local quantities

For mechanisms that are located on a neuron, its biophysical properties nearby are also available:

- `v`: Membrane voltage
- `caConc`: Calcium ion concentration (named `ca`) for the mechanisms that care about it. *Note:* There is also a way to specify an alternate Ca^{2+} species[158] `ca2` which does not affect mechanisms.

4.3.3 Custom point neurons

Base `<ComponentType>`s for this type to `extend` include `<baseCell>`[159], `<baseSpikingCell>`[160], `<baseCellMembPot>`[161], `<baseCellMembPotCap>`[162], `<baseCellMembPotDL>`[163], `<baseIaf>`[164], `<baseIafCapCell>`[165], `<baseCell>`[166], and `<basePyNNCell>`[167], `<basePyNNIaFCell>`[168], `<basePyNNIaFCondCell>`[169].

[157] https://en.wikipedia.org/wiki/Biological_neuron_model#Cable_theory_and_compartmental_models
[158] https://github.com/NeuroML/NeuroML2/blob/v1.9.1/HISTORY.md?plain=1#L170
[159] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#basecell
[160] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#basespikingcell
[161] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#basecellmembpot
[162] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#basecellmembpotcap
[163] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#basecellmembpotdl
[164] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#baseiaf
[165] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#baseiafcapcell
[166] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#basecell
[167] https://docs.neuroml.org/Userdocs/Schemas/PyNN.html#basepynncell
[168] https://docs.neuroml.org/Userdocs/Schemas/PyNN.html#basepynniafcell
[169] https://docs.neuroml.org/Userdocs/Schemas/PyNN.html#basepynniafcondcell

The LEMS *interface* for point neurons is as follows:

Available quantities to have as`<Requirement>`*s:*

- `iSyn`: The total inward current flow, from attached synapses *and* (unlike its name would suggest) input sources.
- `ISyn`: Similar, but used for *dimensionless* current flowing into cells with *dimensionless* state variables (e.g. FitzHugh-Nagumo or the original Izhikevich formulation)

Optional`<Exposure>`*:*

- `v`: The membrane voltage of the cell, needed by synapses and certain input sources.

Optional`<EventPort>`*:*

- `spike` (`direction`: `out`): The one and only (for now) spike output of cells.

Note that neurons does not receive events directly hence they don't have an inward `<EventPort>`; incoming spikes are mediated by synapses instead.
If the model requires that spikes change the *internal state* of a neuron directly, see EDEN's `<WritableRequirement>` extension (page 123).

Example: The Quadratic Integrate-and-Fire neuron model

As we saw in the NeuroML tutorial (page 13), NeuroML includes in its standard component library[170] linear integrate-and fire neurons (as it does AdEx neurons and some classic models).

But we may prefer a higher-quality model for our experiments. Here we'll see how to make a *custom neuron type*, using the Quadratic Integrate-and-Fire[171] model as example.

Its dynamics equations are:

$$C_m \frac{dV_m}{dt} = k\,(V_m - V_{\text{rest}})\,(V_m - V_{\text{crit}}) + i_{\text{syn}}$$

$$V_m > V_{\text{peak}} \Rightarrow V_m := V_{\text{reset}}$$

And this is how it's written in LEMS:

```
%%writefile Custom_Cell_Model.nml
<neuroml>

<!-- Let's make a new ComponentType ! -->
<ComponentType name="QifCell"
    extends="baseCellMembPotCap"
    description="Integrate-and-fire cell with quadratic Vm dynamics">

    <!-- Reminder: The following are inherited from "baseCellMembPotCap" -->
    <!-- <Exposure name="v" dimension="voltage" description="Membrane potential"/>␣
↪--> 
    <!-- <EventPort name="spike" direction="out" description="Spike event"/> -->
    <!-- <Parameter name="C" dimension="capacitance" description="Total␣
↪capacitance of the cell membrane"/> -->
    <!-- <Exposure name="iMemb" dimension="current" description="Total current␣
↪crossing the cell membrane"/> -->
    <!-- And iSyn is provided and added as requirement by EDEN -->
    <!-- <Requirement name="iSyn" dimension="current" description="Total current␣
↪due to synaptic inputs"/> -->
```

(continues on next page)

[170] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#iafcell
[171] https://neuronaldynamics.epfl.ch/online/Ch5.S3.html

(continued from previous page)

```
    <!-- Now let's add our own stuff -->
    <Parameter name="v_rest" dimension="voltage" description="Resting potential"/>
    <Parameter name="v_crit" dimension="voltage" description="Critical potential␣
↪for initiating spike"/>
    <Parameter name="v_peak" dimension="voltage" description="Threshold potential␣
↪to register the spike and reset"/>
    <Parameter name="v_reset" dimension="voltage" description="Reset␣
↪(repolarisation) potential"/>
    <Parameter name="v2_factor" dimension="conductance_per_voltage" description=
↪"Intrinsic rate scaling factor"/>

    <Dynamics>

        <!-- It is a one-dimensional model, it has one state variable -->
        <StateVariable name="v" dimension="voltage" exposure="v"/>

        <!-- Extract iMemb from the rate equation to clear things up,
            and satisfy the inherited Exposure as well -->
        <DerivedVariable name="iMemb" dimension="current" exposure="iMemb"
          value="v2_factor * (v-v_rest) * (v-v_crit) + iSyn"/>

        <!-- Here comes the rate, it's reasonable -->
        <TimeDerivative variable="v" value="iMemb / C"/>

        <OnStart> <!-- Start at resting state -->
            <StateAssignment variable="v" value="v_rest"/>
        </OnStart>

        <OnCondition test="v > v_peak"> <!-- Send spike and reset -->
            <StateAssignment variable="v" value="v_reset"/>
            <EventOut port="spike"/>
        </OnCondition>

    </Dynamics>
</ComponentType>

<!-- Here comes the concrete cell type with parameters ! -->
<QifCell id="MyFirstQif" C="200 pF" v2_factor="0.7 nS_per_mV"
    v_rest="-60 mV" v_crit="-30 mV" v_peak="+30 mV" v_reset="-70mV" />

<!-- Add a current clamp to rouse the cell -->
<pulseGenerator id="A_DC_Clamp" delay="100ms" duration="500ms" amplitude="0.21nA"/>
<!-- Here comes the network -->
<network id="Net" type="networkWithTemperature" temperature="37degC" >
    <!-- Add a population with a single cell -->
    <population id="Pop" component="MyFirstQif" size="1"/>
    <!-- Apply a stim on said cell -->
    <inputList id="Inps" population="Pop" component="A_DC_Clamp">
        <input id="0" target="Pop[0]" destination="synapses"/></inputList>
</network>
</neuroml>
```

```
Writing Custom_Cell_Model.nml
```

And add a `<Simulation>` file, don't forget to record the cell's `vm`:

```
%%writefile LEMS_Custom_Cell_Sim.xml
<Lems>
<include file="Custom_Cell_Model.nml" /> <!-- Remember to check the filename -->
<Simulation id="Sim" length="0.7 s" step="0.1 ms" target="Net" >
    <OutputFile id="MyFirstOutputFile" fileName="results.gen.txt">
        <OutputColumn id="vm" quantity="Pop[0]/v"/>
```

(continues on next page)

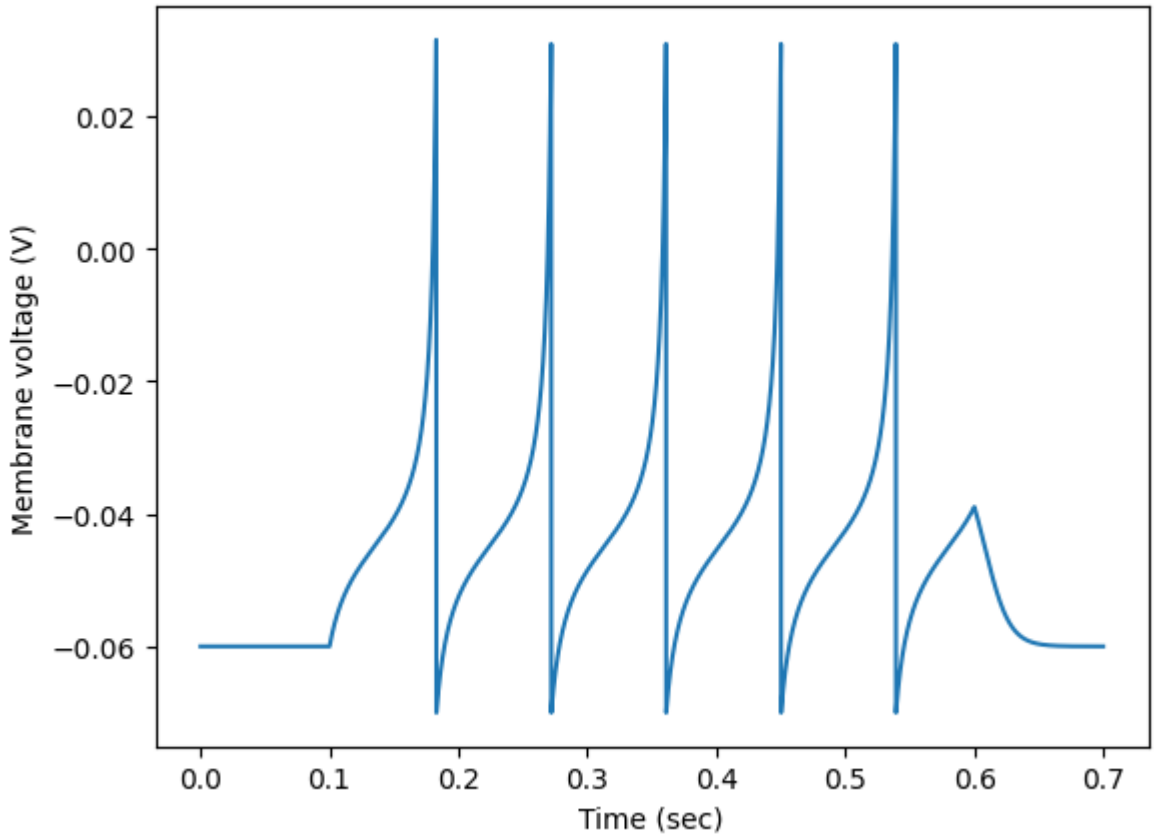
(continued from previous page)

```
    </OutputFile>
</Simulation>
<Target component="Sim"/>
</Lems>
```

```
Writing LEMS_Custom_Cell_Sim.xml
```

Look at the action potentials that this cell generates:

```
from eden_simulator import runEden; from matplotlib import pyplot as plt
results = runEden('LEMS_Custom_Cell_Sim.xml')
plt.plot(results['t'],results['Pop[0]/v'])
plt.xlabel('Time (sec)'); plt.ylabel('Membrane voltage (V)');
```



4.3.4 Custom synapses

Note that every synapse components is attached to a particular neuron, either on the pre- or post-synaptic side. Hence bi-directional synapses are made up of two components.

Base <ComponentType>s for this type to extend include <baseSynapse>[172], <baseVoltageDepSynapse>[173], <baseCurrentBasedSynapse>[174], <baseConductanceBasedSynapse>[175], <baseConductanceBasedSynapseTwo>[176], <basePynnSynapse>[177], and <baseGradedSynapse>[178], <gapJunction>[179].

[172] https://docs.neuroml.org/Userdocs/Schemas/Synapses.html#basesynapse
[173] https://docs.neuroml.org/Userdocs/Schemas/Synapses.html#basevoltagedepsynapse
[174] https://docs.neuroml.org/Userdocs/Schemas/Synapses.html#basecurrentbasedsynapse
[175] https://docs.neuroml.org/Userdocs/Schemas/Synapses.html#baseconductancebasedsynapse
[176] https://docs.neuroml.org/Userdocs/Schemas/Synapses.html#baseconductancebasedsynapsetwo
[177] https://docs.neuroml.org/Userdocs/Schemas/PyNN.html#basepynnsynapse
[178] https://docs.neuroml.org/Userdocs/Schemas/Synapses.html#basegradedsynapse
[179] https://docs.neuroml.org/Userdocs/Schemas/Synapses.html#gapjunction

The LEMS *interface* for synapses is as follows:

Available quantities to have as`<Requirement>`*s:*

- All the intra-cell quantities (page 66), plus:
- `vpeer` (`voltage`): The membrane potential of the non-local cell (other side of the synaptic connection). Available only if exposed by the peer cell.

Required`<Exposure>`*s:*

- `i`: The electrical current that this synapse injects into the cell. Must be of the same `dimension` as the cell's `iSyn`. May also be capital `I` as per the convention for dimensionless current.

Optional`<Exposure>`*s:*

- `g`: The electrical conductance of the mechanism, basically $\frac{di}{dV_m}$. May help with simulation accuracy when available.
- Other flows that a synapse may inject to a cell using the Multiflux extension (page 121).

Optional`<EventPort>`*:*

- `in` (`direction`: `in`): The incoming spikes that (re)activate the synapse.

The following examples show how to make a custom chemical synapse (page 70), gap junction (page 72) and more (page 75).

Example: A time-varying chemical synapse

It is known that sometimes, the effect of chemical synapses is modulated by the concurrent presence of other chemical signals. For example, the spike-induced increment in neurotransmitter release could be more or less depending on external conditions.

Here we'll show an exponential-decay post-synaptic *current* synapse, whose weight is a function of *elapsed simulated*`time`. The generated current decays by a time constant of τ_{syn}, and it is incremented by $I_{\text{base}} \cdot sin(kt)$ on every spike.

Therefore, its dynamics equations are:

$$\frac{dI_{\text{syn}}}{dt} = -\frac{I_{\text{syn}}}{\tau_{\text{syn}}}$$

$$spike\ in \ \Rightarrow \ I_{\text{syn}} \ := I_{\text{syn}} + I_{\text{base}}\ sin(2\pi \frac{time}{T_{weight}})$$

And this is how it's written in LEMS:

```
%%writefile Custom_ChemSyn_Model.nml
<neuroml>

<!-- Let's make a new ComponentType ! -->
<ComponentType name="SinSyn"
    extends="baseCurrentBasedSynapse"
    description="Time-varying exponential current synapse">

    <!-- Reminder: The following are inherited from "baseCurrentBasedSynapse" -->
    <!-- <Exposure name="i" dimension="current" description="Generated current"/> -
↪->
    <!-- <EventPort name="in" direction="in" description="Synapse trigger"/> -->

    <!-- Now let's add our own stuff -->
    <Parameter name="i_base" dimension="current" description="Base peak-current␣
↪level (mefore modulation)"/>
    <Parameter name="tau_syn" dimension="time" description="Time constant of␣
```

(continues on next page)

(continued from previous page)

```
↪exponential decay"/>
    <Parameter name="T_weight" dimension="time" description="Period of time-
↪varying weight"/>

    <!-- Note: we may need some unit constants because physical values are not␣
↪directly allowed in LEMS expressions -->
    <Constant name="pA" dimension="current" value="1 pA"/>
    <Constant name="pi" dimension="none" value="3.14159"/> <!-- Add the␣
↪dimensionless constant π -->

    <!-- Add dependence on "time" ! -->
    <Requirement name="time" dimension="time" description="Time elapsed during the␣
↪simulation"/>

    <Dynamics>

        <!-- It is a one-dimensional model, it has one state variable -->
        <StateVariable name="i" dimension="current" exposure="i"/>
        <!-- And a rate of change -->
        <TimeDerivative variable="i" value="- i / tau_syn"/>

        <OnStart> <!-- Start at idle, no generated current -->
            <StateAssignment variable="i" value="0 * pA"/>
        </OnStart>

        <OnEvent port="in"> <!-- Increase current flow -->
            <StateAssignment variable="i" value="i + i_base * sin(2*pi*time/T_
↪weight)"/>
        </OnEvent>

    </Dynamics>
</ComponentType>

<!-- Here comes the concrete synapse type with parameters ! -->
<SinSyn id="MyCustomChemSynapse" i_base="3 nA" tau_syn="1 msec" T_weight="450 msec
↪" />

<!-- Add a simple cell, and spike source to use our new synapse with -->
<iafCell id="MyFirstCellType" C="200 pF" leakConductance="10 nS" leakReversal="-70␣
↪mV" reset="-70mV" thresh="-50mV" />
<spikeGenerator id="spikeGenRegular" period="70 ms"/>
<!-- Here comes the network -->
<network id="Net" type="networkWithTemperature" temperature="37degC" >
    <!-- Add a single cell and spike source -->
    <population id="Pop" component="MyFirstCellType" size="1"/>
    <population id="Spi" component="spikeGenRegular" size="1" />
    <!-- Connect them with a synapse -->
    <projection id="Pro" presynapticPopulation="Spi" postsynapticPopulation="Pop"␣
↪synapse="MyCustomChemSynapse">
        <connection id="0" preCellId="0" postCellId="0"/> </projection>
</network>
</neuroml>
```

```
Writing Custom_ChemSyn_Model.nml
```

And let's see what happens:

```
%%writefile LEMS_Custom_ChemSyn_Sim.xml
<Lems>
<include file="Custom_ChemSyn_Model.nml" /> <!-- Remember to check the filename -->
<Simulation id="Sim" length="1 s" step="0.1 ms" target="Net" >
    <OutputFile id="MyFirstOutputFile" fileName="results.gen.txt">
```

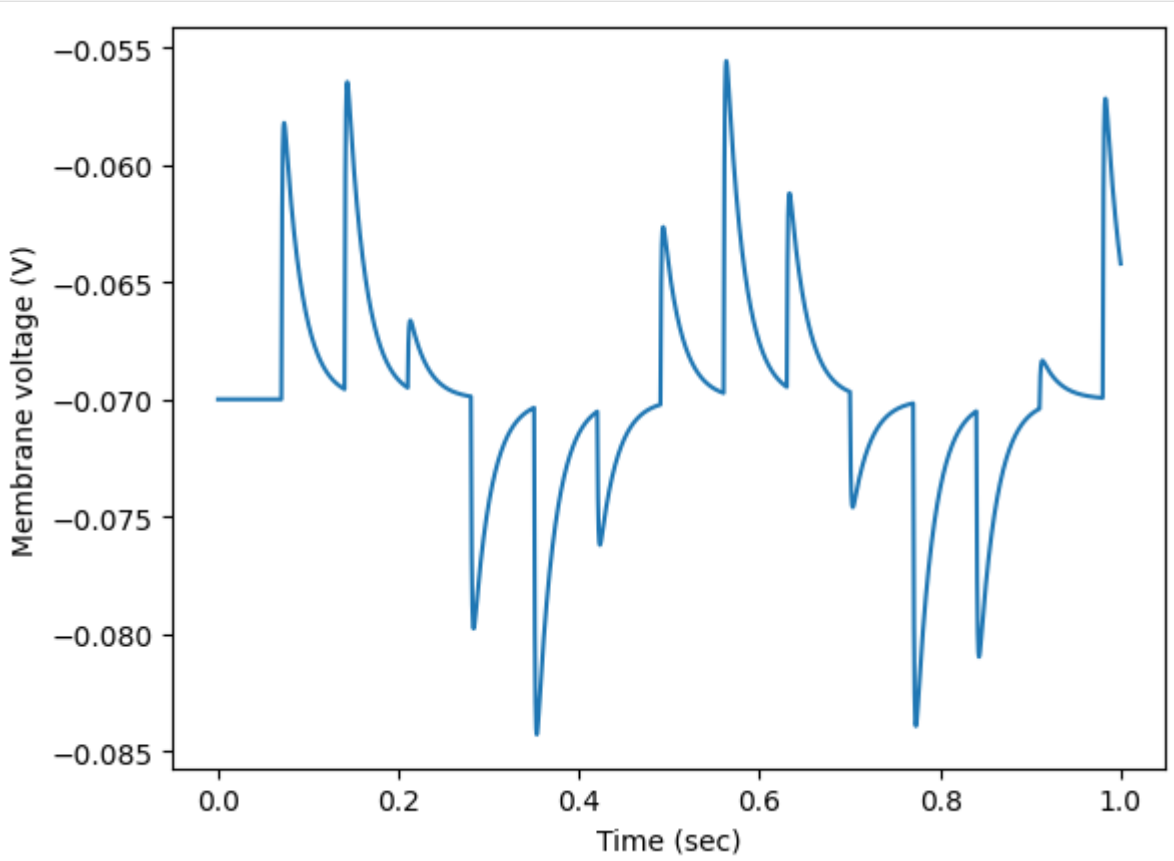
(continues on next page)

(continued from previous page)

```
        <OutputColumn id="vm" quantity="Pop[0]/v"/>
    </OutputFile>
</Simulation>
<Target component="Sim"/>
</Lems>
```

```
Writing LEMS_Custom_ChemSyn_Sim.xml
```

```
from eden_simulator import runEden; from matplotlib import pyplot as plt
results = runEden('LEMS_Custom_ChemSyn_Sim.xml')
plt.plot(results['t'],results['Pop[0]/v'])
plt.xlabel('Time (sec)'); plt.ylabel('Membrane voltage (V)');
```



The same technique may also be used for factors other than `time`, and for other mechanism types (page 66) as well.

Example: A rectifying gap junction

It is also known[180] that the electrical gap junctions coupling cells aren't always ohmic; sometimes the magnitude of current flow depends on polarity! Importantly, this means that the current flow function is also *not symmetric*.

Here we'll model a non-linear gap junction just like that. We'll roughly follow the current vs. voltage curve of a diode with internal resistance[181], based on the HHExpLinearRate[182] formula.

The conductance equation will then be of the form:

$$I = I_0 \left(\frac{x}{1 - e^x} - 1\right), \quad x = \frac{V}{V_0}$$

[180] https://doi.org/10.1016/s0248-4900(02)00022-9
[181] https://en.wikipedia.org/wiki/Diode_modelling
[182] https://github.com/NeuroML/NeuroML2/blob/v1.9.1/NeuroML2CoreTypes/Channels.xml#L53

for I the current from the anode to the cathode, V the voltage between cathode and anode respectively.

And this is how it's written in LEMS:

```
%%writefile Custom_GradSyn_Model.nml
<neuroml>

<!-- Let's make a new ComponentType ! -->
<ComponentType name="GapDiode_Cathode"
    extends="baseGradedSynapse"
    description="Non-symmetrical gap junction, cathode half">

    <!-- Reminder: The following are inherited from "baseGradedSynapse" -->
    <!-- <Exposure name="i" dimension="current" description="Generated current"/> -
↪->
    <!-- And vpeer is provided and added as requirement by EDEN -->
    <!-- <Requirement name="vpeer" dimension="voltage" description="Potential of␣
↪peer cell"/> -->

    <!-- Now let's add our own stuff -->
    <Parameter name="I_rev_max" dimension="current" description="Asymptotic max␣
↪reverse current"/>
    <Parameter name="forward_conductance" dimension="conductance" description=
↪"Marginal (ie small-signal) conductance when fwd polarised"/>

    <Dynamics>
        <!-- It is a *static* model, it has no state variables ! -->
        <!-- Use a derived variable to simplify the formula -->
        <DerivedVariable name="x" dimension="none" value="(vpeer - v)/(I_rev_max/
↪forward_conductance)"/>

        <ConditionalDerivedVariable name="i" exposure="i" dimension="current">

            <!-- Special case for near the singularity, linearise ! -->
            <Case condition="abs(x) < 0.0001" value="x * I_rev_max / 2"/>

            <Case  value="I_rev_max * ( (x / (1 - exp(-x))) - 1 )"/> <!-- Usual␣
↪case, but beware of exp(x) = 1 ! -->
        </ConditionalDerivedVariable>

    </Dynamics>
</ComponentType>
<!-- And one more for the other side ! -->
<ComponentType name="GapDiode_Anode" extends="baseGradedSynapse" description="Non-
↪symmetrical gap junction, anode half">
    <Parameter name="I_rev_max" dimension="current" description="Asymptotic max␣
↪reverse current"/>
    <Parameter name="forward_conductance" dimension="conductance" description=
↪"Marginal (ie small-signal) conductance when fwd polarised"/>
    <Dynamics>

        <!-- Now things get interesting. FLIP x to sample the same asymmetric␣
↪curve,
            and then FLIP the result for the opposite current into this side !  -->
        <DerivedVariable name="x" dimension="none" value="(v - vpeer)/(I_rev_max/
↪forward_conductance)"/>
        <ConditionalDerivedVariable name="i" exposure="i" dimension="current">=
            <Case condition="abs(x) < 0.05" value=" - I_rev_max * x"/>
            <Case  value=" - I_rev_max * ( (x / (1 - exp(-x))) - 1 )"/> <!-- Usual␣
↪case -->
        </ConditionalDerivedVariable>

    </Dynamics>
```

(continues on next page)

(continued from previous page)

```
</ComponentType>

<!-- Here comes the concrete synapse type with parameters ! -->
<GapDiode_Cathode id="MyCathode" forward_conductance="10 nS" I_rev_max="0.001 nA"/>
<GapDiode_Anode   id="MyAnode"   forward_conductance="10 nS" I_rev_max="0.001 nA"/>

<!-- Add a simple cell type, and a pair of pulse sources -->
<iafCell id="MyFirstCellType" C="200 pF" leakConductance="10 nS" leakReversal="-70␣
↪mV" reset="-70mV" thresh="-50mV" />
<pulseGenerator id="Inp0" delay="150ms" duration="100ms" amplitude="0.1nA"/>
<pulseGenerator id="Inp1" delay="750ms" duration="100ms" amplitude="0.1nA"/>
<!-- Here comes the network -->
<network id="Net" type="networkWithTemperature" temperature="37degC" >
    <!-- Add a pair of cells -->
    <population id="Pop" component="MyFirstCellType" size="2"/>
    <!-- Connect them with a synapse -->
    <continuousProjection id="Pro" presynapticPopulation="Pop"␣
↪postsynapticPopulation="Pop">
        <continuousConnection id="0" preCell="0" postCell="1" preComponent="MyAnode
↪" postComponent="MyCathode"/>
    </continuousProjection>
    <!-- And stimulate them -->
    <inputList id="Inpli0" population="Pop" component="Inp0"><input id="0" target=
↪"0" destination="synapses"/></inputList>
    <inputList id="Inpli1" population="Pop" component="Inp1"><input id="0" target=
↪"1" destination="synapses"/></inputList>
</network>
</neuroml>
```

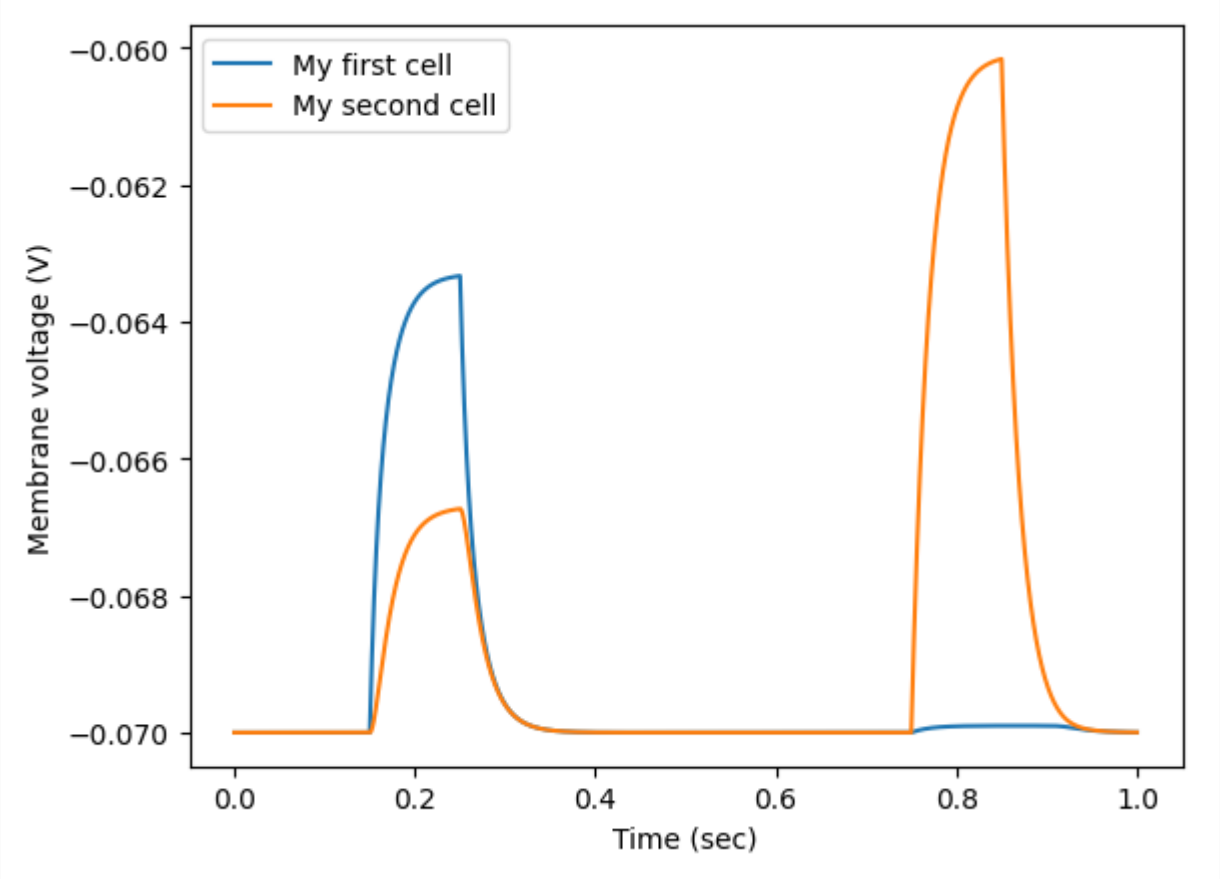
```
Writing Custom_GradSyn_Model.nml
```

And let's see what happens:

```
%%writefile LEMS_Custom_GradSyn_Sim.xml
<Lems>
<include file="Custom_GradSyn_Model.nml" /> <!-- Remember to check the filename -->
<Simulation id="Sim" length="1 s" step="0.1 ms" target="Net" >
    <OutputFile id="MyFirstOutputFile" fileName="results.gen.txt">
        <OutputColumn id="vm" quantity="Pop[0]/v"/>
        <OutputColumn id="v2" quantity="Pop[1]/v"/>
    </OutputFile>
</Simulation>
<Target component="Sim"/>
</Lems>
```

```
Writing LEMS_Custom_GradSyn_Sim.xml
```

```
from eden_simulator import runEden; from matplotlib import pyplot as plt
results = runEden('LEMS_Custom_GradSyn_Sim.xml')
plt.plot(results['t'],results['Pop[0]/v'], label='My first cell')
plt.plot(results['t'],results['Pop[1]/v'], label='My second cell')
plt.xlabel('Time (sec)'); plt.ylabel('Membrane voltage (V)'); plt.legend();
```



When current is injected to the first cell, a significant part flows to the second cell; but when the second cell is stimulated, barely any flows "back" to the first cell. Observe that the potential peak is in the former case is less than in the latter, since the current is divided among both cells.

Note that a single ComponentType could also be used here: the $V - I$ curve can be flipped by means of an additional "polarity" `Parameter`. And in fact, in this model the `I_rev_max` parameter happens to affect `x` linearly in both places where a flip is needed: flip the sign on `MyCathode` and see what happens vs. `MyAnode` !

More synapse examples

For further LEMS examples of on complicated synapse models, check out an auxiliary spike counter[183] implemented as a synapse, and highly-retailed models of multi-path plasticity[184] (Ebner et al. 2019[185]), and different interacting[186] receptor-transmitter pairs (Schwartz et al. 2012[187]).

[183] https://github.com/OpenSourceBrain/GranCellRothmanIf/blob/master/neuroConstruct/lemsSimulations/rate_IO/LEMS_rate_IO.xml#L10
[184] https://github.com/OpenSourceBrain/EbnerEtAl2019/blob/master/NeuroML2/fourPathwaySyn.synapse.nml#L11
[185] https://doi.org/10.1016/j.celrep.2019.11.068
[186] https://github.com/OpenSourceBrain/GranCellRothmanIf/blob/master/neuroConstruct/cellMechanisms/RothmanMFToGrCAMPA/RothmanMFToGrCAMPA.nml#L40
[187] https://doi.org/10.1523/JNEUROSCI.5736-11.2012

4.3.5 Custom input sources

There are two types of input sources: neuron-attached probes, and self-standing spike generators.

Base <ComponentType>s for this type to extend include for the former <basePointCurrent>[188], <baseVoltageDepPointCurrent>[189], <baseVoltageDepPointCurrentSpiking>[190], <basePointCurrentDL>[191], <baseVoltageDepPointCurrentDL>[192], and for the latter <baseSpikeSource>[193].

Note that if some neuron-attached input sources are implied to emit spikes in the LEMS definitions, they actually don't: the existence of ports is just a technicality of the LEMS underpinnings, outside the NeuroML context. Therefore <baseVoltageDepPointCurrentSpiking> is equivalent to <baseVoltageDepPointCurrent>.

The LEMS *interface* for probes is as follows:

Available quantities to have as<Requirement>*s:*

- All the intra-cell quantities (page 66), plus:

Suggested<Exposure>*s:*

- i: The current that this probe injects into the cell.
- g: The electrical conductance of the mechanism, basically $\frac{di}{dV_m}$. May help with simulation accuracy when available.
- I: Similar, but used for dimensionless current flowing into cells with dimensionless state variable.)
- Other flows that a probe may inject to a cell using the Multiflux extension (page 121).

For spike sources, requirements and exposure are as for point neurons (page 66); refer to the same section on making custom spike sources.

Example: Ornstein-Uhlenbeck noise

An popular electrical signal to stimulate cells with is Ornstein-Uhlenbeck noise[194] which is thought to better[195] approximate neural activity[196] than, say, white noise. This is a stochastic process that evolves as a *random walk*. A very interesting feature of random walks is that due to their fractal nature, the change of its state *non-linear on the simulation's timestep.* Hence a <TimeDerivative> won't cut it, regardless of (first order accurate or better) ODE solver. We need to update the value with an *explicit* formula on every timestep[197]!

Besides showing how to make a custom stimulus probe, this model also shows two commonly requested tricks:

- Random numbers following the *normal* distribution instead of the *uniform* that random() provides;
- *Discrete* updates that happen on *every timestep*. **Note:** This is outside the LEMS specification and does *not necessarily* behave the same on all simulators! But this is a proposed feature for LEMS and it may get codified in the future.

```
%%writefile OU_Input_Model.nml
<neuroml>
<!-- Let's make a new ComponentType ! -->
<ComponentType name="Noise_OU" extends="basePointCurrent">
    <Parameter name="mean" dimension="current" description="Average current to␣
↪revert to"/>
```

(continues on next page)

[188] https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#basepointcurrent
[189] https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#basevoltagedeppointcurrent
[190] https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#basevoltagedeppointcurrentspiking
[191] https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#basepointcurrentdl
[192] https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#basevoltagedeppointcurrentdl
[193] https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#basespikesource
[194] https://en.wikipedia.org/wiki/Ornstein-Uhlenbeck_process
[195] https://doi.org/10.1007/BF01845839
[196] http://hdl.handle.net/11299/199178
[197] https://en.wikipedia.org/wiki/Ornstein%E2%80%93Uhlenbeck_process#Numerical_simulation

(continued from previous page)

```
    <Parameter name="sigma" dimension="current" description="Standard deviation of␣
↪current"/>
    <Parameter name="tau" dimension="time" description="Correlation time constant"/
↪>
    <Parameter name="delta_t" dimension="time" description="Sampling period for␣
↪noise"/> <!-- Should be exactly  a <Simulation> step ! -->
    <Dynamics>
        <StateVariable name="i" exposure="i" dimension="current"/>
        <DerivedVariable name="randn" exposure="randn" dimension="none" value=
↪"sqrt(-2*ln(random(1)))*cos(2*3.14159265359*random(1))" description="A normally␣
↪random variable for every moment !"/>
        <OnCondition test="1+1==2"> <StateAssignment variable="i" value="i + (mean␣
↪- i)*delta_t/tau + sigma*randn*sqrt(2*delta_t/tau)"/> </OnCondition>
        <OnStart> <StateAssignment variable="i" value="mean"/> <!-- or to taste -->
↪ </OnStart>
    </Dynamics>
</ComponentType>

<!-- Add a simple cell type, and this input source -->
<iafCell id="MyFirstCellType" C="200 pF" leakConductance="10 nS" leakReversal="-70␣
↪mV" reset="-70mV" thresh="-50mV" />
<Noise_OU id="MyNoise" mean="0.12 nA" sigma="0.02 nA" tau="300 msec" delta_t="0.1␣
↪ms"/>

<network id="Net" type="networkWithTemperature" temperature="37degC" >
    <!-- Add a cell -->
    <population id="Pop" component="MyFirstCellType" size="1"/>
    <!-- And stimulate it -->
    <inputList id="Inpli0" population="Pop" component="MyNoise"><input id="0"␣
↪target="0" destination="synapses"/></inputList>
</network>
</neuroml>
```

```
Writing OU_Input_Model.nml
```

```
%%writefile LEMS_OU_Input_Sim.xml
<Lems>
<include file="OU_Input_Model.nml" /> <!-- Remember to check the filename -->
<Simulation id="Sim" length="1 s" step="0.1 ms" target="Net" seed="20190109" > <!--
↪ seed used -->
    <OutputFile id="MyFirstOutputFile" fileName="results.gen.txt">
        <OutputColumn id="vm" quantity="Pop[0]/v"/>
        <OutputColumn id="I" quantity="Inpli0[0]/i"/>
    </OutputFile>
</Simulation>
<Target component="Sim"/>
</Lems>
```

```
Writing LEMS_OU_Input_Sim.xml
```

And let's see what happens.

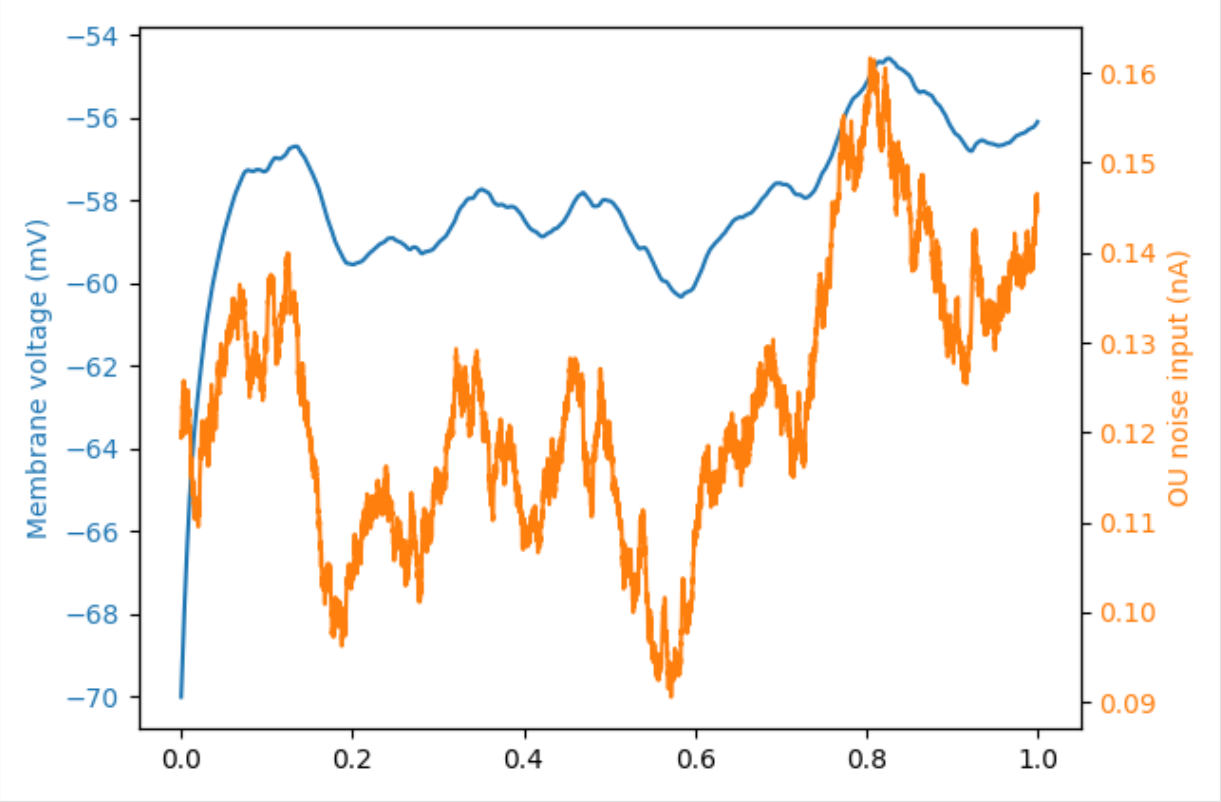
Note: We'll use extended LEMS paths (page 87) to access the source's current, so we can show it side by side with voltage.

```
from eden_simulator import runEden; from matplotlib import pyplot as plt
results = runEden('LEMS_OU_Input_Sim.xml')
def halflines(ax,x, y, color, label):
    ax.set_ylabel(label, color=color)
```

(continues on next page)

(continued from previous page)

```
    ax.plot(x,y, color=color)
    ax.tick_params(axis='y', labelcolor=color)
col1 = 'tab:blue'
halflines(plt.gca(), results['t'],results['Pop[0]/v']*1000, color='tab:blue',␣
↪label='Membrane voltage (mV)')
ax2 = plt.gca().twinx()
halflines(plt.gca(), results['t'],results['Inpli0[0]/i']*1e9, color='tab:orange',␣
↪label='OU noise input (nA)')
plt.xlabel('Time (sec)');
```



Observe the delay between swings of input current and current. The membrane becomes less polarised than normally, due to the "positive" input current.

Set mean input to 0 or increase sigma and see what happens to the plots: autoscale can be misleading !

For an arbitrary sampling rate (and more commonly agreed behaviour), it just takes keeping track of the time of last update. Refer to the example[198] from the OpenSourceBrain Stochasticity Showcase[199].

4.3.6 Custom ion channels, gates and rates

In biophysically modelled (page 33) neurons, NeuroML offers a quite rich variety of modelling options. The summary is that *ion channels* are made up of *gates*, which open and close according to *transition rates*. Each of these parts can be modelled by a component from the core NeuroML library, or by a component described in LEMS. Details and examples on how to model each part follow:

[198] https://github.com/OpenSourceBrain/StochasticityShowcase/blob/a5911a6/NeuroML2/NoisyCurrentSource.xml#L36

[199] https://github.com/OpenSourceBrain/StochasticityShowcase

Custom gate transition rates

The most common customisation is to just select different α, β or τ, x_∞ rates for the gate variables or ion channels, under either the Hodgkin-Huxley or Markov[200] formulation.

Base `<ComponentType>`s for this type to `extend` include `<baseVoltageDepRate>`[201], `<baseVoltageConcDepRate>`[202], `<baseVoltageDepTime>`[203], `<baseVoltageConcDepTime>`[204], `<baseVoltageDepVariable>`[205], and `<baseVoltageConcDepVariable>`[206]. Different `<Requirement>`s are inherited, depending on the base `<componentType>`.

The LEMS *interface* for gate rates is as follows:

Available quantities to have as`<Requirement>`*s:*

- All the intra-cell quantities (page 66).

Required`<Exposure>`*s:* One of either, depending on formula type:

- `r` (`dimension`: `per_time`): for `forwardRate` and `reverseRate`
- `t` (`dimension`: `time`): for `timeCourse`
- `x` (`dimension`: `none`): for `steadyState`

Example: Custom HH rates for a calcium channel

An example from the NeuroML-DB on how to write your own α, β rates for your gate variables can be seen here[207] (source[208]).

Example: Custom `timeCourse` for a sodium channel

An example using both a core NeuroML `steadyState` and a custom `timeCourse` can be seen here[209] (source[210]).

Example: A $Ca2^+$-dependent potassium channel

An example using *calcium-dependent* dynamics can be seen here[211] (source[212]).
Both `timeCourse` and `steadyState` are customised into components that extend `<baseVoltageConcDepTime>`[213] and `<baseVoltageConcDepVariable>`[214] and expose `t` and `x` respectively (though a `<fixedTimeCourse>`[215] could have been used in this case).

[200] https://docs.neuroml.org/Userdocs/Schemas/Channels.html#ionchannelks
[201] https://docs.neuroml.org/Userdocs/Schemas/Channels.html#basevoltagedeprate
[202] https://docs.neuroml.org/Userdocs/Schemas/Channels.html#basevoltageconcdeprate
[203] https://docs.neuroml.org/Userdocs/Schemas/Channels.html#basevoltagedeptime
[204] https://docs.neuroml.org/Userdocs/Schemas/Channels.html#basevoltageconcdeptime
[205] https://docs.neuroml.org/Userdocs/Schemas/Channels.html#basevoltagedepvariable
[206] https://docs.neuroml.org/Userdocs/Schemas/Channels.html#basevoltageconcdepvariable
[207] https://neuroml-db.org/render_xml_file?modelID=NMLCH000132
[208] https://neuroml-db.org/model_info?model_id=NMLCH000132
[209] https://neuroml-db.org/render_xml_file?modelID=NMLCH000164
[210] https://neuroml-db.org/model_info?model_id=NMLCH000164
[211] https://neuroml-db.org/render_xml_file?modelID=NMLCH001397
[212] https://neuroml-db.org/model_info?model_id=NMLCH001397
[213] https://docs.neuroml.org/Userdocs/Schemas/Channels.html#basevoltageconcdeptime
[214] https://docs.neuroml.org/Userdocs/Schemas/Channels.html#basevoltageconcdepvariable
[215] https://docs.neuroml.org/Userdocs/Schemas/Channels.html#fixedtimecourse

Note that in all cases, `v` and `caConc` is normalised into dimensionless quantities(of implicit `milliVolt` and `kiloMolar` units), so that they can be used in ~~funky~~*empirical* formulae which wouldn't make sense with dimensional quantities.

Custom gates

One may prefer a fully-custom model for ion channel gates, when customising its rates is not enough. The `<componentType>` to `extend` is `<baseGate>`[216].

The LEMS *interface* for gates is as follows:

Available quantities to have as `<Requirement>` *s:*

- All the intra-cell quantities (page 66).

Required `<Exposure>` *s:*

- `q` (`dimension`: `none`): The 'state' of the gate population, by convention the fraction of gates that are open.
- `fcond` (`dimension`: `none`): The *activation* of the gate (its multiplicative factor on channel conductance). May be, for example, `q` to the power of the gate's multiplicity. Think of it as "*f*raction of *cond*uctivity".

Custom conductance scaling for ion channels

More than ion channel gates, the conductance of an ion channel may be affected by other biophysical factors. Along with the temperature-dependent `<q10ConductanceScaling>` one may also specify custom LEMS components, that may also depend on other factors. Base `<ComponentType>`s for this type to `extend` include `<baseConductanceScaling>`[217] and `<baseConductanceScalingCaDependent>`[218] .

The LEMS *interface* for conductance scaling is as follows:

Available quantities to have as `<Requirement>` *s:*

- All the intra-cell quantities (page 66).

Required `<Exposure>` *s:*

- `factor` (`dimension`: `none`): The number that this scaling effect multiplies the ion channel's base conductance by.

Example: $Ca2^{+}$-dependent base conductance

An example of *calcium-dependent* `baseConductance` for an ion channel can be seen here[219] (source[220]).

Note that in this case it is inserted in the ion channel as a `<baseConductanceScalingCaDependent type="cond_scaling_kc_fast"/>`, the way EDEN detects it is through the `type` attribute of the tag (or the tag name that should match the `ComponentType` if `type`), and the fact that it `extends baseConductanceScalingCaDependent`.

[216] https://docs.neuroml.org/Userdocs/Schemas/Channels.html#basegate
[217] https://docs.neuroml.org/Userdocs/Schemas/Channels.html#baseconductancescaling
[218] https://docs.neuroml.org/Userdocs/Schemas/Channels.html#baseconductancescalingcadependent
[219] https://neuroml-db.org/render_xml_file?modelID=NMLCH000152
[220] https://neuroml-db.org/model_info?model_id=NMLCH000152

Custom ion channel models

Finally, one may prefer a fully-custom model for the whole ion channel. The `<componentType>` to `extend` is `<baseIonChannel>`[221].

The LEMS *interface* for gates is as follows:

Available quantities to have as`<Requirement>`*s:*

- All the intra-cell quantities (page 66).

Required`<Exposure>`*s:*

- `fopen` (`dimension`: `none`): The fraction of ion channels that are open, *after* accounting for conductance scaling.
- `g` (`dimension`: `conductance`): The electrical conductance of the ion channel, in absolute terms.

4.3.7 Custom ion concentration models

Ion concentration models determine the dynamical relation between local ion concentration and the ion fluxes that feed it. For custom dynamics, the base `<ComponentType>`to `extend` is `<concentrationModel>`[222].

The LEMS *interface* for concentration models is as follows:

Available quantities to have as`<Requirement>`*s:*

- All the intra-cell quantities (page 66), plus:
- `surfaceArea`: The total surface area of the compartment's membrane.
- `initialConcentration`: The initial value for concentration, as specified with `<species>` tags inside `<intracellularProperties>`.
- `initialExtConcentration`: The same for the outer side of the membrane.
- `iCa`, `iCa2`: the total per-compartment inflow in terms of `current` for the two independent calcium ion species that NeuroML supports. Perhaps the other ion flows should also be exposed like this? What about species without a charge?

Required`<Exposure>`*s:*

- `concentration`: The current value of internal concentration.
- `extConcentration`: Likewise for the outer side.

Example: An extended concentration model from Hay et al. 2011

An alternative concentration model from Hay et al. 2011[223] can be seen here[224] (source[225]). It is better than the similar `<decayingPoolConcentrationModel>`[226] from core NeuroML, in that the latter assumes the compartment to be a *sphere* and calculates the near-membrane volume accordingly. Hence the core NeuroML model is better suited for neurons modelled as spheres, whereas the `<concentrationModelHayEtAl>` model is better suited to anatomically detailed (page 33) cells.

Note: Do take into account what this model does *before* using it, and customise to fit your own model!

[221] https://docs.neuroml.org/Userdocs/Schemas/Channels.html#baseionchannel
[222] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#concentrationmodel
[223] https://doi.org/10.1371/journal.pcbi.1002107
[224] https://neuroml-db.org/render_xml_file?modelID=NMLCN000001
[225] https://neuroml-db.org/model_info?model_id=NMLCN000001
[226] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#decayingpoolconcentrationmodel

Example: A concentration model for an independent calcium pool

Some models have not one, but two separate species of calcium ion that have different effects and dynamics (see also `<cell2CaPools>` and friends.) A concentration model for the second species can be specified as in here[227] (source[228]).

This article listed the sorts of ways for EDEN users to specify whole new mechanisms in NeuroML.

Refer also to the official NeuroML user guide:

- On how NeuroML and LEMS intersect[229]
- More on making new components[230]
- The core NeuroML component type library[231]
- The user guide for pure LEMS[232] (also beyond what NeuroML covers)

Further than the NeuroML specification, EDEN also offers some extensions (page 85) that give even more powers to LEMS components.

[227] https://neuroml-db.org/render_xml_file?modelID=NMLCN000069
[228] https://neuroml-db.org/model_info?model_id=NMLCN000069
[229] https://docs.neuroml.org/Userdocs/NeuroMLv2AndLEMS.html
[230] https://docs.neuroml.org/Userdocs/ExtendingNeuroMLv2.html
[231] https://docs.neuroml.org/Userdocs/NeuroMLv2.html
[232] https://docs.neuroml.org/Userdocs/LEMS.html

Part B

Beyond NeuroML: EDEN-specific extensions

NeuroML and LEMS support modelling a fairly wide range of sub-system dynamics of interactions, but often these are not enough to capture all the properties that that modellers ask for. EDEN offers a further set of capabilities that can support an even wider variety of model dynamics, while keeping most of the model description within official NeuroML.

CHAPTER

FIVE

EXTENDED LEMS PATHS

A NeuroML model's variables and event sources can be designated in the form of LEMS paths[233]. These paths can be used for recording, among other uses. However, there are some limitations of the paths currently used in NeuroML:

- LEMS is not enough to cover most aspects of spatial cells, and discretisation (page 33) is one of them. Hence, there should be a way to refer to a specific location on a cell, in order to capture the individual compartments in all cases.
- The way that `<Attachment>`s (that is, synapses and input sources) are accessed is ambiguous. They are supposed to be numbered in the order that *mechanisms* were attached to the cells, *without distinction to the different component types, or logical groups of mechanisms attached* (for example, instances of the same synapse type that exist on the same cell are lumped together for all `<projection>`s). Even worse, the order that mechanisms are added is *unspecified*.

Hence, EDEN also supports alternative forms to specify precise locations (page 38) on a spatially detailed cell, or a synapse or input mechanism, reliably. These *extended LEMS paths* can be used to designate and record quantities and event sources alike, and EDEN's `CustomSetup` extension (page 89) is also based on them. They are written as follows:

5.1 LEMS paths for cell locations

The typical way to access variables in a spatially detailed (page 33) cell is LEMS paths of the form:

- **`population/cell_id[/segment[.fractionAlong]]/`: (options continue from this path as follows)**
  - `v` for membrane voltage
  - `caConc` for `ca` ion concentration
  - `ca2Conc` for `ca2` ion concentration (a separate ion species of the same atom)
  - **`biophysicalProperties/membraneProperties/`:**
    - `channel distribution/`:
    - `iDensity` or `gdensity` for the local current and conductance *per area* of the ion channel distribution
    - `i` or `g` for the current and conductance of the mechanism *over the whole pointed compartment* (recommended only in special cases)
    - `channel mechanism/gate_name/q` for gate variables
- Another way to specify a cell is `population[cell_id]/...` and the path follows the same way.

The parts in square brackets are optional (if the part in enclosing brackets is used). If `segment` is not specified, then it is assumed to be `0` (the soma, by convention). If `fractionAlong` is not specified, then it is assumed to be `0.5` (the middle of the segment).

[233] `https://docs.neuroml.org/Userdocs/Paths.html`

This way, any site on a neuron can be referred to with a LEMS path. This is especially useful in conjunction with `explain_cell` (page 241) and `GetLemsLocatorsForCell` (page 241), and for fully recording over a cell, as seen in NeuroML and spatially detailed cells (page 33) and other examples.

5.2 LEMS paths for input list elements

Instead of the form[234] `population/cell_id/synapses:mechanism name:serial number of attached instance/property`, which is supported by jLEMS (under certain assumptions), this is how quantities of `input` mechanisms can be accessed in EDEN as elements of their `<inputList>`:

```
input_list/0/variable_name
input_list[0]/...
```

5.3 LEMS paths for synaptic projection elements

Instead of the LEMS form which may assign serial number in unpredictable ways, EDEN offers an alternative form similar to that for `<inputList>`s. After the synapse instance and the mechanism's elements, a `pre` or `post` locator is added to select between either half of the synapse (for simple `<projection>`s, only `post` is valid). The pattern is then as follows:

```
projection/0/(pre or post)/variable_name
projection[0]/(pre or post)/...
```

Using these forms, one can record or otherwise access (page 107) the variables of individual synapses or input sources, in the consistent order that they had in their respective `<projection>`s or `<inputList>`s.

5.4 LEMS paths for input streams

The same form can be further used to point to elements of `<EdenTimeseriesReader>` (page 116)s and `<EdenEventSetReader>` (page 118)s and their properties, for `<VariableReference>` (page 107)s to point at or for recording:

- `time series/instance[/column]` (`column` can be omitted if elements have just one)
- `event set/instance[/port]` (`port` can be omitted if elements have just one)
- And the `group[instance]/...` form can also be used as usual.

[234] `https://docs.neuroml.org/Userdocs/Paths.html`

CHAPTER

SIX

CUSTOM PER-ELEMENT VARIABILITY

As we saw in the NeuroML introduction tutorial (page 13), models comprise a `<network>` of neuron *populations* connected by synapse *projections*, and some experimental rig around them to interact with through.

In practice, a problem that emerges often when modelling is that NeuroML defines populations of *identical* neurons, and projections with identical components. The state of each instance may of course vary through the simulation, but the parameters and initial state are the same for all instances of a cell (or synapse or input type, or generally LEMS component instance).

However, this is not always sufficient to make a realistic model, as neurons can show considerable physiological variability in nature. And even when neurons and synapses are of the same type, a small amount of variability is desired in models to make them more robust[235].

For these reasons, modellers often require that each neuron or synapse may have its parameters *slightly different* than the rest; and what's more, the *kind* of variability is often unique to that model part, or at least project or research laboratory. Typical examples of variability models include mathematical formulae, anatomical data or random sampling of statistical distributions. What's then needed is a way to apply *any* kind of variability to the model directly. And in the general case, this includes assigning an *explicit* set of values, one by one, to a collection of mechanisms.

EDEN supports models with arbitrary per-instance variability, in a way that allows for any modelling pipeline by design. It does this through the `<EdenCustomSetup>` extension, and a special *setup language* for supplying and designating the variability data.

Tip

This is a simulator-specific interim data format. It may be replaced with equivalent NeuroML capabilities as they appear.

The first step in using custom variability is to add the tag `<EdenCustomSetup filename="<file name>" />` inside the usual `<Simulation>` tag.

The file under `file name` then, contains the customisation information as follows.

6.1 The `EdenCustomSetup` file specification

The setup language is (to this point) essentially a sequence of `set` statements, which set the values of quantities (i.e. `<Property>`s) over a set of mechanisms.

Note that these statements are considered in *strict order*: a `set` statement may supersede values assigned by a previous statement to the same mechanism!

In order to specify a *collection* of mechanisms instead of just a single one, the LEMS path machinery is used in a kind of re-mixed form. It is broken down into:

- the part that locates an entity in the `<network>`, which in turn is broken down into:

[235] https://doi.org/10.1371/journal.pone.0080694

  - the "collection" of entities in the network, that is a neuron `<population>`, synaptic `<projection>`, or `<inputList>`,
  - and the "identifiers" of entities within said collection. A `set` statement may name multiple identifiers, to share the same quantity value or get each one value from a list of `multi`ple, as we'll see below.
  - For spatially-detailed `<cell>` populations, there is also a list of different cell locations (page 38) that comes after the list of cell identifiers. The mechanisms in these locations may in turn take the same value, or different ones.
- and the path *within* the entity that locates the quantity in terms of the sub-mechanism that it belongs to, and its identifier name within.

For the extended set of LEMS paths that EDEN understands (and a hint as to how they may be applied here), refer to EDEN's LEMS paths reference (page 87).

6.1.1 General structure

`<EdenCustomSetup>` files are written as human-readable text lines; an equivalent binary format for higher data density is also planned.

The contents of the file are arranged in *lines*; each line contains *tokens* separated with space and tab characters, and other "white-space" characters.

The types of lines are as follows:

- `set` statements, for:
  - `cell` populations
  - `synapse` projections
  - `input` lists
- `values` lines, which follow the `set` statements that specify `multi`ple values;
- empty and "comment" lines to aid readability for humans.

The structure of these lines is described below.

6.1.2 `set cell` statement

A `set cell` statement line is a sequence of tokens as follows:

```
set cell <population> <cell list> <location list> <attribute> <value>
```

The tokens have the following meanings, in order:

- `population`: The name of the neuron `<population>` whose cells' quantities are customised.
- `cell list`: Either a comma-separated list of cell instance `id`'s in strictly ascending order, or `all` to designate the whole population.
- `location list`: Can be either:
  - a comma-separated list of segment locators (page 87) on cell;
  - `all` to apply all over the cell;
  - or the name of a "cable" (page 40) segment group, to sample values along a track of sampling points.

  For point neurons, options are the equivalent `all` or `0` (the implicit singular "segment" `id`, as per LEMS paths (page 87)).
- `attribute`: The part of the LEMS path (page 87) *within* the neuron, that follows after `population[id]/segment/`

- `value`: The single quantity value ,or `multi` to specify more, in the `values` lines that follow. Examples for how `values` look like:
  - `100 pA` to specify an amount of electrical current;
  - `multi mV` to specify one or more voltage values, in one or more `values` lines that follow;
  - `3.14159` to specify a dimensionless, pure number;
  - `multi` to specify one or more dimensionless numbers, or <VariableRequirement> paths (page 93).

If the `location list` is "all" and `multi`ple values are specified, one `values` line follows, with one value per cell instance in the `cell list`.

If the `cell list` includes only one cell, the `location list` contains multiple segment locators and `multi`ple values are specified, one `values` line follows, with one value per segment locator in the `location list`.

When both the `cell list` contains multiple cells and the `location list` contains multiple segment locators, there are different ways to specify multiplicity. The `multi` value specifier has to be replaced by:

- `multi_location`, in which case the same values are applied to all cells, and one `values` line follows with the value for each segment locator;
- `multi_cell`, in which case the same value is applied to all segment locators per cell, and one `values` line follows with the value for each cell;
- `multi_multi`, in which case a different value is applied to each cell on each segment locator. As many `values` lines as the number of mentioned cells follow, each line with the value for each segment locator on the listed cell.

Cable mode

When the location list is a "cable" `<segmentGroup>`, a special *cable interpolation mode* is engaged, wherein the mechanism value is sampled across the cable's extent for each compartment, based on a piecewise-linear function that's provided in the `values` lines that follow.

The `value` token must be `cable` or `cable_multi`. At least two `values` lines follow:

- The first `values` line is special; it contains the sampling points for the values that follow. These sampling points represent the *relative fractional length along the cable, starting from the proximal end and up to the distal end*, and thus may have values from 0 to 1.
- If `value` is `value_multi`, one `values` line for each cell specified in the `cell list` follows, each line with the piecewise-linear function's values at the already specified sampling points.

Tip

This mode is targeted toward porting models written for NEURON, when values for `DENSITY` mechanism quantities are assigned over a neuron's `section` with any sort of variability. The values instantiated on the NEURON model can be extracted with:

```
access <section>; for (x) print x, <property>(x)
```

as seen in chapter 5 of the NEURON book, section "Working with range variables".

6.1.3 `set synapse` statement

A `set synapse` statement line is a sequence of tokens as follows:

```
set synapse <projection> <synapse list> <site> <attribute> <value>
```

The tokens have the following meanings, in order:

- `projection`: The name of the synaptic `<projection>` whose components' quantities are customised.
- `synapse list`: Either a comma-separated list of `<connection>` `id`'s in strictly ascending order, or `all` to designate the whole projection.
- `site`: either or `pre` or `post`. Since classical event-based `<projection>`s have a synaptic component only on the post-synaptic site, `post` is the only allowed site for them.
- `attribute`: The part of the LEMS path (page 87) *within* the synaptic component, that follows after `projection[id]/post/`.

Examples for `values` are the same as with cells (page 90).

When `multi`ple values are specified, one `values` line follows, with one value per mechanism instance in the specified list.

6.1.4 `set input` statement

A `set input` statement line is a sequence of tokens as follows:

```
set input <list name> <input list> <attribute> <quantity with units>
```

The tokens have the following meanings, in order:

- `projection`: The name of the `<inputList>` whose probes' quantities are customised.
- `input list`: Either a comma-separated list of `<input>` `id`'s in strictly ascending order, or `all` to designate the whole `<inputList>`.
- `attribute`: The part of the LEMS path (page 87) *within* the input source component, that follows after `inputlist[id]/`.

Examples for `values` are the same as with cells (page 90).

When `multi`ple values are specified, one `values` line follows, with one value per mechanism instance in the specified list.

6.1.5 Comment lines and inline comments

In the text-based format, there may be some space between, and on, the statement lines to write comments on, to aid people (or perhaps an automated process):

- *Empty lines* may be present anywhere in the text; they are ignored, but may not split other lines.
- Within each line, the part starting from the first token that starts with a `#` character, is also ignored.

6.1.6 Setting `VariableRequirements`

Along with numerical values for regular quantities, `set` statements can also be used to assign the *targets* of `<VariableRequirement>` (page 107)s; in fact, this is the only way for them to take a value (which they must!). In this case, the value to specify is a LEMS path (page 87) instead of a number. There is no need to specify units, as the model's quantities are automatically managed by EDEN.

Note: When a `VariableRequirement` set of values is interpolated over an unbranched section using cable mode (page 91) with sample points, nearest-neighbour interpolation[236] is used instead of the linear interpolation that's used for quantities. (If there is one sampling point per compartment, the effect is the same.)

6.2 Example: A network with variability over neurons, synapses and probes

To show how the above specification works, let's make a 2-D connected grid of neurons and randomise parameters throughout:

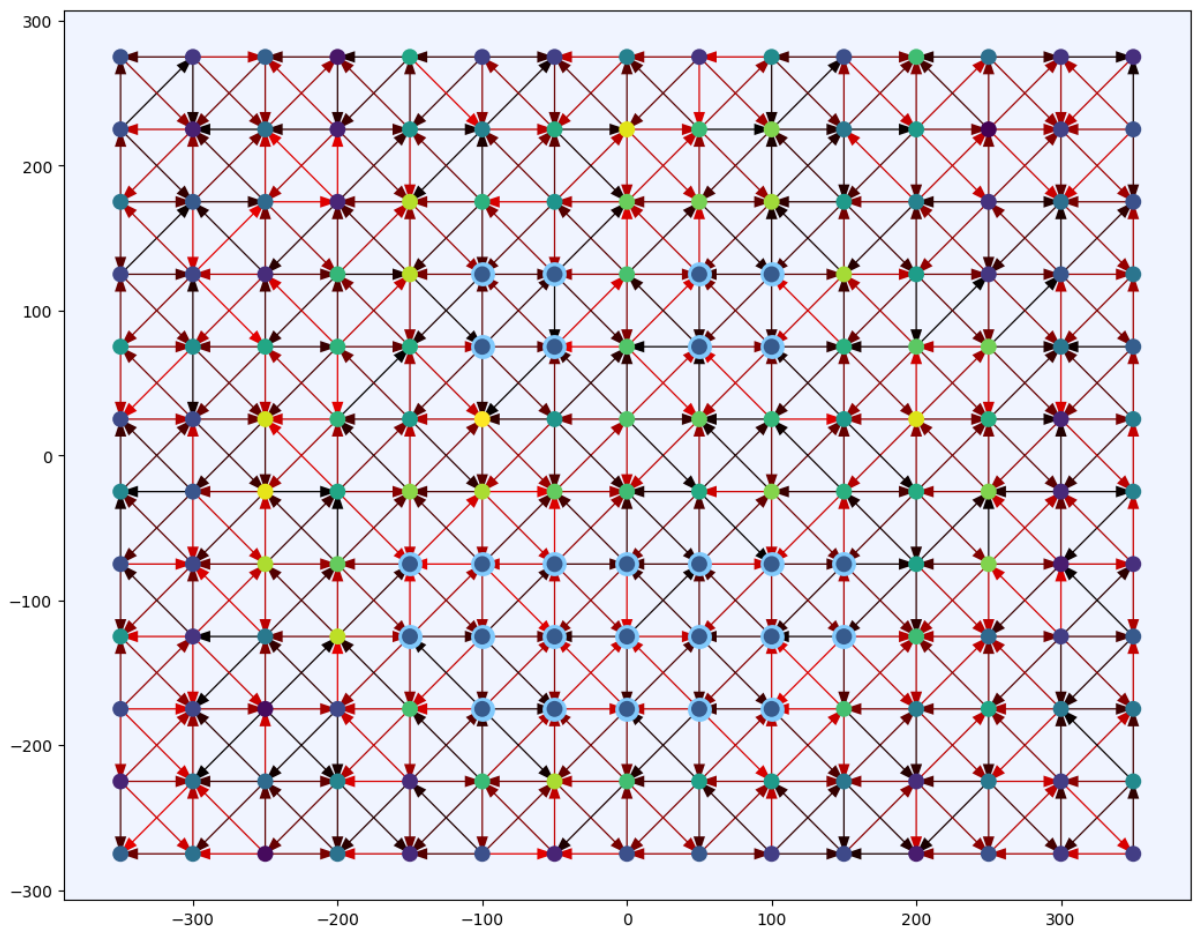
```
import numpy as np; from numpy.linalg import norm; from matplotlib import pyplot␣
↪as plt
rng = np.random.default_rng(seed=43264326)
offsets = [(1,0), (-1,1), (0,1), (1,1)] # sequential-first neighborhood
# for bidirectional: offsets = [(x,y) for y in [-1, 0, +1] for x in [-1, +1]]+[(0,
↪y) for y in [-1, +1]]

# Make a 2D grid and then linearise it
grid_step_size_microns = 50
X_steps = 15; x_space = range(X_steps); X_hw = (X_steps-1)*grid_step_size_microns/2
Y_steps = 12; y_space = range(Y_steps); Y_hw = (Y_steps-1)*grid_step_size_microns/2
xmat, ymat = np.meshgrid(x_space, y_space, indexing='ij')
neuron_points_steps = np.stack((xmat,ymat),axis=-1).reshape((-1,2)); neurons =␣
↪len(neuron_points_steps)
def xy_to_index(x,y): return x*Y_steps + y
# and in spatial, not abstract coordinates
neuron_points_microns = (np.array(neuron_points_steps) + .5 - (X_steps/2, Y_steps/
↪2)) * grid_step_size_microns
pts = neuron_points_microns # shorthand

# Now gather the possible synapses under Neumann adjacency (ie 8-way)
syn_list = [(xy_to_index(x,y), xy_to_index(xx,yy))
    for (x,y) in neuron_points_steps
    for (dx, dy) in offsets for (xx, yy) in [(x+dx, y+dy)]
    if 0 <= xx < X_steps and 0 <= yy < Y_steps
]
# Randomly flip the directions of some connections (could have both directions but␣
↪it would be hard to show that)
syn_list = np.array(syn_list)
which_to_flip = rng.uniform(size=len(syn_list)) < 0.5
syn_list[which_to_flip] = [(y, x) for (x,y) in syn_list[which_to_flip]]

# And set up some parameters
input_stim_idxs = np.where( 0
    | ( norm((pts - (+3*X_hw/14,+Y_hw/2.75))/(1), axis=-1) < Y_hw*.5/2.75)
    | ( norm((pts - (-3*X_hw/14,+Y_hw/2.75))/(1), axis=-1) < Y_hw*.5/2.75)
    | (
        ( norm((pts - (0,-Y_hw/2.75))/(1,.55), axis=-1) < X_hw/2)
        & (pts[:,1] < -50)
    )
)[0]
```

(continues on next page)

[236] https://en.wikipedia.org/wiki/Nearest-neighbor_interpolation

(continued from previous page)

```
input_stim_delay_msec = 20; input_stim_duration_msec = 200+50*rng.
↪uniform(size=len(input_stim_idxs));

neuron_Vrest_mV = ( -70
    + 30* (norm(pts - (0,0), axis=-1) < Y_hw*.95)
    + 10*rng.normal(size=neurons))
neuron_Vrest_mV[input_stim_idxs]=-70 # Don't differentiate the

syn_tau_msec = 0.1+10*rng.uniform(size=len(syn_list)); max_syn_tau_msec = np.
↪max(syn_tau_msec)

# And display
fig, ax = plt.subplots(figsize=(16,10)); ax.set_aspect('equal')
for idx,(i,j) in enumerate(syn_list):
    plt.arrow(*pts[i], *pts[j]-pts[i], color=(0.9*(syn_tau_msec[idx]/max_syn_tau_
↪msec),0,0),
              head_width=8, width = 1,lw = 0, length_includes_head=True)

plt.scatter(*neuron_points_microns[input_stim_idxs].T, color='xkcd:sky', s=225,␣
↪lw=0, )
plt.scatter(*neuron_points_microns.T, c=neuron_Vrest_mV, s=100, lw=0)
ax.set_facecolor('#f0f4ff')
```



There's a network with variability all over the place.

Now let's *generate* the model file using f-strings[237] and list comprehensions[238]:

[237] https://docs.python.org/3/reference/lexical_analysis.html#f-strings
[238] https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

```
tabline = '\n    ' # annoyingly enough, \ may not exist at all in a f string, not␣
↪even in brackets. Fixed in Python 3.12+
model_file = f'''
<neuroml>
<izhikevich2007Cell id="IzhCell" v0 = "-60mV" C="100 pF" k = "0.7 nS_per_mV"
                    vr = "-60 mV" vt = "-30 mV" vpeak = "35 mV"
                    a = "0.03 per_ms" b = "-2 nS" c = "-50 mV" d = "100 pA"/>
<pulseGenerator id="DcProbe" delay="{input_stim_delay_msec} ms" duration="0 ms"␣
↪amplitude="0.5nA"/>
<expOneSynapse id="ExpCondSynapse" gbase="8.1nS" erev="20mV" tauDecay="5ms"/>
<network id="Net" type="networkWithTemperature" temperature="37degC" >

    <population id="Pop"  component="IzhCell">
    {tabline.join([ f'<instance id="{i}"><location x="{x}" y="{y}" z="{0}"/></
↪instance>' for i, (x,y) in enumerate(pts)])}
    </population>

    <inputList id="Inp" population="Pop" component="DcProbe">
    {tabline.join([ f'<inputW id="{i}" target="Pop[{idx}]" destination="synapses"␣
↪weight="1"/>' for i, idx in enumerate(input_stim_idxs)])}
    </inputList>

    <projection id="GridProjection" presynapticPopulation="Pop"␣
↪postsynapticPopulation="Pop" synapse="ExpCondSynapse">
    {tabline.join([ f'<connectionWD id="{ii}" preCellId="{pre}" postCellId="{post}
↪" weight="1" delay="5 ms"/>' for ii, (pre,post) in enumerate(syn_list)])}
    </projection>

</network>
<Simulation id="MySim" length="1 s" step="100 us" target="Net"> <!-- no need for a␣
↪seed yet, add one if you want ! -->
    <OutputFile id="MyOutFile" fileName="results.gen.txt">
    {tabline.join([ f'<OutputColumn id="v_{i}"  quantity= "Pop[{i}]/v"/>' for i, _␣
↪in enumerate(pts)])}
    </OutputFile>

    <!-- Use EdenCustomSetup ! -->
    <EdenCustomSetup filename="Example_CustomSetup.txt"/>
</Simulation>
<Target component="MySim"/></neuroml>
'''
# print(model_file)
with open('Example_CustomSetup.nml', 'wt') as f: f.write(model_file)
```

And the `Example_CustomSetup.txt` that goes with it:

```
setup_file = ''
setup_file += 'set cell Pop all all vr multi mV\n'
setup_file += 'values '+ ' '.join(f'{x}' for x in neuron_Vrest_mV)+'\n'

setup_file += 'set synapse GridProjection all post tauDecay multi msec\n'
setup_file += 'values '+ ' '.join(f'{x}' for x in syn_tau_msec)+'\n'

setup_file += 'set input Inp all duration multi msec\n'
setup_file += 'values '+ ' '.join(f'{x}' for x in input_stim_duration_msec)+'\n'
# print(setup_file)
with open('Example_CustomSetup.txt', 'wt') as f: f.write(setup_file)
```

And run the simulation:

```
from eden_simulator import runEden
results = runEden('Example_CustomSetup.nml')
```

(continues on next page)

(continued from previous page)

```
neuron_waveforms = np.array([ results[f"Pop[{i}]/v"] for i, _ in enumerate(pts) ])
# plt.imshow(neuron_waveforms, interpolation='none', aspect='auto');plt.colorbar()
↪#plt.clim((-0.07, -0.04))
# plt.plot(neuron_waveforms[input_stim_idxs].T);
```

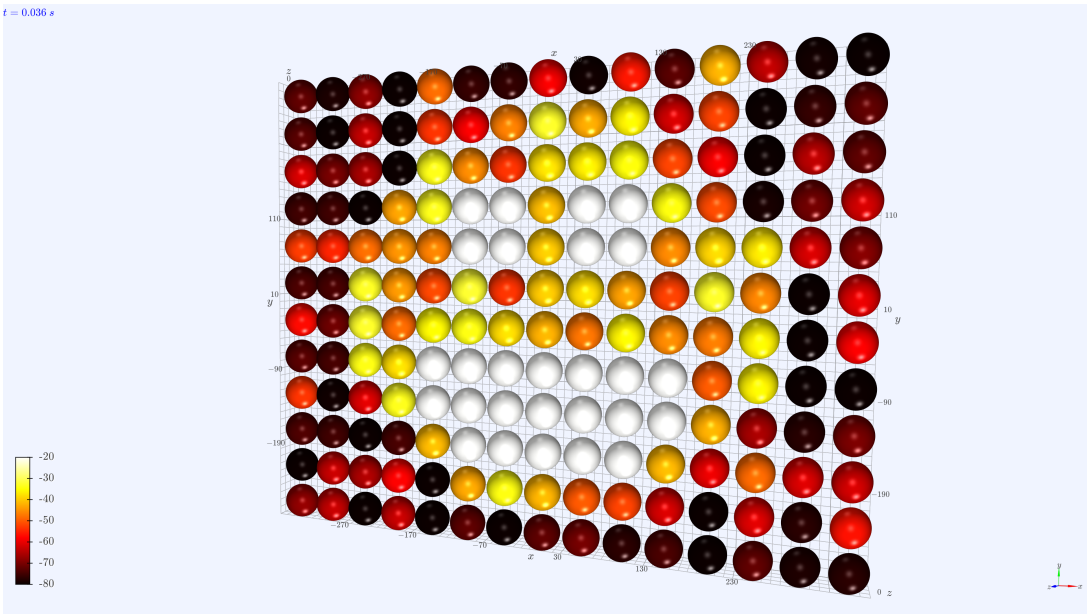
Let's see the neuron activity in pretty 4-D (surprisingly more compact than matplotlib animations):

```
from eden_simulator.display.animation import subsample_trajectories
samples, anim_axis, sampled_time_axis, [sampled_soma_voltage] = subsample_
↪trajectories(
    results['t'], [neuron_waveforms.T * 1000], animation_speed=0.0096, animation_
↪frames_per_second=48)
```

```
!pip install -q k3d
```

```
import k3d; from eden_simulator.display.spatial.k3d import Plot
points_plot = plot = Plot(background_color=0xf0f4ff, camera_auto_fit=False) # to␣
↪override camera orientation

k3d_anim_dict = { str(real_time): x.astype(np.float32)
                 for (real_time, x) in zip(anim_axis, sampled_soma_voltage)  }
k3d_label_dict = { str(real): f't = {sim:.3f} ~ s' for (real, sim) in zip(anim_
↪axis, sampled_time_axis)  }
plt_points = k3d.points(positions=np.c_[pts,[0]*len(pts)].astype(np.float32),
↪attribute=k3d_anim_dict,
        color_range=[-80, -20],point_size=30,color_map=k3d.matplotlib_color_maps.
↪Hot); plot += plt_points

plot.camera = plot.get_auto_camera(pitch=30, yaw=10)[:6]+[0,1,0] # set 'y' to up !␣
↪vecs are pos, tgt, up
plt_label = k3d.text2d(k3d_label_dict, (0.,0.), label_box=False); plot += plt_
↪label # add 2d elements AFTER setting auto camera
plot.fps = 48;
from IPython.display import display, HTML, IFrame
plot.snapshot_type = 'inline'; display(HTML(plot.get_snapshot()))
```

```
<IPython.core.display.HTML object>
```



Note that this a very simple network, and its behaviour is quite sensitive to the parameter values. Since it lacks inhibitory synapses, the balance between full and absent firing is roughly the balance between:

(I_{syn} *times* τ_{syn} times synaptic out-degree) versus (I_{leak} *times* synaptic delay)

6.2.1 More examples

As explained above (page 90), setting parameters over the extent of spatially-detailed cells can slightly more involved. This is shown in a follow-up example (page 99).

The usage of `set` statements for `<VariableReference>`s is shown in the `<EdenTimeseriesReader>` example (page 116).

CHAPTER

SEVEN

ARBITRARY PARAMETERS ON SPATIALLY DETAILED CELLS

It is often the case that biophysical properties *vary spatially* across the spatial extent of a neuron. NeuroML has a provision for modelling the variability of the base conductance($\bar{g}$) of ion channel distributions, as a user-specified function of distance from the soma (measured along neurites). Still, modellers may desire more types of variability, and on different parameters as well. This article shows how Eden's `<CustomSetup>` (page 89) extension can be used to model any kind of variability, on any attribute of a mechanism present on spatially-detailed cells (page 33).

As seen on the main article (page 89) about this extension, the model attributes of neurons can be customised in any conceivable way through `set cell` statements (page 90). For point neurons, the *location list* is always `all`. For spatially-detailed cell models, the list can be much more specific - and we'll see right now how to use it.

Note

This feature is still in development.

First, let's get a sample cell to try variability on:

```
nmldb_cell_name = 'NMLCL000001'; zip_file_name = f'{nmldb_cell_name}.zip'
# Download the zip file with the model
import urllib.request
# Because NMLDB is hoving some issues with HTTPS, don't demand HTTPS verification
import ssl; ssl._create_default_https_context = ssl._create_unverified_context
urllib.request.urlretrieve(f'http://neuroml-db.org/GetModelZip?modelID={nmldb_cell_
↪name}&version=NeuroML', zip_file_name)
# and unpack it
from zipfile import ZipFile
with ZipFile(zip_file_name, 'r') as zipp: zipp.extractall(nmldb_cell_name+'/')
# Find the cell's .nml file
import os
cell_filenames = [ nmldb_cell_name+'/'+name for name in os.listdir(nmldb_cell_
↪name) if name.endswith('.cell.nml') ]
assert len(cell_filenames) == 1, "Didn't find a lonely cell nml file!"; cell_
↪filename = cell_filenames[0]
```

7.1 Sampling the cell's morphology

Let's get some facts about the neuron on which we'll define variability:

```
import eden_simulator
cells_info = eden_simulator.experimental.explain_cell(cell_filenames[0])
# one cell in the model file, get it
cell_info = list(cells_info.values())[0]
nComps = len(cell_info['comp_midpoint'])
comp_midpoint = cell_info['comp_midpoint']
# list(cell_info.keys())
```

As mentioned in a previous chapter (page 33), spatially-modelled cells eventually have to be split into discrete *compartments*, within which every biophysical quantity is considered to have the same value. Hence, spatially-varying quantities on a modelled cell may be assigned values that differ for only individual compartments.
Note: This is the way that *any* distribution, deterministic or stochastic, of parameters can be faithfully applied on neurons; the particular values are entirely controlled by the modeller.

Given the information from `explain_cell`, it is straightforward to write the `set cell` statement: Set the *location list* field as a comma-separated list of `seg.fractionAlong` cell locators (page 87), and add as desired for variability, `multi` (to `set` one cell) or `multi_location`, `multi_cell` or `multi_multi` (for multiple cells in a go), and the `values` lines that follow.

Note

When overriding parameters over spans of the neuron, make sure to *not* miss any spots (especially for `<VariableRequirement>` (page 93)s!). A reliable way is using explain_cell (page 241) as shown here to get `cell-info`the list of compartments and their `segment+fractionAlong` midpoints, and the list of compartments per segment group of interest as `cell_info['segment_groups'][groupName]`.

The following examples will show how to implement different frequently used distributions for biophysical parameters. In all cases, variability within a cell boils down to a vector of values, one for each compartment. Here's a helper routine to write a CustomSetup (page 90) file, that `sets` a `cell` with different values all over:

```
def set_cell_vals(cell_info, popname, instances, attribute, vals, units):
    locs = ','.join([str(loc) for loc in eden_simulator.experimental.
↪GetLemsLocatorsForCell(cell_info)])
    lines = f'set cell {popname} {instances} {locs} {attribute} multi {units}\n'\
        'values '+'\t'.join(map(str,vals))+'\n'
    return lines
```

In the following, we'll use these routines to visualise the different cases of variability over the neuron. We'll use PyVista[239] for 3-D graphics, and EDEN's helper pyvista.plot_neuron (page 241) to colour a mesh per compartment.

```
import pyvista as pv

def plot_neuron(cell_info, vals, valsname, cmap=None, bgcol=None):
    import pyvista as pv
    from eden_simulator.display.spatial.pyvista  import get_neuron_mesh
    mesh = get_neuron_mesh(cell_info)
    mesh[valsname] = vals

    pl = pv.Plotter();
    pl.add_mesh(mesh, clim = None,cmap=cmap)
    pl.background_color = "#fefefe"
    if bgcol: pl.background_color = bgcol

    pl.camera.position = (000, 150, +1050)
    pl.camera.focal_point = (0.2, 0.3, -5.3)
    pl.camera.up = (0.0, 1., 1.)
    pl.camera.zoom(.8)
    # pl.camera.position = (1000, 150, -50)
    # pl.camera.focal_point = (0.2, 150.3, -50.3)
    # pl.camera.up = (0.0, 1.0, 0.6)
    # pl.camera.zoom(.8)
    return pl, mesh
def show_plot(pl):
    # displays won't show except with this trick... https://github.com/pyvista/
↪pyvista/pull/5168
```

(continues on next page)

[239] https://pyvista.org

(continued from previous page)

```
    scene = pl.show(jupyter_backend="html", return_viewer=True)
    # but scene.value is not present on older versions ...
    # s = scene.value if hasattr(scene,'value') else ''
    # as seen on https://github.com/mikedh/trimesh/blob/4.5.1/trimesh/viewer/
↪notebook.py#L72
    height=500
    s = pl.export_html(None).read()
    import html; s = html.escape(s)
    s = f'<div><iframe srcdoc="{s}" \
    width="100%" height="{height}px" \
    style="border:none;"></iframe></div>'
    import types; scene_wrapper = types.SimpleNamespace( # note: __repr__ needs an␣
↪actual class to override
        scene=scene, _repr_html_=lambda : s, _repr_latex_=lambda : '')
    from IPython import display; display.display(scene_wrapper)
    return scene_wrapper
# pl,_ = plot_neuron(cell_info, vals, 'Distance from plane (µm)','reds','azure')
# Plot_Vert = pl; Show_Vert = show_plot(pl);
```

7.2 Variability as distance from a plane

One may wish to model a quantity with a *lateral* distribution over the neuron; that is to say, a quantity whose value is increasing (or decreasing) towards a certain *direction*. Let $\hat{\mathbf{n}}$ be the unit vector[240] pointing to that direction.

Then one can assume a signed *directional distance metric* from a reference point $\mathbf{p}_0$, for every point in space $\mathbf{x}$, as follows:

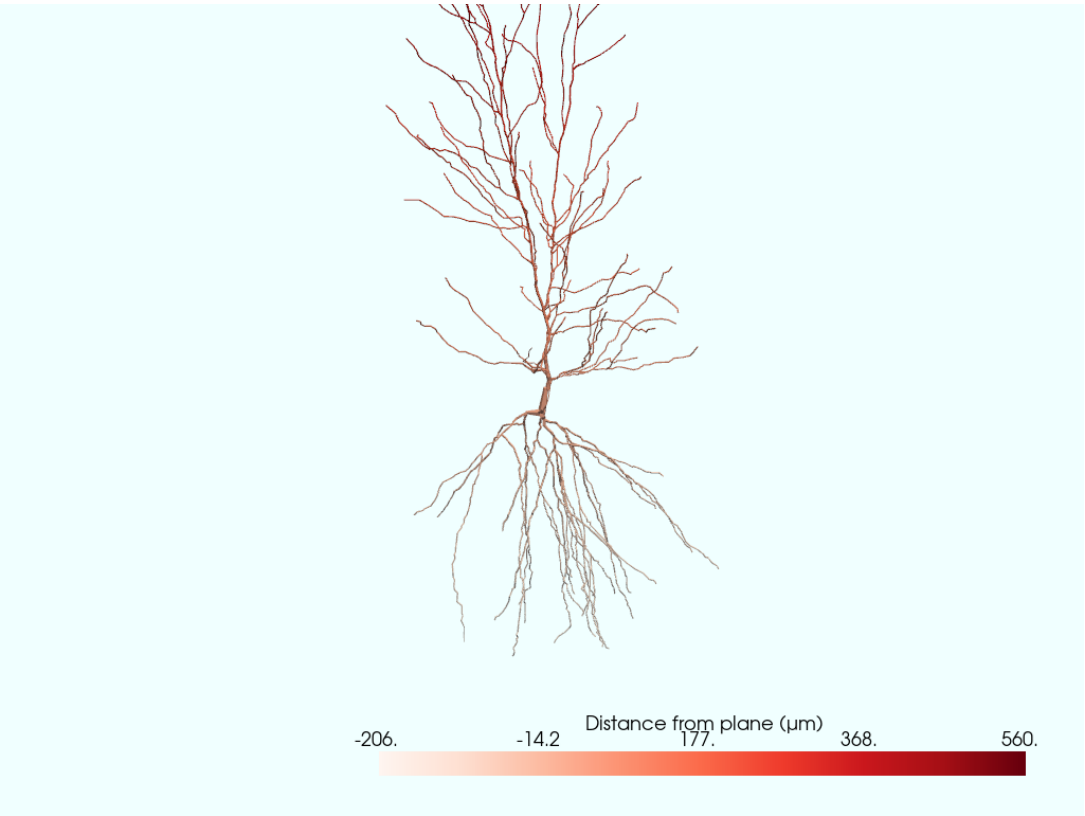
$$d_{\hat{\mathbf{n}},\mathbf{p}_0}(\mathbf{x}) = (\mathbf{x} - \mathbf{p}_0) \cdot \hat{\mathbf{n}}$$

which can be converted to "quantity" units as desired. Note that d has the same value over every *plane* perpendicular to $\hat{\mathbf{n}}$.

Let's sample d for the midpoint of each compartment and see the result.

```
import numpy as np
from numpy.linalg import norm
def unit_vec(v): return v / norm(v, axis=-1)
def dist_by_plane(point_0, dist_direction, sample_points):
    n = unit_vec(dist_direction)
    return (sample_points - point_0).dot(n)
# Try n = [1,0,0] also
vals = dist_by_plane([0,0,0], [0,1,0], comp_midpoint)
pl,_ = plot_neuron(cell_info, vals, 'Distance from plane (µm)','reds','azure')
Plot_Vert = pl; Show_Vert = show_plot(pl);
```

[240] https://en.wikipedia.org/wiki/Unit_vector



7.3 Variability as distance from an axle

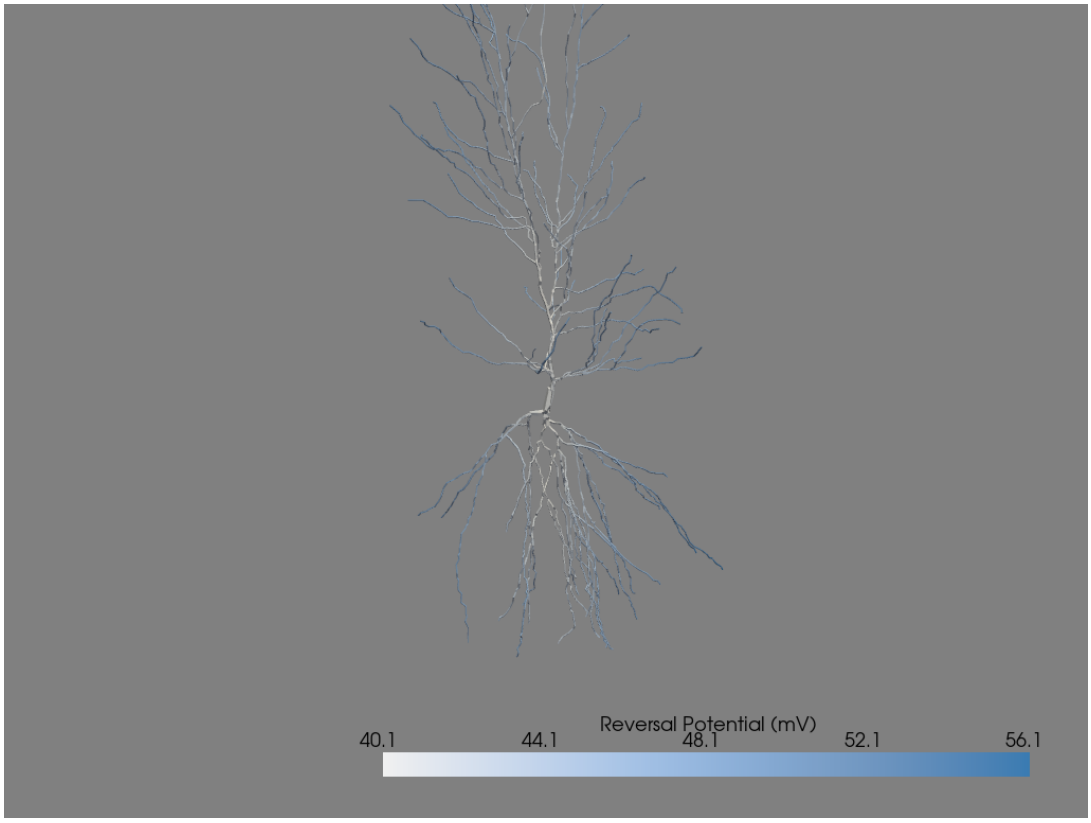
Another sort of spatial distribution is as a function of *transverse distance* from an axle (designated by a reference point $\mathbf{p}_0$ on the axle and a direction vector $\hat{\mathbf{n}}$). The *transverse* distance vector is thus:

$$\mathbf{d}_{\perp,\hat{\mathbf{n}},\mathbf{p}_0}(\mathbf{x}) = \mathbf{r} - (\mathbf{r}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}},\ \mathbf{r} = \mathbf{x} - \mathbf{p}_0$$

in which case, the distance is the same over each *tube* centered at the axle.

Let's distribute the *reversal potential* E_{Na} of the ion channel distribution `Na_all` as a function of that, then:

```
def dist_by_axle(p0, axle_direction, x):
    n = unit_vec(axle_direction)
    d = (x - p0)
    aligned =  d.dot(n)[:,None]*n[None,:]
    return norm(d - aligned, axis=-1)

vals = dist_by_axle([0,0,0], [0,1,0], comp_midpoint) * (10/100) + 40
pl,_ = plot_neuron(cell_info, vals, 'Reversal Potential (mV)','blues','gray')
with open('cell_axle.txt', 'w') as f:
    f.write(set_cell_vals(cell_info, 'pop', 'all', 'biophysicalProperties/
↪membraneProperties/Na_all/erev', vals, 'mV'))
Plot_Axle = pl; Show_Axle = show_plot(pl);
```

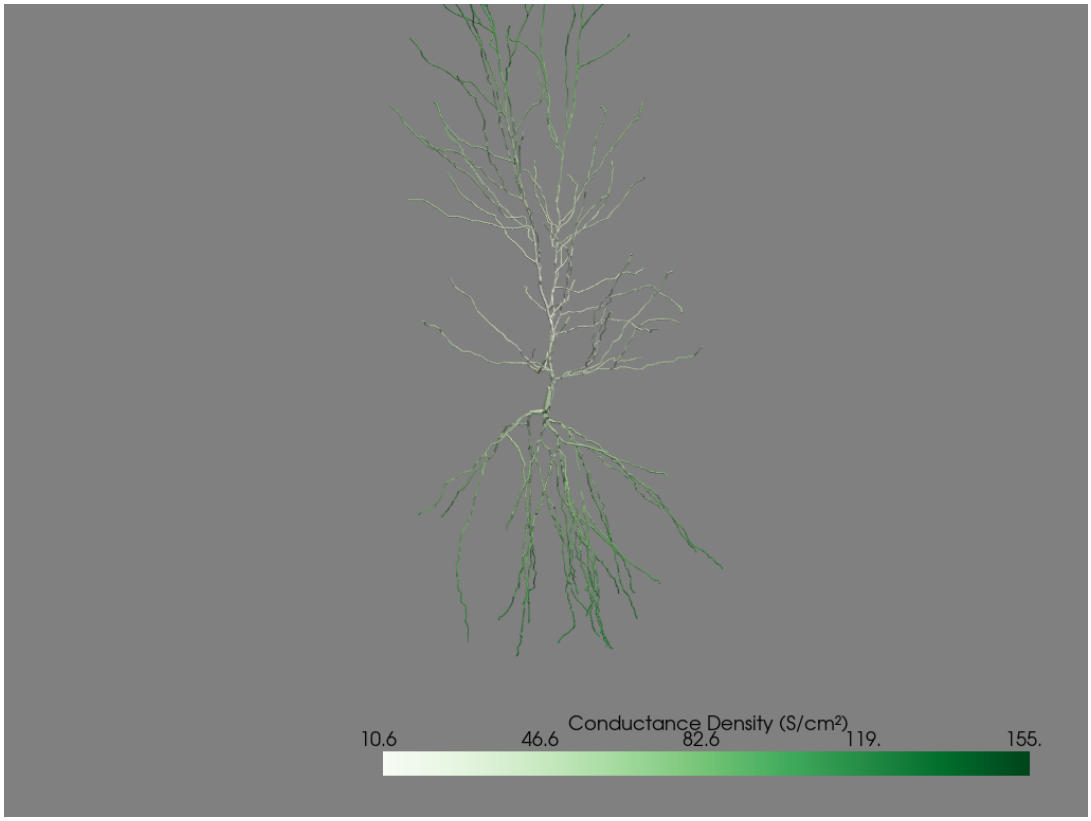


7.4 Variability as by distance from a point in space

A simpler type of distribution is *radial distance* from a reference point $\mathbf{p}_0$, that's simply $\mathbf{d} = \mathbf{x} - \mathbf{p}_0$. In this case, the distance is the same over each *sphere* centered at $\mathbf{p}_0$.

Let's distribute the *base conductivity* $\bar{g}_{Na}$ of `Na_all` as a function of euclidean distance from a reference point:

```
def dist_by_point(p0, x):
    return norm(x - p0, axis=-1)

vals = dist_by_point([0,100,0], comp_midpoint) * (3/10) + 10
pl,_ = plot_neuron(cell_info, vals, 'Conductance Density (S/cm²)','Greens','gray')
with open('cell_dist.txt', 'w') as f:
    f.write(set_cell_vals(cell_info, 'pop', 'all', 'biophysicalProperties/
↪membraneProperties/Na_all/condDensity', vals, 'S_per_cm2'))
Plot_Dist = pl; Show_Dist = show_plot(pl);
```



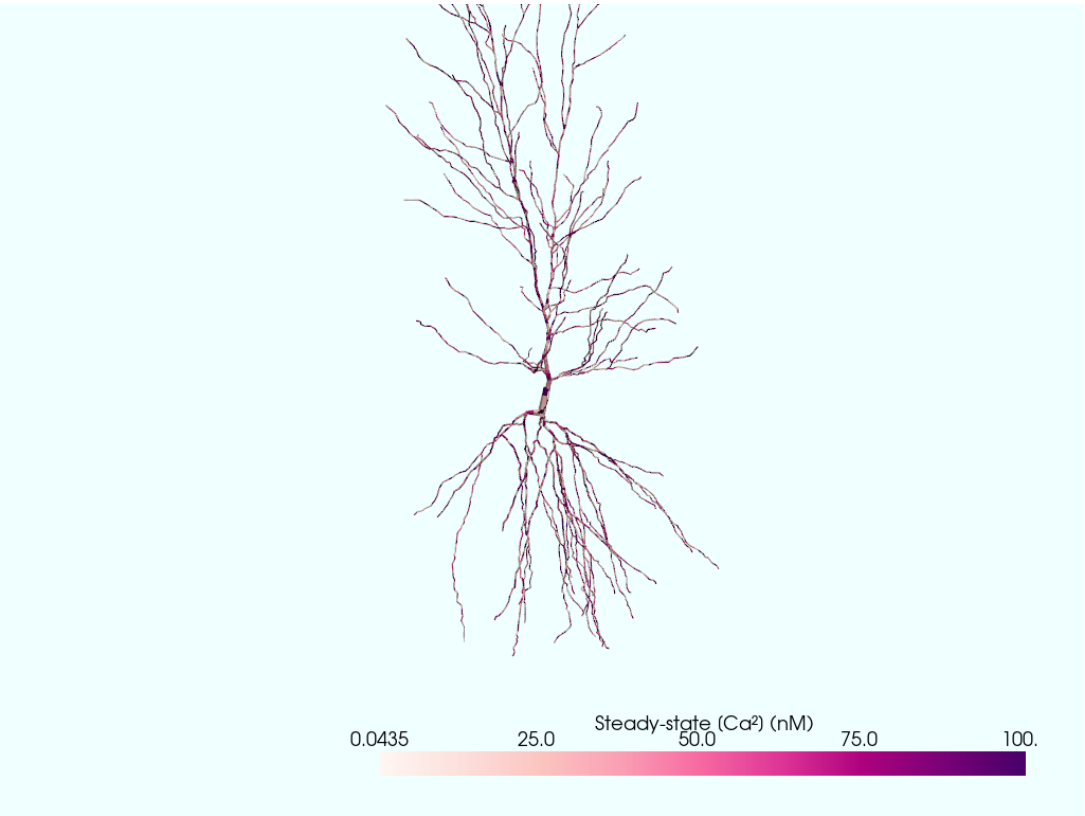
7.5 Randomised parameters

Finally, a quantity may simply exhibit a slight variation which could follow any random distribution. As mentioned in the previous article (page 89), it's often better to assume some variation across neuron and mechanism instances, though this will also require more memory to store the variability.

For example, the *steady-state concentration* of a <decayingPoolConcentrationModel>[241] used for a <species> distribution (page 51) may vary following an uniform distribution:

```
rng = np.random.default_rng(seed=43264326)
vals = rng.uniform(0,100,nComps)

pl,_ = plot_neuron(cell_info, vals, 'Steady-state [Ca2+] (nM)','RdPu','azure')
with open('cell_rand.txt', 'w') as f:
    f.write(set_cell_vals(cell_info, 'pop', 'all', 'biophysicalProperties/
↪intracellularProperties/ca_conc_all/restingConc', vals, 'nM'))
Plot_Rand = pl; Show_Rand = show_plot(pl);
```

[241] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#decayingpoolconcentrationmodel



Beside the above suggestions, a distribution may depend on multiple factors, always following the intent of the modeller (or undisputed biophysical findings). By evaluating the values of quantities for each compartment, distributions can be implemented in the simulation with full precision.

7.6 Aside for NEURON users

Readers familiar with the NEURON simulator may have assigned spatial variability for parameters over the cell, with `SubsetDomainIterator`[242] or in the general case through HOC code (or the equivalent Python):

```
forsec <section_name or all>{
    for(x){
        // x is fraction along section for the compartment, and area(x), ri(x), x,
↪y,z3d(x) and others are properties of the compartment.
        // Users can then set RANGE properties however they like based on the␣
↪above properties.
    }
}
```

As shown, EDEN provides a similar mode of setting variability, albeit in a 'flatter' way. `explain_cell` provides a list of per compartments and associated quantities over the whole cell, thus *one* loop over the explicitly listed compartments equates to `forsec ... for(x) ...` in HOC. Variability over a section is then done by iterating only over the appropriate subset of compartments, provided by `cell_info['segment_groups'][groupName]` for the section's named `<segmentGroup>`.

[242] `https://nrn.readthedocs.io/en/8.2.6/guide/cellbuilder3.html`

CHAPTER

EIGHT

LEMS EXTENSION: <VARIABLEREQUIREMENT>

As we have seen in Introduction to LEMS (page 61), for each type of mechanism, there are certain quantities that it can access, whose values are provided by the nearby environment as per the NeuroML formulation. However, we (will always) need more to model uncommon, or *entirely novel* interactions in our mechanisms. Hence, we need a way for mechanisms to be possibly influenced by any *arbitrary quantity* that exists in our model.

EDEN makes this possible through the `<VariableRequirement>` tag. This tag may be present inside a LEMS component (page 61), among the standard `<Requirement>`s that are fulfilled in the classic NeuroML way.

The value that a `<VariableRequirement>` provides at any point in time is that of its *target*. That could be any state variable, that's specified through a LEMS path (page 87).

Tip: This concept is also known as `POINTER` s when using the NEURON simulator, and as `linked` variables when using BRIAN. But there is a difference as follows:

- Under EDEN's rules, the quantities can be *observed* but not *modified* directly, since that could cause unpredictable results between users of the variable. If modifying an observed variable is needed nonetheless, this can be achieved by firing spikes to synaptic components, that may then modify the same value through `<WritableRequirement>` (page 123). (By the way, the same 'read-only' rule applies for CoreNEURON among other high-speed simulators.)

8.1 Usage

After specifying a `<VariableRequirement>` in a LEMS mechanism, all that remains to resolve its *target*. Since they exist beyond the classic LEMS specification, and any value that they could have depends on the model that they exist in, the target has to be specified for every single instance using `<EdenCustomSetup>` (page 89). Instead of a numerical value with units, a LEMS path (page 87) is assigned as the *target* (immediate "value" in a sense) of every `<VariableRequirement>` property that exists in the model.

Note: Any `<VariableRequirement>` that has not been resolved will yield a `NaN` value, that will propagate to all the state and derived variables that it touches. This may seem frustrating at first, but it ensures that *all* references are accounted for; lax tracking of references has often been a cause of difficult to trace mistakes in the past.

In later versions, EDEN may thoroughly scan for unresolved `<Reference>`s and refuse to run the model until they are resolved, instead reporting the locations of missing references to the user.

8.2 Examples

As <VariableRequirement>s by their nature get to be used in highly model-specific ways, it is not so easy to single out one simple and general example of use. However, they are instrumental in the functioning of many models that are included in this guide. The interested user may refer to, among others, these straightforward uses:

- Accessing the variables of a <EdenTimeSeriesReader> input stream (page 116).
- Exchanging the ball and paddle positions with the requested control forces between the playing field and the players, in the Pong example (page 177).

CHAPTER

NINE

EXTENDED INPUT-OUTPUT OPTIONS

Neural structures are fascinating and very interesting to observe even in isolation, but they are most useful when there is a way to externally influence them and observe the resulting differences in behaviour. To cover this need, NeuroML offers an extensive set of `<input>`s that can be modelled and applied on a `<network>`, and the `<OutputFile>` specification format to record and observe the simulated quantities.

However, the scientifically interesting set of input sources can often exceed what NeuroML can model through its dynamical equations (for example, experimental captures of real phenomena), and the `<OutputFile>` specification is also impractical for large-scale recording and real-time processing. Furthermore, experimental setups may include some sort of external environment (be it another brain region, organ or the organism's surroundings) that the simulated model *interacts* with. There are cases where the full system and interaction can be captured inside a NeuroML model (see the Pong (page 173) example) but again all kinds of interaction cannot be expressed within NeuroML plus LEMS.

This article introduces EDEN's extended capabilities for getting data in and out of the simulation, beyond what NeuroML already offers.

The descriptions and usage examples for each of EDEN's extended input/output capabilities follows:

- General concepts (page 109)
- Extended output:
  - Time series with `<EdenOutputFile>` (page 110)
  - Event series with `<EdenEventOutputFile>` (page 114)
- Extended input:
  - Time series with `<EdenTimeSeriesReader>` (page 116)
  - Event series with `<EdenEventSetReader>` (page 118)

9.1 General concepts

EDEN receives information from and provides information to other parts of the computer it's running on, in the form of streams[243]. These streams are sequences of bits of information (of possibly unknown length), that flow in strict (simulation) time order. Different types of transmitted information may always be cast to streams (for example, independent packets can be serialised into time series, with information imputed for the lost packets as appropriate). This is done with auxiliary programming that's specific to each experimental apparatus, and is thus not part of EDEN but part of the experiment.

These streams of information within EDEN may take one of two forms: they can be either:

- *Time series*, which are observed as quantities changing values over time (for example, an electrical potential or dopamine eligibility trace); or
- *Event series* which are instances of an event that occur at specific points in time (for example, action potentials from outside the simulated network, other instantaneous signals).

[243] `https://en.wikipedia.org/wiki/Streaming_media`

EDEN's extended I/O tags commonly use two attributes, `href` and `format` to specify *where* (in computer access space) and *how* (in form of expression) to exchange data. Due to the technologies involved, these two parameters are not entirely independent. The values of these attributes are as explained below:

9.1.1 Data URL's

The URI scheme[244] (the `<scheme>://` part of a URL) determines *how* an information resource should be found and accessed (common examples are `http`, `tel` and `zoom`™). The rest of the URL that follows is decoded and retrieved following that scheme, so we have to specify it first.

The available options for source/destination URI's and their meanings are:

- *no scheme*: A regular file, with a path relative to (i.e. starts from the same point as) the file the URL is written on. The meaning is the same as the `fileName` in classic NeuroML.
- `file://`: A regular file, with a path relative to the *current working directory* (e.g. the notebook that's being used, the folder open when clicking on a program, or what the command line starts with, try `os.getcwd()` if in doubt).

The URI schemes `tcp://`, `pipe://`(named pipes for Windows), `unix://`(Unix domain pipes), and `mpi://` are planned to be supported soon.

9.1.2 Data format

The options for the *format of data* transmitted to/from the specified URL are:

- `ascii_v0`: The first text-based format type, that closely resembles the de facto NeuroML output format. It's human-readable text files with each sample of the recording on a separate line, that (typically) starts with the simulated time that the sample refers to.

More data formats may be introduced as needed.

9.2 Extended output

9.2.1 Time series with `EdenOutputFile`

In NeuroML, the evolution over time of quantities in the simulated `<network>` can be saved to files and retrieved thanks to `<OutputFile>`s in the `<Simulation>`. Each `<OutputFile>` writes down a time series of one or more physical `quantity`s specified by the `<OutputColumn>`s, throughout the simulation.

A problem that arises is that `<OutputFile>` records the quantities' values for *every single timestep* of the simulation, which very quickly makes it impractical to record many time series over a long simulation duration - even if only few of the points in time that are recorded are actually needed. Another minor issue (which nonetheless adds unnecessary friction) is that quantities are always recorded in SI units[245].

To address these limitations, EDEN offers an alternative form of `<OutputFile>` fittingly called `<EdenOutputFile>`. Along with the general `href` and `format` mentioned above (page 109), this tag also accepts more parameters to get a nicer looking output.

When using this tag, each `<OutputColumn>` that records a dimensioned `quantity` must also specify the `output_units` to record the quantity in. This helps with ambiguity and can also save some unit conversion when the experimental pipeline uses non-SI units already.

More importantly, one can specify *when* to record using this tag. There are two ways to specify the sampling points in time:

[244] `https://www.w3.org/2001/tag/doc/SchemeProtocols.html`
[245] `https://en.wikipedia.org/wiki/SI_derived_unit`

- either *periodically* over an interval,
- or at some *explicitly specified* points in time.

The first way involves specifying the following attributes on the `EdenOutputfile` tag:

- `starting_from` (default: 0): The point in (simulation) time to start recording from.
- `up_to_excluding` (default: ∞): The point in time when recording ends.
- `sampling_interval` (default: simulation step) The sampling period (interval between samples).

As an alternative to specifying a recording interval and sampling period (or going ahead with the default values), one may instead add the child tag `<SamplePoints>`. This tag must contain one or more tags in the form `<st="time to record">`, and the time values must be in strictly increasing order.

Example: Reducing the size of simulation output

Let's take the first single-neuron model from Introduction (page 13) and record it as usual:

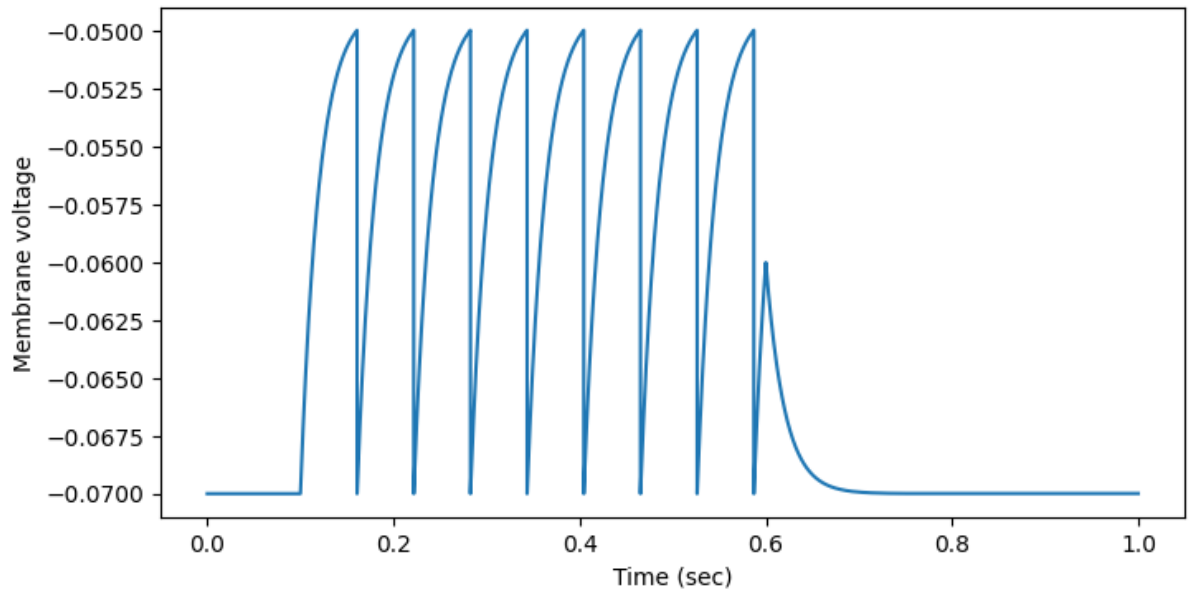
```
%%writefile SingleCell_Model.nml
<neuroml>
<iafCell id="MyFirstCellType" C="200 pF" leakConductance="10 nS" leakReversal="-70␣
↪mV" reset="-70mV" thresh="-50mV" />
<pulseGenerator id="MyFirstInputSource" delay="100ms" duration="500ms" amplitude=
↪"0.21nA"/>
<network id="MyNetwork" type="networkWithTemperature" temperature="37degC" >
    <population id="Pop" component="MyFirstCellType" size="1"/>
    <inputList id="MyFirstInputList" population="Pop" component="MyFirstInputSource
↪">
        <input id="0" target="Pop[0]" destination="synapses"/>
    </inputList>
</network>
</neuroml>
```

```
Writing SingleCell_Model.nml
```

```
%%writefile LEMS_RecordLots_Sim.xml
<?xml version="1.0" encoding ="UTF-8" ?>
<Lems>
<include file="SingleCell_Model.nml" />
<Simulation id="MySim" length="1 s" step="10 us" target="MyNetwork" >
    <OutputFile id="MyFirstOutputFile" fileName="results_more.gen.txt">
        <OutputColumn id="my_first_vm" quantity="Pop[0]/v"/>
    </OutputFile>
</Simulation>
<Target component="MySim"/>
</Lems>
```

```
Writing LEMS_RecordLots_Sim.xml
```

```
from eden_simulator import runEden; from matplotlib import pyplot as plt
def plot_single_cell(results, units='V'):
    plt.figure(figsize=(8,4))
    plt.plot(results['t'],results['Pop[0]/v'])
    plt.xlabel('Time (sec)'); plt.ylabel('Membrane voltage');

results = runEden('LEMS_RecordLots_Sim.xml')
plot_single_cell(results)
```

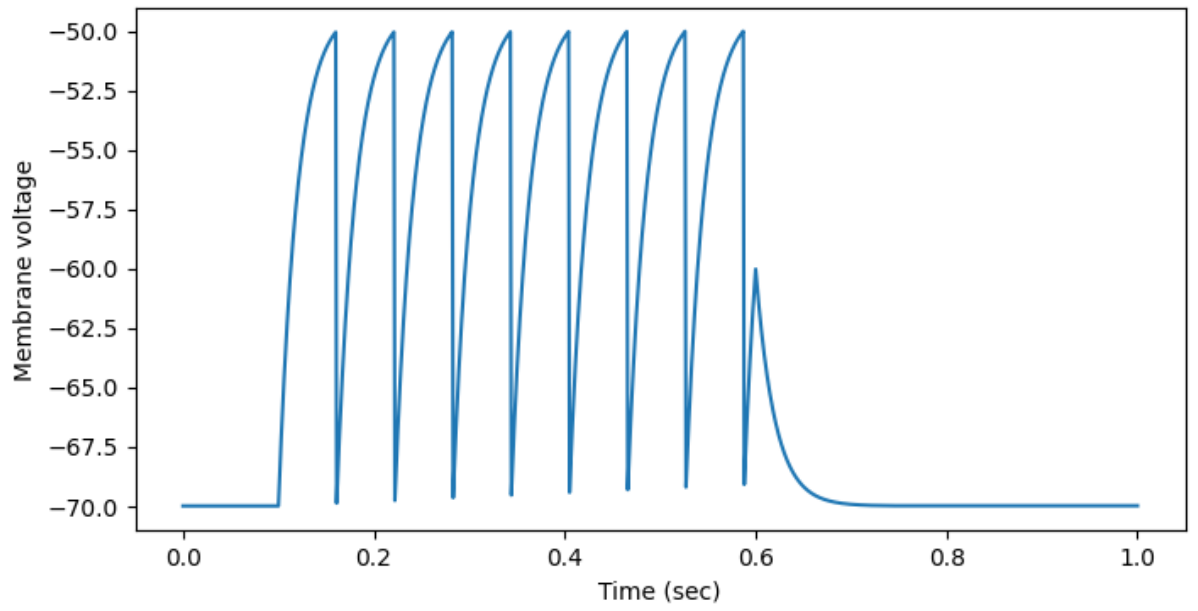


Now, let's use <EdenOutputFile> to record on just every millisecond:

```
%%writefile LEMS_RecordLess_Sim.xml
<?xml version="1.0" encoding ="UTF-8" ?>
<Lems>
<include file="SingleCell_Model.nml" />
<Simulation id="MySim" length="1 s" step="10 us" target="MyNetwork" >
    <!-- Add an EdenOutputFile ! -->
    <EdenOutputFile id="MyOutputFile" href="results_less.gen.txt" format="ascii_v0
↪" sampling_interval="1 msec">
        <OutputColumn id="my_first_vm" quantity="Pop[0]/v" output_units="mV"/> <!--
↪ With units ! -->
    </EdenOutputFile>
</Simulation>
<Target component="MySim"/>
</Lems>
```

```
Writing LEMS_RecordLess_Sim.xml
```

```
results = runEden('LEMS_RecordLess_Sim.xml')
plot_single_cell(results, units='mV')
```



And compare the two recordings in size:

```
import os; from si_prefix import si_format # or humanize or whatever
for x in [ 'results_more.gen.txt', 'results_less.gen.txt' ]:
    print(f'{x}: {si_format(os.path.getsize(x))}bytes')
```

```
results_more.gen.txt: 3.4 Mbytes
results_less.gen.txt: 34.0 kbytes
```

And naturally, this difference is multiplied by the number of quantities that are being recorded.

Example: Recording a specific range in time

Another use case is when someone is not that interested in the whole course of the simulation, but rather on the behaviour that the network converges to (typically near the end of the simulation). In that case, the time range to record over can be specified with `starting_from` and `up_to_excluding`.

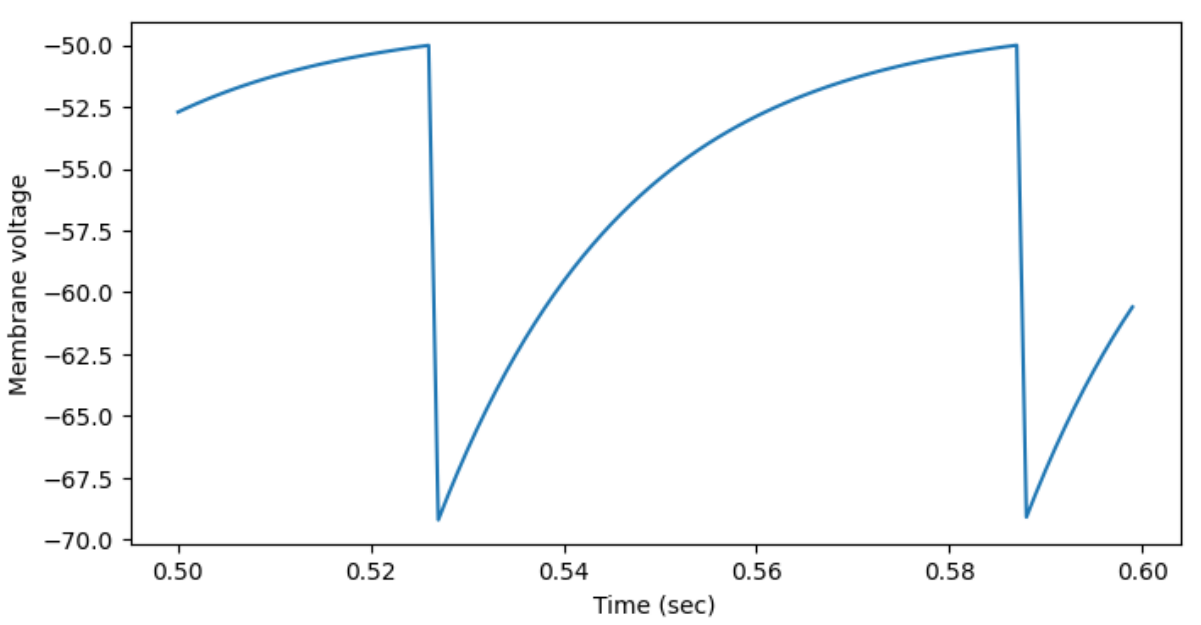
Note: This is *not* recommended to do while the model is still being developed. When one is still not sure about how the network starts off, there is a concern it might have settled to a different (and possibly unrealistic!) state than what's expected. But when there's no such concern any more and only the final phase needs to be recorded, it can also help save space further.

Here's how to record a specific range then, on the above model:

```
%%writefile LEMS_RecordRange_Sim.xml
<?xml version="1.0" encoding ="UTF-8" ?>
<Lems> <include file="SingleCell_Model.nml" />
<Simulation id="MySim" length="1 s" step="10 us" target="MyNetwork" >
    <!-- Add an EdenOutputFile with a specific range ! -->
    <EdenOutputFile id="MyOutputFile" href="results_less.gen.txt" format="ascii_v0"
       starting_from="0.5 s" up_to_excluding="0.6 sec"  sampling_interval="1 msec">
        <OutputColumn id="my_first_vm" quantity="Pop[0]/v" output_units="mV"/>
    </EdenOutputFile>
</Simulation>
<Target component="MySim"/> </Lems>
```

```
Writing LEMS_RecordRange_Sim.xml
```

```
results = runEden('LEMS_RecordRange_Sim.xml')
plot_single_cell(results, units='mV')
```



9.2.2 Event series with `EdenEventOutputFile`

An alternative that NeuroML offers to recording the trajectories of quantities with `<OutputFile>` is to record discrete events that occur (in practice that's basically spikes from neurons, to capture the activity raster). This is done with `<EventOutputfile>` tags in the `<Simulation>`. Each `<EventOutputFile>` writes down a sequence of when and which spikes occurred, so that each occurrence appears as a text line with the time reference and spike source number. Different spike sources are listed by `<EventSelection>`s inside the tag, and identified through their numerical `id`'s.

Although clear and concise, this format has some limitations when the outputs are not just ordinary files, but actually continuous streams that get fed to a post-processing pipeline (and possibly fed back to the simulation through EDEN's input extensions (page 115)):

- When reading a stream line by line, it is not clear whether this is the last spike to occur on the timebase, or more spikes on the same timebase follow. If the stream recipient waits to receive more information, they may have to wait until whenever the next spike appears, and if it doesn't wait and proceeds with processing up to the time base, it may miss more events that happened on the same timestep.
- In the same way, it is not known when the next event follows, and there can be any distance between events. This can be a problem for the spike stream recipient when there are real-time constraints or feedback with the simulator, as post-processing cannot know up to how far it can process the timebase.

To address these limitations, EDEN offers an alternative form of `<EventOutputFile>` fittingly called `<EdenEventOutputFile>`. Along with the general `href` and `format` mentioned above (page 109), this tag also accepts more attributes to control output:

- `starting_from` (default: 0): The point in (simulation) time to start recording from.
- `up_to_excluding` (default: ∞): The point in time when recording ends.
- `maximum_interval` (default: ∞): The maximum time interval between lines.

Note that the `format` parameter is also used in NeuroML, with allowed values `ID_TIME` or `TIME_ID`. For this tag, the `format` can only be `ascii_v0` (more similar to `TIME_ID`).

The `ascii_v0` format expresses the event stream as a sequence of lines, with the difference that all spikes that happened on the same timestep are listed on a single line, and extra lines with no events may be inserted when `maximum_interval` requires it. Each line is therefore in this format:

```
<simulation time> <whitespace-separated list of zero or more ID's>
```

Example: Adding "heartbeats" to an event stream

Let's take the first single-neuron model from the previous example and record its spikes, but also add a `maximum_interval` this time:

```
%%writefile LEMS_RecordSpikes_Sim.xml
<?xml version="1.0" encoding ="UTF-8" ?>
<Lems> <include file="SingleCell_Model.nml" />
<Simulation id="MySim" length="1 s" step="10 us" target="MyNetwork" >
    <!-- Add an EdenOutputFile with a specific range ! -->
    <EventOutputFile id="SpikesOut" fileName="spikes_out.gen.txt" format="TIME_ID">
        <EventSelection id="0" select="Pop[0]" eventPort="spike"/>
    </EventOutputFile>
    <EdenEventOutputFile id="DeLuxe" href="spikes_out_deluxe.gen.txt" format=
↪"ascii_v0"
        maximum_interval="100 ms">
        <EventSelection id="0" select="Pop[0]" eventPort="spike"/>
    </EdenEventOutputFile>
</Simulation>
<Target component="MySim"/> </Lems>
```

```
Writing LEMS_RecordSpikes_Sim.xml
```

```
runEden('LEMS_RecordSpikes_Sim.xml')
with open('spikes_out.gen.txt') as f: lines_before = list(f.readlines())
with open('spikes_out_deluxe.gen.txt') as f: lines_after = list(f.readlines())
from difflib import HtmlDiff; from IPython.display import display, HTML
mydiff = HtmlDiff(tabsize=4)
display(HTML(mydiff.make_file(lines_before, lines_after, fromdesc='EventOutputFile
↪', todesc='EdenEventOutputFile')))
# see also: !diff --side-by-side spikes_out.gen.txt spikes_out_deluxe.gen.txt
```

```
<IPython.core.display.HTML object>
```

These timestamp-only lines may look superfluous, but they may be necessary to go forward in processing pipelines as explained above. Arguably, one could also add a regular `<spikeGenerator>`[246] to that effect. But only `EdenEventOutputFile` can pack events the by timestep, which is also very important.

Exercise: Play with the `starting_from` and `up_to_excluding` attributes and see their effect.

Exercise: Set up a sizeable network that's buzzing with activity and watch how each `EventOutputFile` handles spikes on the same timestep.

9.3 Extended input

EDEN is able to provide a simulation with information streams coming from outside it. This greatly extends the modelling possibilities, since:

- input signals can be created by *any* method and data pipeline, beyond what can be expressed with differential equations;
- the SNN model can be plugged into a *closed loop* experiment, and receive feedback from processes working in a completely different way.

The external input streams to add to a simulation are specified through the `<EdenTimeSeriesReader>` and `<EdenEventSetReader>` tags. Unlike the `OutputWriters`, these are located inside the `<network>` since they effectively are part of it alongside the neurons and synapses.

[246] `https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#spikegenerator`

The form that the information streams take is similar to that of point neurons, in that they can have one or more scalar `Exposure`s and `<OutputPort>`s. This is intentional, as they are meant to interact with parts of the model as if they are regular (albeit *read-only*) objects of the model.

9.3.1 Time series with `EdenTimeSeriesReader`

A `<EdenTimeSeriesReader>` is a source of one or more time-evolving quantities, whose values are retrieved from a URL. These quantities are presented to the model as a collection of *elements*, where each element is uniformly structured as a collection of *named scalars*. One can think of *elements* as pseudo-point-neurons, and of their *named scalars* as LEMS `<Exposure>`s.

Along with the general `href` and `format` mentioned above (page 109), this tag also has the attribute `instances` which represents the number of identical elements that make up the `<EdenTimeSeriesReader>`. Inside the tag, `<InputColumn>` tags list the different *named scalars*, with attributes:

- `id`: The name of the scalar quantity.
- `dimension`: The dimensionality of the quantity, `none` for dimensionless as usual for LEMS.
- `units` (if dimensional): The units for the numerical values of the quantity.

Since input streams are an extension to NeuroML (and they offer many more ways of interaction than, say, `<gapJunction>`s allow), they cannot be interacted with through core NeuroML. Instead, mechanisms can rely on these quantities to adjust their behaviour through `<VariableRequirement>` (page 107)s, which is another extension that EDEN adds to NeuroML. What this means is that if a mechanism's dynamics depend on an external quantity, then this quantity is specified as a `<VariableRequirement>` (page 107) (instead of a local `<Requirement>`) within the LEMS-specified mechanism (page 61).

The `ascii_v0` data `format` to supply `<EdenTimeSeriesReader>`s with is similar to what `<OutputWriter>` already generates:

- There is one line for each sample point in time, followed by all the mentioned time series. Sample points are laid out in strict ascending order. A difference is that the time points for samples don't have to be regularly spaced (just like with `<EdenOutputFile>`'s `SamplePoints`): between sample points, the quantities will retain the values given from the last sample (they will be piecewise constant[247] over time).
- The quantity values for each element are laid along the line in element order. When there is more than one `<InputColumn>` per element, the new value for each of its named scalars is laid out in the order the `InputColumns` were declared. Hence there are `elements * scalars` quantity values per line (plus the timestamp at the start of the line) and the value order is element-major, scalar-minor.

Example: An arbitrary-pulse current probe

The simplest (and popular) example to demonstrate the power of `<EdenTimeSeriesReader>` with is a current probe of a *user-specified waveform*.
First, let's see how to set up stuff model-wise:

```
%%writefile ReadTimeSeries.nml
<neuroml>
<!-- A simple yet versatile current source ! -->
<ComponentType name="RefCurrentSource" extends="basePointCurrent"
    description="Generates instantaneous current sourced from a VariableReference.
↪">
    <!-- Here comes the VariableRequirement ! -->
    <VariableRequirement name="iPointed" dimension="current" />
    <Dynamics>
        <DerivedVariable name="i" dimension="current" exposure="i" value="iPointed
```

(continues on next page)

[247] https://en.wikipedia.org/wiki/Step_function

(continued from previous page)

```
↪"/>
    </Dynamics>
</ComponentType>

<iafCell id="MyFirstCellType" C="200 pF" leakConductance="10 nS" leakReversal="-70␣
↪mV" reset="-70mV" thresh="-50mV" />
<RefCurrentSource id="MyRefInputSource"/> <!-- We'll have to set up a␣
↪VariableRequirement for all instances ! -->

<network id="MyNetwork" type="networkWithTemperature" temperature="37degC" >
    <population id="Pop" component="MyFirstCellType" size="1"/>
    <inputList id="MyInputList" population="Pop" component="MyRefInputSource">
        <input id="0" target="Pop[0]" destination="synapses"/>
    </inputList>

    <!-- Add an input stream ! -->
    <EdenTimeSeriesReader id="MyFirstInputStream"  href="file://timeseries.gen.txt
↪" format="ascii_v0" instances="1" >
        <InputColumn id="i" dimension="current" units="nA"/>
    </EdenTimeSeriesReader>
</network>

<Simulation id="MySim" length="1 s" step="10 us" target="MyNetwork" >
    <EdenOutputFile id="MyOutputFile" href="results_less.gen.txt" format="ascii_v0
↪" sampling_interval="1 msec">
        <OutputColumn id="my_first_vm" quantity="Pop[0]/v" output_units="mV"/>
    </EdenOutputFile>

    <!-- Add EdenCustomSetup to set up the VariableReferences ! -->
    <EdenCustomSetup filename="ReadTimeSeries_CustomSetup.gen.txt"/>
</Simulation>
<Target component="MySim"/>
</neuroml>
```

```
Writing ReadTimeSeries.nml
```

```
%%writefile ReadTimeSeries_CustomSetup.gen.txt
set input MyInputList all iPointed MyFirstInputStream[0]/i # add more inputs and␣
↪see what happens :)
```

```
Writing ReadTimeSeries_CustomSetup.gen.txt
```

Then, cook up a signal to pass to the simulation and save it in the `ascii_v0` format:

```
import numpy as np;
N_samples = 1000; signal_dt_sec = 0.001;
times = np.array(range(N_samples)) * signal_dt_sec
# and then add stuff
signal = 0*times
signal[150:526] = 0.3 * np.sin( (times[150:526]- times[150]) / 0.04 )
signal[650:850] = 0.5 * ( (times[650:850]- times[650]) / 0.2 ) ** 2
# and write it down
with open('timeseries.gen.txt', 'wt') as f:
    for (t,i) in zip(times,signal):
        f.write(f'{t} {i}\n')
```

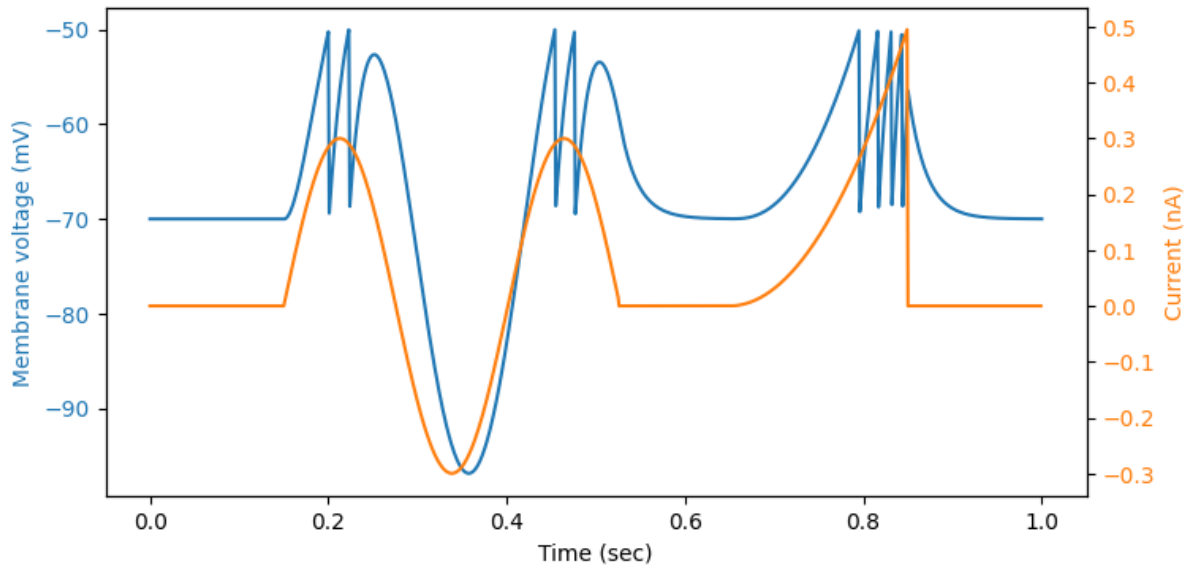
And run it all together:

```
results = runEden('ReadTimeSeries.nml')
def halfaxes(ax, color, label):
    ax.set_ylabel(label, color=color)
    ax.tick_params(axis='y', labelcolor=color)
```

(continues on next page)

(continued from previous page)

```
plot_single_cell(results); halfaxes(plt.gca(), 'tab:blue', 'Membrane voltage (mV)')
ax2 = plt.gca().twinx()
ax2.plot(times, signal, color='tab:orange'); halfaxes(ax2, color='tab:orange',␣
↪label='Current (nA)')
```



Exercise: Try writing a (short) `timeseries.gen.txt` by hand, or set up your favourite waveform and see what happens to the cell.

9.3.2 Event series with `EdenEventSetReader`

An `<EdenEventSetReader>` is a set of one or more spike sources, whose occurrences are retrieved from a URL. The spike sources are presented to the model as a collection of *elements*, where each element is uniformly structured as a collection of *named ports*. One can think of *elements* as pseudo-point-neurons, and of their *named ports* as LEMS `<EventPort>`s.

Along with the general `href` and `format` mentioned above (page 109), this tag also has the attribute `instances` which represents the number of identical elements that make up the `<EdenTimeSeriesReader>`. Inside the tag, `<Port>` tags list the different *named ports*, with no specific attributes.

Even though they are an extension to NeuroML, since the elements resemble neurons with one or more spike emitting sites, they can appear to the model as pseudo-neuron populations (just like ordinary spike sources are) and be directly used in `<projection>`s as pre-synaptic sites!

- When connecting them to post-synapses on real neurons with a `<projection>`, the interpretation of the usual `<connection>` parameters `preCellId` and `preSegment` changes:
  - `preCellId` will stand for the numbered *element* of the `EventSet`;
  - `preSegment` will stand for the *named port* of said element. If unspecified when there are multiple ports, it's assumed to be the typical port `spike`;
  - `preFractionAlong` is not applicable, as is the case with point neurons.
- Of course there may be no synaptic component on their side; hence they can only be the `presynapticPopulation` in a simple, AP based `<projection>`.

The `ascii_v0` data `format` to supply `<EdenEventSetReader>`s with is similar to what `<EdenEventOutputFile>` already generates:

- There is one line for each distinct sample point in time, followed by all the event identifiers that occur on it. Sample points are laid out in strict ascending order.

- A difference with `<EdenEventOutputFile>` is how the multiple ports of elements are identified, when present: instead of a simple integer (that identifies the elements, when each has only one port) there are *two* integers separated by a `!` character.

  In this case:

  - the first number is that of the *element* in the event set;
  - the second number is that of the event `<Port>`, in the order they were listed;
  - both are needed to determine which port of which element emitted an event.

  Do note, however, that `<(Eden)EventOutputFile>` with its simple list of `<EventSelection>`s can't write streams in this "multi-port" format.

Example: An arbitrary-train multi-port spike source

A simple example to show how `<EdenEventSetReader>` works is with a multi-port event stream that projects to different neurons.

First, let's see how to set up stuff model-wise:

```
%%writefile ReadEventSet.nml
<neuroml>
<iafCell id="MyFirstCellType" C="200 pF" leakConductance="10 nS" leakReversal="-70␣
↪mV" reset="-70mV" thresh="-50mV" />
<expOneSynapse id="MySyn" gbase="11nS" erev="0V" tauDecay="5ms"/>
<network id="MyNetwork" type="networkWithTemperature" temperature="37degC" >

    <!-- Add a population with two cells -->
    <population id="Pop" component="MyFirstCellType" size="2"/>

    <!-- Add an event stream ! -->
    <EdenEventSetReader id="MyFirstEventStream" href="eventset.gen.txt" format=
↪"ascii_v0" instances="2">
        <Port id="zero"/>
        <Port id="one"/>
    </EdenEventSetReader>

    <!-- Apply a synaptic projection from the event stream to the cells ! -->
    <projection id="MyFirstStreamProjection" presynapticPopulation=
↪"MyFirstEventStream" postsynapticPopulation="Pop" synapse="MySyn">
        <connectionWD id="0" preCellId="0" preSegmentId="zero" postCellId="0"␣
↪weight="1" delay="10 ms"/>
        <connectionWD id="1" preCellId="0" preSegmentId="one"  postCellId="1"␣
↪weight="1" delay="10 ms"/>
        <connectionWD id="2" preCellId="1" preSegmentId="zero" postCellId="1"␣
↪weight="1" delay="10 ms"/>
        <connectionWD id="3" preCellId="1" preSegmentId="one"  postCellId="0"␣
↪weight="1" delay="10 ms"/>
    </projection>

</network>
<Simulation id="MySim" length="1 s" step="100 us" target="MyNetwork" >
    <OutputFile id="MyFirstOutputFile" fileName="results.gen.txt">
        <OutputColumn id="my_first_vm"  quantity="Pop[0]/v"/>
        <OutputColumn id="my_second_vm" quantity="Pop[1]/v"/>
    </OutputFile>
</Simulation>
<Target component="MySim"/>
</neuroml>
```

```
Writing ReadEventSet.nml
```

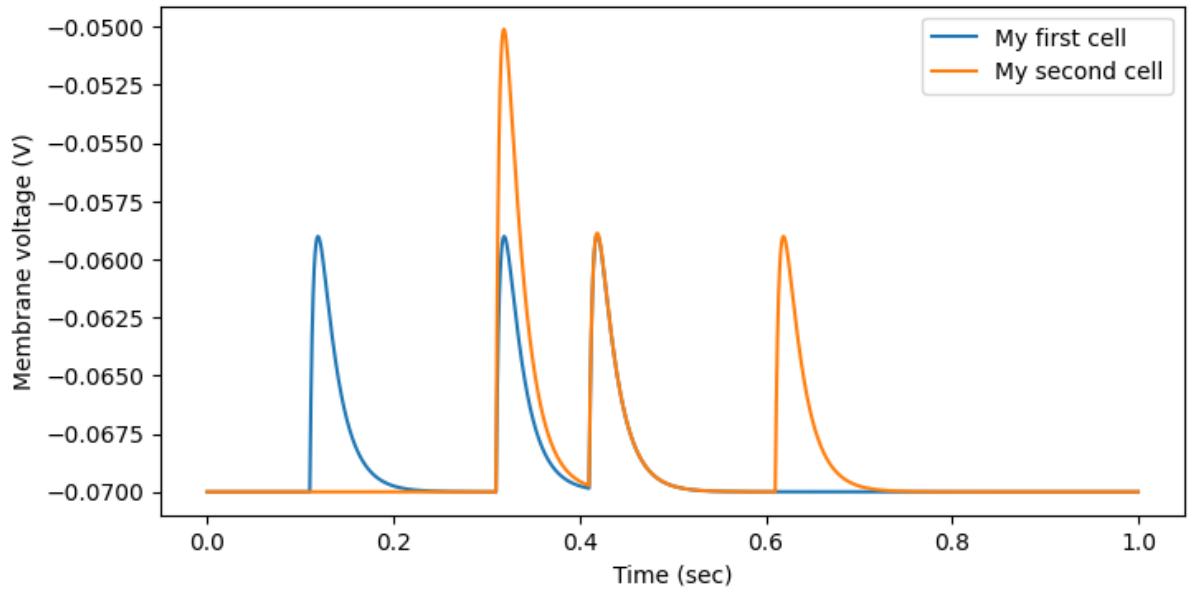
And then write up some "events":

```
%%writefile eventset.gen.txt
0.1 0!0
0.3 0!1 1!0 1!1
0.4 1!0 1!1
0.6 0!1
```

```
Writing eventset.gen.txt
```

And run the model plus injected spikes:

```
results = runEden('ReadEventSet.nml')
plt.figure(figsize=(8,4))
plt.plot(results['t'],results['Pop[0]/v'], label='My first cell')
plt.plot(results['t'],results['Pop[1]/v'], label='My second cell')
plt.xlabel('Time (sec)'); plt.ylabel('Membrane voltage (V)'); plt.legend();
```



Notice the delay between when the spikes happen and when the synapses are activated, just as `<connectionWD>` specifies.

Exercise: Repeat the example, with elements of a single port with this time, and hence simple integers as spike identifiers. In that case, the format is the same as what `<EdenEventOutputFile>` outputs.

In this simple case (of just a few pre-calculated spikes) one could have simply used the classic NeuroML `<spikeArray>`[248]s, but this gets unwieldy with population-wide spike trains and long durations. And, of course, this precludes dynamic live-streaming from another part of the experimental rig.

9.3.3 On shorting readers and writers

Note that directly recording data that comes in from readers is not officially supported, because of the instantaneous input-to-output requirement they imply *in conjunction with* the problematic cases that can occur when running on a MPI cluster if input and output streams are distributed among cluster nodes in arbitrary ways. It is permitted without support, but there may be a delay of one timestep between the information fed to the `<Eden...Reader>`s and that emitted by an `(Event)OutputWriter` that samples these input streams.

[248] `https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#spikearray`

CHAPTER

TEN

NEUROML EXTENSION: MULTIPLE FLOWS PER NEURON

In the NeuroML formulation, neurons are stimulated by various synapses and input sources that inject *non-specific* current in them, influencing the membrane potential.

However, there may often be different *flows* of ions or other substances, that should be tracked separately of total electrical current. Or more generally, a neuron may be influenced by a sum of mechanisms that supply something *other* than current.

To cover this use case, EDEN allows mechanisms to inject *multiple different things* flowing into cells, like ion channels do already. This is done by an existing pure-LEMS concept, that is generalised in EDEN so that different mechanisms can provide different sets of flows.

Tip: This is similar to how pure LEMS works already, with the difference that not all mechanisms need to expose every sort of flow; each mechanism that doesn't expose one of the additional flows is assumed to contribute `0` towards that flow.

Note: This feature is only supported for point neurons at the moment.

10.1 The `<DerivedVariable>` tag revisited

As we saw in the LEMS chapter (page 64), a `<DerivedVariable>` may take a `value` as a LEMS expression of other variables. Another option to resolve the value is by a `select` aggregate expression. EDEN supports the following tag attributes that describe the aggregate expression:

- `select` as `synapses[*]/`, followed after the `/` by the name of an `<Exposure>` that some synapses or probes may offer.
- `reduce` as `add` only, to add up the exposures of the same name.

The tag to gather all exposures of e.g. `iB` then gets to look like this:

```
<DerivedVariable name="ib" dimension="current" select="synapses[*]/iB" reduce="add
↪" />
```

To prevent ambiguity, each flow must have the same `<Exposure>` name among all mechanisms that contribute to it. Hence different flows should be `exposed` by components as something other than `i`.

- If components must expose the same flow under different names (or the modeller happens to treat them the same in one case), they can be `select` ed separately and an additional `<DerivedVariable>` may be defined for the total flow.

Note: Presently, the LEMS formulation of NeuroML concepts adds all "sort of external" mechanisms (known as `Attachment>`s) all into a singular `synapses` group, that includes *both* synapses instantiated by `<connection>`s and `<input>`s from `<inputList>`s!

An example of non-current input flow is seen in Tsodyks, Uziel, Markram (2000) (page 193). More examples using different flows may be provided, as requested (page 3). See another usage example in EDEN's program tests[249] which shows how to separate the flow from different synapses and input probes.

[249] `https://gitlab.com/c7859/neurocomputing-lab/Inferior_OliveEMC/eden/-/blob/main/testing/validation_tests/neuroml/EdenTest_Extension_Multiflux.nml`

CHAPTER

ELEVEN

LEMS EXTENSION: <WRITABLEREQUIREMENT>

In the NeuroML formulation, a neuron may be influenced by external mechanisms (such as synapses and input sources) that inject it with current (or chemical substances, when using pure LEMS or EDEN's Multiflux extension (page 121)). The neuron receives inward flows as *aggregates* for each chemical species of interest, and its state is influenced by every aggregate.

However, there exist neuron models where a synapse may *instantly* and *non-linearly* alter a neuron's internal state, in a way that changes the effect of other synapses (for example, fast saturation effects). EDEN allows such types of instant synapse-to-neuron effects through the `<WritableRequirement>` extension to LEMS (page 62) `<ComponentType>` dynamics.

Note: At the moment, this feature is only supported for synapses attached to point neurons.

11.1 The <WritableRequirement> tag

`<WritableRequirement>` tags look much like `<Requirement>`s, with required `name` and `dimension` attributes. The difference is that they can also be *assigned* values from the mechanism requiring them — even though they exist in another!

Assigning a value to a `<WritableRequirement>` is done with a `<StateAssignment>`; where the `variable` name typically refers to a `<StateVariable>` of the same mechanism, a `<WritableRequirement>` of another mechanism's quantity may be used instead.

For a straightforward use of `<WritableRequirement>` to implement "delta" synapses, refer to the Brunel 2000 tutorial (page 187). See also another toy example in EDEN's regression tests[250] which shows how to set a "trigger flag" of a *cell* from a *synapse*.

[250] `https://gitlab.com/c7859/neurocomputing-lab/Inferior_OliveEMC/eden/-/blob/main/testing/validation_tests/neuroml/EdenTest_Extension_WritableRequirement.nml`

Part C

Usage examples

The existing features of NeuroML, also combined with EDEN's extensions, allow users to set up *many* and *diverse* in-silico experiments.

Knowing what's been laid out in the previous chapters, one can proceed with conducting an *in silico neuroscience* project between *model conception* and *results collection*. (Some knowledge of systems neuroscience, and the functioning of the model-related parts, is also still needed for before and after these points.)

This part walks the reader through various tutorials and popular modelling use cases, introducing additional versatile techniques and ideas along the way. The reader is encouraged to repurpose parts of these examples (also together with examples from the previous parts, and their natural curiosity) and use this guide as a springboard to set up and explore their own models using EDEN (and NeuroML in general, as applicable).

CHAPTER

TWELVE

TUTORIAL: A NETWORK OF DETAILED CELLS

This tutorial shows how to build an active network of detailed neurons, simulate it to get the neurons' potentials over time in full spatial detail, and display these data as an animated 3D display.

Refer also to the previous introductory articles for the basics of NeuroML (page 13) and spatially detailed cells (page 33).

First of all, let's install the required software, if on certain platforms like Colab that run the bare notebooks:

```
import os; from pathlib import Path
if 'COLAB_BACKEND_VERSION' in os.environ:
  !TMP=$(mktemp -d); git clone https://eden-simulator.org/repo --depth 1 -b␣
↪development "$TMP"; cp -r "$TMP/." .; rm -rf "$TMP"
  exec(Path('.binder/install_livenb.py').read_text())
if 'DEEPNOTE_PROJECT_ID' in os.environ: exec(Path('../.binder/install_livenb.py').
↪read_text())
```

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
```

12.1 Constructing the network

Building a network model generally involves the following actions:

- place the *cells* forming the network;
- connect the cells with *synapses*;
- add the experimental *rig*: add external stimuli and probe the electro-chemical variables of interest.

12.1.1 Placing neurons

First, let's decide where to place the neurons in the model. For this tutorial, assume that neurons are uniformly spread over, say a tall cylinder that stands for a microcolumn. We'll use an evenly spread quasi-pseudo-random distribution[251], which won't randomly form misleading clumps like a true random sample would. We'll also sort the cells by distance along the cylinder, to give more meaning to the the resulting rasters and correlation matrices.

Check the following code also for a neat way to get uniformly random points over an arbitrary shape:

- check they fall on the shape, keep only those who do.
- resample to fill in the remaining slots.

[251] https://en.wikipedia.org/wiki/Low-discrepancy_sequence

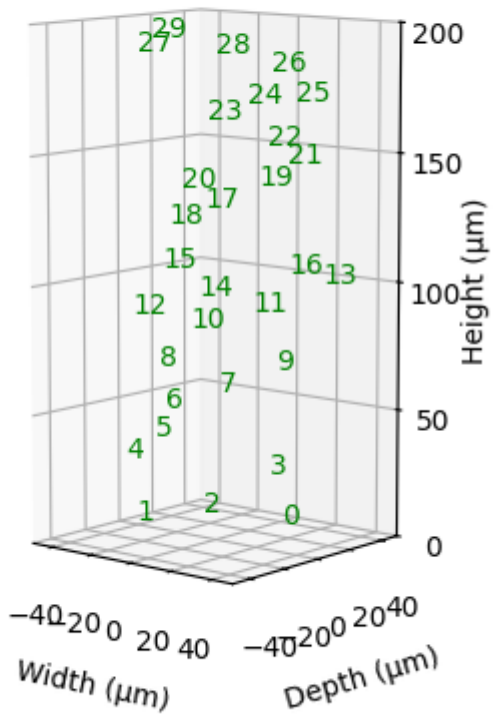
Kind of like painting with a stencil :)

```
!pip install -q 'scipy>=1.12' # for an evenly spread, *quasi* pseudo random␣
↪distribution
```

```
# get a few cells
# randomize x, y, z using e.g. a cylinder
# https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.qmc.Halton.html
↪#scipy.stats.qmc.Halton
seed = 123
N_cells = 30
# Points will be placed in a cylinder of given radius and height
region_radius = 50 # microns
region_height = 200

from scipy.stats import qmc
position_rng = qmc.Halton(d=3, scramble=True, seed=seed)

# Use rejection, to make the 3D points conform to an arbitrary domain
remaining_cells = N_cells
cell_positions = []
while len(cell_positions) < N_cells:
    # Get points over a uniform box
    point_samples = (
        position_rng.random(n=N_cells-len(cell_positions))
        * np.array([2*region_radius,region_height,2*region_radius])
        - np.array([region_radius,0,region_radius]) #list()
    )
    valid_points = [ (x,y,z) for (x,y,z) in point_samples if (x**2+z**2<region_
↪radius**2) and (y > 0 and y < region_height) ]
    cell_positions += valid_points
cell_positions = np.array(cell_positions)

# Also! Sort by height to make our lives easier!
cell_positions = cell_positions[np.argsort(cell_positions[:,1])]
# cell_positions is now ready!
```

Inspect the distribution visually. If running the notebook, rotate the display to see the distribution from all sides:

```
%matplotlib widget

fig = plt.figure(1); fig.clear(); fig.set_size_inches(4,5)
ax = fig.add_subplot(projection='3d')
# Note: usually in SWC and NeuroML files, +Y is 'up', zero is the soma middle, and␣
↪units are in microns.
# But these also vary with the provenance of the files, always check before using.
# Swap Y and Z for viz purposes.
for i, (x, z, y) in enumerate(cell_positions):
    ax.text(x, y, z, f'{i}', color='green')
# Tweaking display region and labels
ax.set_xlim(-region_radius,+region_radius)
ax.set_ylim(-region_radius,+region_radius)
ax.set_zlim(0, region_height)
ax.set_aspect('equal')
ax.set_xlabel('Width (µm)'); ax.set_ylabel('Depth (µm)'); ax.set_zlabel('Height␣
↪(µm)')
ax.view_init(elev=9, azim=-50, roll=0)
plt.show()
```



12.1.2 Adding synapses

Now let's hook up the neurons with synapses that will cause the cells to stimulate[252] each other, creating an interesting effect. To keep the code simple, we'll consider connections over each pair of cells (it's fine if there are not thousands of them).

We will assume *distance-dependent probability* (of each pair-wise synapse forming at all), *weight* (slightly randomized) and *delay* (linear with distance). Check the code for the specific formulae.

For this tutorial, we'll only consider soma-to-soma connections (i.e. between `<segment>s 0` of each); feel free to use your preferred models and methods. Refer to the relevant chapter (page 33) on how to handle spatially-detailed neurons.

```
from numpy.random import default_rng
synapses_seed = 123
rng = default_rng(synapses_seed)

# The range constant (sigma, in this case) for the synapses.
syn_radius = 100

from scipy.spatial.distance import pdist, squareform
d_mat = squareform(pdist(cell_positions)) # Distance between all pairs of cell␣
↪somata
```

(continues on next page)

[252] https://en.wikipedia.org/wiki/Excitatory_postsynaptic_potential

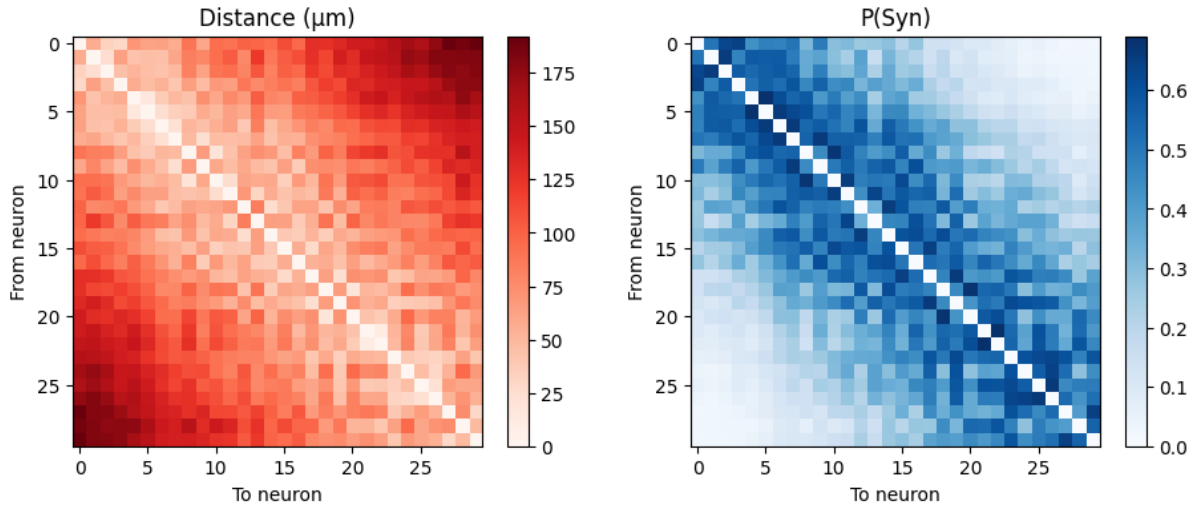
(continued from previous page)

```
p_mat = 0.7*np.exp(-(d_mat/syn_radius)**2) # Chance for connection = f(distance)
p_mat = p_mat - np.diag(np.diag(p_mat)) # Remove the diagonal

w_mat = 0.5*(rng.standard_normal(d_mat.shape)+2) * np.exp(-(d_mat/syn_radius)**2)
↪# Variable weight between pairs
t_mat = 0.001*np.exp(-(d_mat/syn_radius)**2) # Deterministic delay between pairs,␣
↪in seconds

pre = []; post = []; weight = []; delay = []; # Parallel pre/post synaptic cell,␣
↪weight, delay for ...
for pre_cell in range(N_cells):
    post_cells = (np.argwhere(p_mat[:,pre_cell] > rng.random(N_cells))).flatten()
↪# Get synapse targets that pass the Bernoulli test
#     print(post_cells)
    pre += ([pre_cell] * len(post_cells));  # append the relevant syn pairs, with␣
↪weight and delay
    post.extend(post_cells);
    weight.extend(w_mat[pre_cell,post_cells]);
    delay.extend(t_mat[pre_cell,post_cells]);
# print( "from:", pre, "\nto  :", post, "\nwei :", weight, "\ndel :", delay )
```

Let's show the per-neuron-pair matrices of the factors involved.

First, tabulate the absolute distance, and the probability of a connection for each neuron pair:

```
%matplotlib inline
plt.close('all')
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(11,4));
axs[0].set_title('Distance (µm)'); im = axs[0].imshow(d_mat, cmap='Reds'); fig.
↪colorbar(im, ax=axs[0]);
axs[1].set_title('P(Syn)'); im = axs[1].imshow(p_mat, cmap='Blues'); fig.
↪colorbar(im, ax=axs[1]);
for i in range(axs.shape[0]): axs[i].set_ylabel('From neuron'); axs[i].set_xlabel(
↪'To neuron');
```



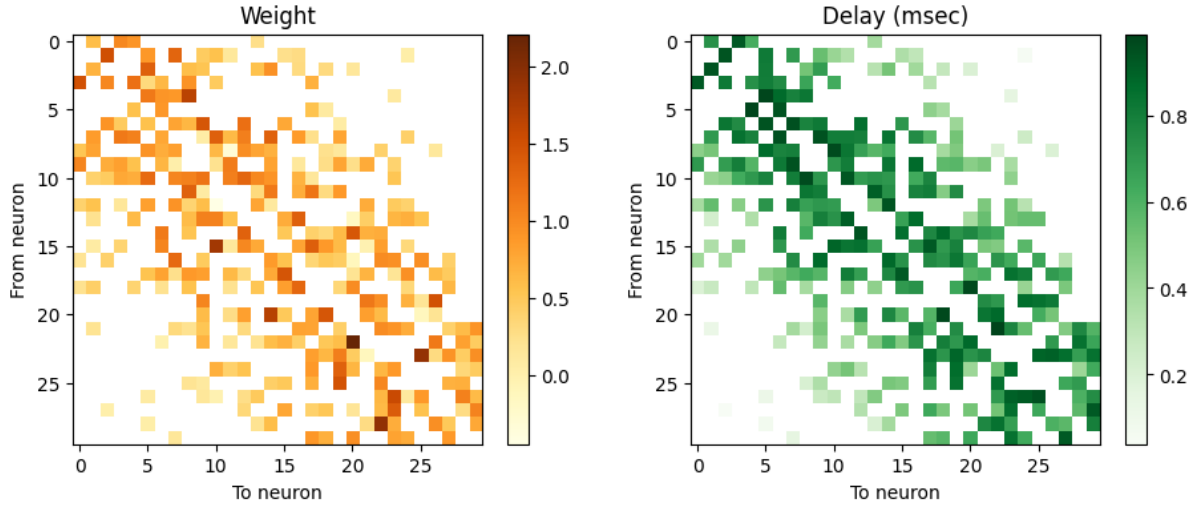
And now, tabulate weight and delay for the connections that were actually realized:

```
w_2d = np.zeros((N_cells,N_cells))*np.nan; d_2d = w_2d + 0 # full mask
w_2d[pre,post] = weight; d_2d[pre,post] = delay # fill in the values that apply

fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(11,4));
axs[0].set_title('Weight'); im = axs[0].imshow(w_2d, cmap='YlOrBr'); fig.
↪colorbar(im, ax=axs[0]); # could also use a diverging cmap centered at 0
```

(continues on next page)

(continued from previous page)

```
axs[1].set_title('Delay (msec)'); im = axs[1].imshow(d_2d*1000, cmap='Greens');␣
↪fig.colorbar(im, ax=axs[1]);
for i in range(axs.shape[0]): axs[i].set_ylabel('From neuron'); axs[i].set_xlabel(
↪'To neuron');
```



12.1.3 Adding stimuli

Now let's play with the network through unnatural means – for example, apply a one-off DC clamp to cells in the bottom 20% of the cylinder.

The attentive reader may have noticed that what's been specified so far is not specific to NeuroML. That's because NeuroML can run any *distribution* of cells and synapses just the same; they can be constructed independently with any method, and expressed in the NeuroML data format just before running the model.

The more involved and detailed descriptions(neuron shape, chemistry, dynamics and such) for the individual parts of the model will be specified in NeuroML, in the section right after.

```
stim_cells = [i for i, (x,y,z) in enumerate(cell_positions) if y < region_height/5]
stim_cells
```

```
[0, 1, 2, 3, 4, 5]
```

12.2 Generating NeuroML for the model

Now it's time to make the NeuroML files in order to run the simulation. Let's put them in a sub-folder of the working directory, to avoid polluting the folder that the notebook is in too much.

```
nml_dir = 'tut_net/'
os.makedirs(nml_dir, exist_ok=True)
```

For the cell's description, let's cheat and grab an existing description from the NeuroML-DB.

```
nmldb_cell_name = 'NMLCL000625'
zip_file_name = f'{nmldb_cell_name}.zip'
# Download the zip file with the model
import urllib.request
# Because NMLDB is having some issues with HTTPS, don't demand HTTPS verification
import ssl; ssl._create_default_https_context = ssl._create_unverified_context
urllib.request.urlretrieve(f'http://neuroml-db.org/GetModelZip?modelID={nmldb_cell_
```

(continues on next page)

(continued from previous page)

```
↪name}&version=NeuroML', zip_file_name)
# and unpack it
from zipfile import ZipFile
with ZipFile(zip_file_name, 'r') as zipp: zipp.extractall(nml_dir+nmldb_cell_name+
↪'/')
```

We'll also need the cell type's identifier from the NML file, which for some reason is not the `NMLCLxxxxxx` identifier. (Neither is the NML file)

This is the same as the `<name>.cell.nml` filename, so let's scan for that.

Also, the `<spikeThresh>` to register an action potential is not specified for some reason. Because we're using classical chemical synapses, we'll have to add it to `<biophysicalProperties>`.

```
# LATER get it with libNML or with the NMLDB API.
import os
celltype_name = None
cell_nml_file_suffix = '.cell.nml'
for filename in os.listdir(nml_dir+nmldb_cell_name):
    # print(filename)
    if not filename.endswith(cell_nml_file_suffix): continue # get the cell.nml␣
↪files
    new_celltype_name = filename[:-len(cell_nml_file_suffix)] # there it is
    if celltype_name: raise ValueError(f'cell type: is it {new_celltype_name} or
↪{new_celltype_name}') # :c
    celltype_name = new_celltype_name
print('Celltype name:', celltype_name)

# Also modify the NML file to add spikeThresh
cell_filename = nml_dir+nmldb_cell_name+'/'+new_celltype_name+cell_nml_file_suffix
# Read in the file
with open(cell_filename, 'r') as file: filedata = file.read()
# Replace the target string
if '<spikeThresh' not in filedata:filedata = filedata.replace('</
↪membraneProperties>','<spikeThresh value="0 mV"/></membraneProperties>')
# Write the file out again
with open(cell_filename, 'w') as file: file.write(filedata)
# print(filedata)
```

```
Celltype name: cACint209_L6_NBC_a3972c5d97_0_0
```

Now let's add some routines to construct a NeuroML file incrementally from the population and projection data we collected so far, into one big string. There are also other ways to create and manipulate NeuroML files, but this one here makes for a more direct demonstration.

```
def NmlHeader():
    return '''<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_v2.1.xsd">
    <include file="sim_components.nml"/>'''
def NmlNetworkHeader(): return '''\n   <network id="Net" type=
↪"networkWithTemperature" temperature="37degC">'''
def NmlNetworkFooter(): return '''\n    </network>'''

def NmlPopulation(pop_name, celltype_name, cell_positions):
    return f'''
        <population id="{pop_name}" component="{celltype_name}" size="{len(cell_
↪positions)}">
```

(continues on next page)

(continued from previous page)

```
        '''+'\n\t'.join([
            f'  <instance id="{i}"><location x="{x}" y="{y}" z="{z}"/></instance>'
            for i, (x,y,z) in enumerate(cell_positions) ])+'''
        </population>
    '''

def NmlSynapticProjection(
    proj_name,syncomp_name, pre_pop_name, post_pop_name,
    preCell, postCell, weight=None, delay=None, preSeg=None, postSeg = None,␣
↪preFrac = None, postFrac = None):
    if delay is None: delay = [0]*len(preCell)
    if weight is None: weight = [1]*len(preCell)
    if preSeg is None: preSeg = [0]*len(preCell)
    if postSeg is None: postSeg = [0]*len(preCell)
    if preFrac is None: preFrac = [0]*len(preCell)
    if postFrac is None: postFrac = [0]*len(preCell)

    return f'''
        <projection id="{proj_name}" synapse="{syncomp_name}"␣
↪presynapticPopulation="{pre_pop_name}" postsynapticPopulation="{post_pop_name}">
            '''+'\n\t    '.join([
            f'<connectionWD id="{i}" preCellId="{preCell[i]}" postCellId="
↪{postCell[i]}" '+
            f'weight="{weight[i]}" delay="{delay[i]} s" '+
            f'preSegmentId="{preSeg[i]}" postSegmentId="{postSeg[i]}" '+
            f'preFractionAlong="{preFrac[i]}" postFractionAlong="{postFrac[i]}"/>'
            for i in range(len(pre)) ])+'''
        </projection>
    '''

def NmlInputList(input_list_name,stim_component,target_pop_name,cells, segs=None,␣
↪fracs=None):
    if segs is None: segs = [0]*len(cells)
    if fracs is None: fracs = [0]*len(cells)
    return f'''
        <inputList id="{input_list_name}" population="{target_pop_name}" component=
↪"{stim_component}">
        '''+'\n\t'.join([
            f'  <input id="{i}" target="../{target_pop_name}[{cells[i]}]" '+
            f'segmentId="{segs[i]}" fractionAlong="{fracs[i]}" destination=
↪"synapses"/>'
            for i in range(len(cells)) ])+'''
        </inputList>
    '''

def NmlFooter():return '''\n</neuroml>'''

# celltype_name = 'cACint209_L6_NBC_a3972c5d97_0_0'
# celltype_name = 'cADpyr232_L5_TTPC2_8bab918b58_0_0'
# celltype_name = 'dNAC222_L6_SBC_194972ee43_0_0'

population_name = 'MyCells'

# Write the network file.
with open(nml_dir+"/example.nml", "w") as f:
    f.write(NmlHeader()); f.write(NmlNetworkHeader())
    f.write(NmlPopulation(population_name, celltype_name, cell_positions))
    f.write(NmlSynapticProjection('FirstSynProjection', 'NMDA', population_name,␣
↪population_name, pre, post, weight, delay))
    f.write(NmlInputList('FirstStimList', 'MyStim', population_name, stim_cells))
    f.write(NmlNetworkFooter())
```

(continues on next page)

(continued from previous page)

```
    f.write(NmlFooter())
```

After writing the `net.nml` representing the network made up of cells, synapses and input stimuli, we'll make a companion file `LEMS_<something>.xml` (this is the convention; I guess you could also use `.sim.xml` to make the name further clearer.)

Here, we'll specify some parameters to run the simulation like for how long and with how big a timestep, but also add the last part of the rig: recording certain trajectories and spike trains from the simulated model. Since neurites in each cell usually fire together(in sequence), we'll record the membrane voltage trajectories for the soma of each neuron. (Conveniently enough, if the location on the neuron is not specified in a NeuroML path, the location of the soma is used.)

```
# Write the sim file.
def NmlSimParms(pop_name, N_cells, run_time, run_timestep=25e-6):
    return f'''
        <Simulation id="sim1" length="{run_time}s" step="{run_timestep}s" target=
↪"Net">
            <OutputFile id="first" fileName="tut_net/results.gen.txt">
            '''+'\n\t        '.join([f'<OutputColumn id="v_{i}" quantity="{pop_
↪name}[{i}]/v"/>' for i in range(N_cells)])+'''
            </OutputFile>
        </Simulation>
        <Target component="sim1"/>'''

with open(nml_dir+"/Sim.xml", "w") as f:
    f.write('<?xml version="1.0" encoding="UTF-8"?>\n<Lems>\n<include file=
↪"example.nml"/>\n')
    f.write(NmlSimParms(population_name, N_cells, run_time=0.25, run_timestep=25e-
↪6))
#     f.write(NmlSimParmss(population_name,celltype_name,None, run_time=0.25, run_
↪timestep=25e-6))
    f.write('\n</Lems>\n')
```

```
%%writefile $nml_dir/sim_components.nml
<neuroml>
    <include file="NMLCL000625/cACint209_L6_NBC_a3972c5d97_0_0.cell.nml"/>
    <expTwoSynapse id="NMDA" gbase=".5nS" erev="0mV" tauDecay="15ms" tauRise="0.
↪15ms"/>
    <pulseGenerator id="MyStim" delay="10ms" duration="20ms" amplitude="0.5nA"/>
</neuroml>
```

```
Writing tut_net//sim_components.nml
```

Here are some more tested parameter combinations for alternative cell types. You can replace the `nmldb_cell_name` above, download a new neuron model, then uncomment and run the corresponding snippet in place of the code above. Or even get a new cell type from other sources, such as NeuroML-DB[253], OSB[254] or the internet in general.

```
# %%writefile nml_sim/sim_components.nml
# <neuroml>
#     <include file="NMLCL000693/cADpyr232_L5_TTPC2_8bab918b58_0_0.cell.nml"/>
#     <expTwoSynapse id="NMDA" gbase="6.5nS" erev="0mV" tauDecay="15ms" tauRise="0.
↪15ms"/>
#     <pulseGenerator id="MyStim" delay="10ms" duration="50ms" amplitude="1.5nA"/>
# </neuroml>
```

```
# %%writefile nml_sim/sim_components.nml
# <neuroml>
```

(continues on next page)

[253] https://neuroml-db.org/gallery
[254] https://www.opensourcebrain.org

(continued from previous page)

```
#     <include file="NMLCL001078/dNAC222_L6_SBC_194972ee43_0_0.cell.nml"/>
#     <expTwoSynapse id="NMDA" gbase="0.57nS" erev="-0mV" tauDecay="17ms" tauRise=
↪"0.05ms"/>
#     <pulseGenerator id="MyStim" delay="10ms" duration="100ms" amplitude="0.2nA"/>
# </neuroml>
```

```
import eden_simulator
%time results = eden_simulator.runEden(nml_dir+"/Sim.xml")
```

```
CPU times: user 104 ms, sys: 6.27 ms, total: 110 ms
Wall time: 11.7 s
```

Let's see what we got out of the simulation:

```
print(results.keys())
```

```
dict_keys(['MyCells[0]/v', 'MyCells[1]/v', 'MyCells[2]/v', 'MyCells[3]/v',
↪'MyCells[4]/v', 'MyCells[5]/v', 'MyCells[6]/v', 'MyCells[7]/v', 'MyCells[8]/v',
↪'MyCells[9]/v', 'MyCells[10]/v', 'MyCells[11]/v', 'MyCells[12]/v', 'MyCells[13]/v
↪', 'MyCells[14]/v', 'MyCells[15]/v', 'MyCells[16]/v', 'MyCells[17]/v',
↪'MyCells[18]/v', 'MyCells[19]/v', 'MyCells[20]/v', 'MyCells[21]/v', 'MyCells[22]/
↪v', 'MyCells[23]/v', 'MyCells[24]/v', 'MyCells[25]/v', 'MyCells[26]/v',
↪'MyCells[27]/v', 'MyCells[28]/v', 'MyCells[29]/v', 't'])
```

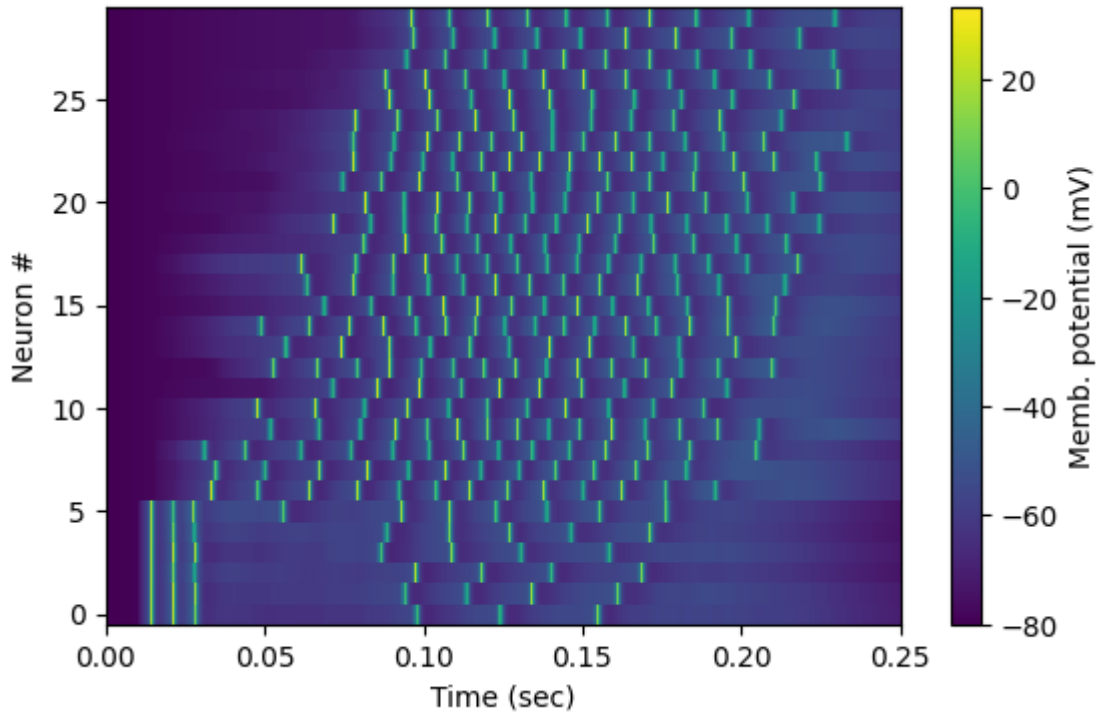
12.3 Displaying per-soma activity

Let's show how each cell behaved during the simulation, with an analog raster.

We observe that the bottom few cells were stimulated together, fired together and then entered a refractory period; meanwhile the other cells stimulate one another into a wave that spreads out (in spatial order!), reverberates for some time, and dies out near the end of the simulation.

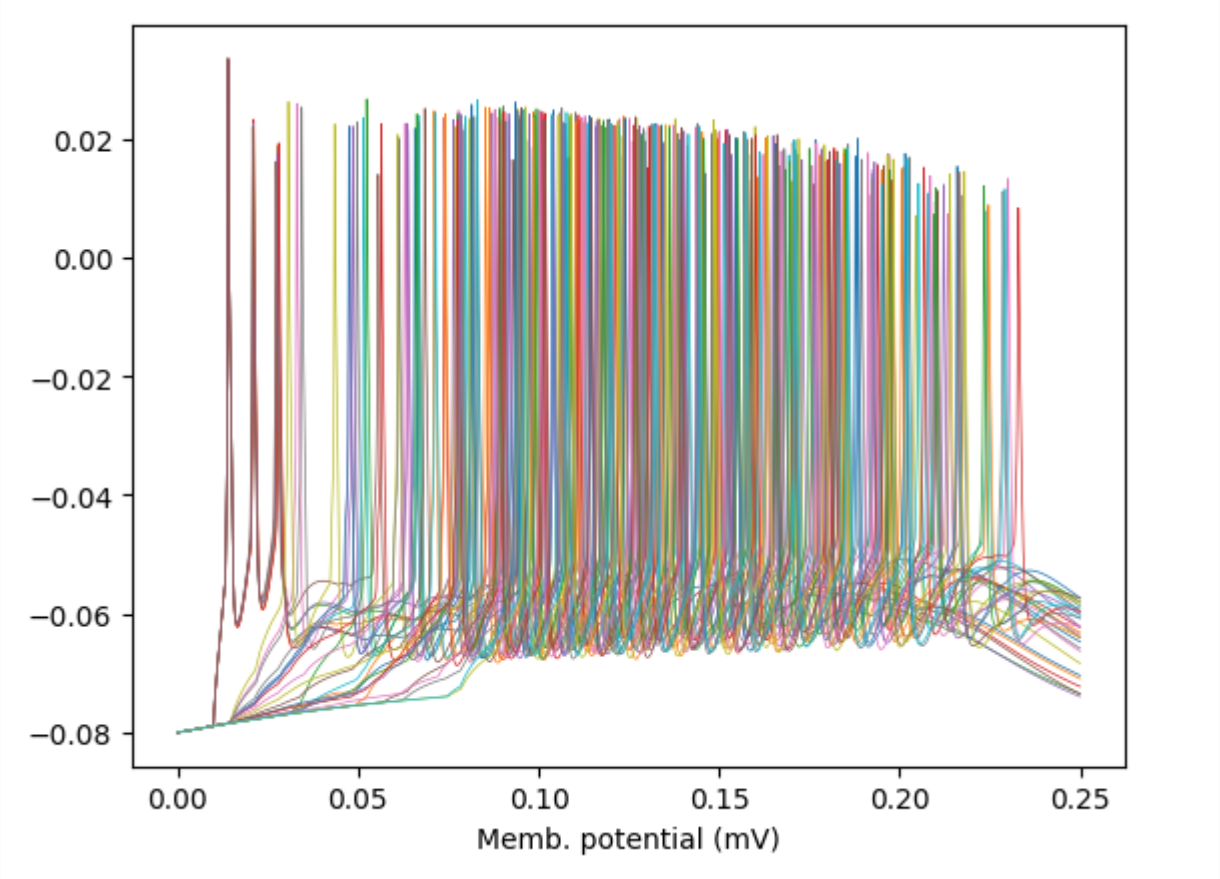
Observe also the sub-threshold behavior; the cells were gradually brought to a depolarisation level before they started firing repetitively.

```
neuron_waveforms = np.array([results[f'{population_name}[{i}]/v'] for i in range(N_
↪cells)])
neuron_appearance_order = np.argsort(cell_positions[:,1]) # just in case they weren
↪'t sorted before
# neuron_appearance_order = range(N_cells) # alternative order, if not sorted
fig = plt.figure(figsize=np.array([8,5])*.8); ax = plt.gca()
im = plt.imshow(1000*neuron_waveforms[neuron_appearance_order,:], # in mVolts
    extent=[ results['t'][0], results['t'][-1], N_cells-.5,-0.5 ],
    aspect='auto', interpolation='none', cmap ="viridis")
cbar = plt.colorbar(im); cbar.set_label('Memb. potential (mV)')
ax.set_xlabel('Time (sec)'); ax.set_ylabel('Neuron #')
ax.invert_yaxis() # to match the cell positions' 'up' in 3-D space
plt.show(); fig.savefig('tut_net_raster.png',dpi=300)
```



Here's an alternative line plot, to see how it gets unwieldy for large population sizes: (it could be ameliorated with a EEG style vertical offset, at the cost of resolution though)

```
fig = plt.figure()
plt.plot(results['t'], neuron_waveforms.T, linewidth=.5)
plt.xlabel('Time (s)');plt.xlabel('Memb. potential (mV)');plt.show()
fig.savefig('tut_net_jumble.png',dpi=300)
```



Since we have the cells' positions, we can display soma potential in 3-(4 including time)-D space as well.

First, we should reduce the amount of data to display to what's *perceiveable* at the animation speed we choose, i.e. downsample the recorded waveforms in time. Note that this was already done previously, through the `plt.figure`'s parameters for `figsize` and `dpi`. This is neatly done by the auxiliary `eden_simulator.display.animation.subsample_trajectories` (page 238) routine, that comes with the `display` (page 238) section of the Python API (page 237).

```
from eden_simulator.display.animation import subsample_trajectories
samples, anim_axis, sampled_time_axis, [sampled_soma_voltage] = subsample_
↪trajectories(
    results['t'], [neuron_waveforms.T * 1000], animation_speed=0.03, animation_
↪frames_per_second=60)
```

Then we'll display the waveforms as an animated 3D scatter plot, using `k3d.points`[255]. We'll also use `eden_simulator.display.spatial.k3d.Plot` (page 240) and `IPython.display.HTML`[256] for improved publishing.

```
import k3d; from eden_simulator.display.spatial.k3d import Plot
Point_plot = plot = Plot(camera_auto_fit=False) # to override camera orientation

k3d_anim_dict = { str(real_time): x.astype(np.float32)
                  for (real_time, x) in zip(anim_axis, sampled_soma_voltage)  }
k3d_label_dict = { str(real): f't = {sim:.3f} ~ s' for (real, sim) in zip(anim_
↪axis, sampled_time_axis)  }
plt_points = k3d.points(positions=cell_positions.astype(np.float32),attribute=k3d_
↪anim_dict,
        color_range=[-80, +20],point_size=10,color_map=k3d.matplotlib_color_maps.
```

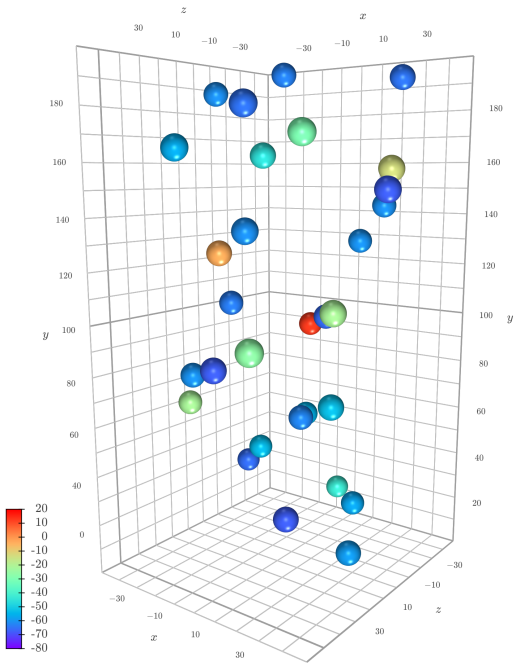
(continues on next page)

[255] https://k3d-jupyter.org/reference/factory.points.html
[256] https://ipython.readthedocs.io/en/stable/api/generated/IPython.display.html#IPython.display.HTML

(continued from previous page)

```
↪Rainbow); plot += plt_points

plot.camera = plot.get_auto_camera(pitch=30, yaw=10)[:6]+[0,1,0] # set 'y' to up !␣
↪ LATER zoom a bit also; vecs are pos, tgt, up
plt_label = k3d.text2d(k3d_label_dict, (0.,0.), label_box=False); plot += plt_
↪label # add 2d elements AFTER setting auto camera
plot.fps = 60;
from IPython.display import display, HTML, IFrame
plot.snapshot_type = 'inline'; display(HTML(plot.get_snapshot()))
```

```
<IPython.core.display.HTML object>
```



But this doesn't look much like the tissue we're supposed to simulate, does it? We have spent all this computer time to simulate all these neurites making up the cell, we deserve more attractive visuals!

Let's move on then to display the membrane voltage all over the cell, in glorious, 24 bit, false color! (You'd need fluo to see it on the real thing anyway.)

12.4 Displaying whole-neuron activity

To observe what happens all over the cell, we'll have to *record* all over, and *display* the whole cell.
And we'll achieve this through some cool add-ons to EDEN that:

- explain how spatially detailed cells are being simulated as discrete elements of neurite,
- and how exactly these parts and the whole neuron look like as polygonal solids.

See also the chapter (page 33) which explains the structure of spatially-detailed neurons, and introduces and uses said tools.

First, we'll retrieve for this cell type, how many *compartments* is is cut into. Each *compartment* in a model neuron is equivalent to a *pixel* in an image; it is a (hopefully) small bit of the neuron where we assume all electrical, chemical etc. properties are uniform throughout.

We can get this as follows:

```
cells_info = eden_simulator.experimental.explain_cell(nml_dir+"/Sim.xml")
print(type(cells_info))
print(cells_info.keys())
cell_info = cells_info[celltype_name]
print(cell_info.keys())
```

```
<class 'dict'>
dict_keys(['cACint209_L6_NBC_a3972c5d97_0_0'])
dict_keys(['comp_parent', 'comp_start_pos', 'comp_end_pos', 'comp_midpoint', 'comp_
↪midpoint_segment', 'comp_midpoint_fractionAlong', 'comp_length', 'comp_path_
↪length_from_root', 'comp_area', 'comp_volume', 'comp_capacitance', 'comp_
↪conductance_to_parent', 'segment_groups', 'mesh_vertices', 'mesh_faces', 'mesh_
↪comp_per_face'])
```

For each *physically modelled** cell type, we got a set of lists, most with as many elements as there are compartments for each cell type. We can then use the `comp_midpoint_segment` and `comp_midpoint_fractionAlong` lists to locate the middle of each compartment, and tap into that to record the membrane voltage for each part of each neuron.

Given this information, let's record for each cell. We'll also cut down on the sampling rate (using the Eden-specific extension `<EdenOutputFile>`) because that's way more than the one trajectory per cell we were recording before.

To learn more about `explain_cell`, refer to its Python API page (page 241).

* that is, excluding artificial cells which typically have unique properties

```
comp_mid_seg = cell_info['comp_midpoint_segment']
comp_mid_fra = cell_info['comp_midpoint_fractionAlong']
n_comps = len(comp_mid_seg) # Number of actual compartments in this cell, may be␣
↪retrieved from most cell_info arrays
print(f'{n_comps} compartments per cell')
```

```
202 compartments per cell
```

Now we'll construct a `SimMore.xml` file that records the voltage of *every single compartment* on *every single cell*.

We'll also cut down how often we record these waveforms to 0.5 msec (default NeuroML behaviour is to record for every single timestep which is quite a lot of steps, and we don't always need this much resolution.)

Note the use of `<EdenOutputFile>`, an EDEN-specific version of the regular `<OutputFile>` with more recording options (page 110). Another one of these is controlling the units per recorded trajectory in `output_units`, which we'll use to record membrane voltage in the more commonly used millivolts, this time.

```
# Set a new routine for making the <Simulation> file with new recording options
def NmlSimParmss(run_time, run_timestep=25e-6, sampling_period=1e-3,
            rec_lines='', href='file://results.gen.txt'):
    return f'''
        <Simulation id="sim1" length="{run_time}s" step="{run_timestep}s" target=
↪"Net">
            <EdenOutputFile id="first" href="{href}" format="ascii_v0" sampling_
↪interval="{sampling_period} s">
            '''+'\n\t        '.join(rec_lines)+'''
            </EdenOutputFile>
        </Simulation>
        <Target component="sim1"/>'''

# Make the traces to record each compartment, and save the file
traces = [f"{population_name}[{neu}]/{celltype_name}/{comp_mid_seg[i]}{('%.9f'
↪%comp_mid_fra[i])[1:]}/v"
    for neu in range(N_cells) for i in range(len(comp_mid_seg))]
rec_lines = [f'<OutputColumn id="v_{i}" quantity="{x}"  output_units="mV"/>' for i,
↪x in enumerate(traces) ]
print(f"recording {len(rec_lines)} waveforms this time !")
with open(nml_dir+"/SimMore.xml", "w") as f:
```

(continues on next page)

(continued from previous page)

```
    f.write('<?xml version="1.0" encoding="UTF-8"?>\n<Lems>\n<include file=
↪"example.nml"/>\n')
    f.write(NmlSimParmss(run_time=0.25, run_timestep=25e-6, sampling_period=0.5e-3,
↪ href='./moresults.gen.txt', rec_lines=rec_lines))
    f.write('\n</Lems>\n')
```

```
recording 6060 waveforms this time !
```

```
%time moresults = eden_simulator.runEden(nml_dir+"/SimMore.xml",verbose=True)
```

```
Using bundled executable: on /home/docs/checkouts/readthedocs.org/user_builds/eden-
↪simulator/envs/latest/lib/python3.10/site-packages/eden_simulator/data/bin/eden
['/home/docs/checkouts/readthedocs.org/user_builds/eden-simulator/envs/latest/lib/
↪python3.10/site-packages/eden_simulator/data/bin/eden', 'nml', 'tut_net//SimMore.
↪xml', 'gcc']
Ran EDEN in 12.43 seconds
CPU times: user 322 ms, sys: 75 ms, total: 397 ms
Wall time: 12.8 s
```

Note: Running simulations with lots of output may often take appreciably more time just to write down and read back all these data …

… now imagine storing all that, times a whole parameter space D:

(Though this time, we cut down the sampling rate by a lot so it balances out)

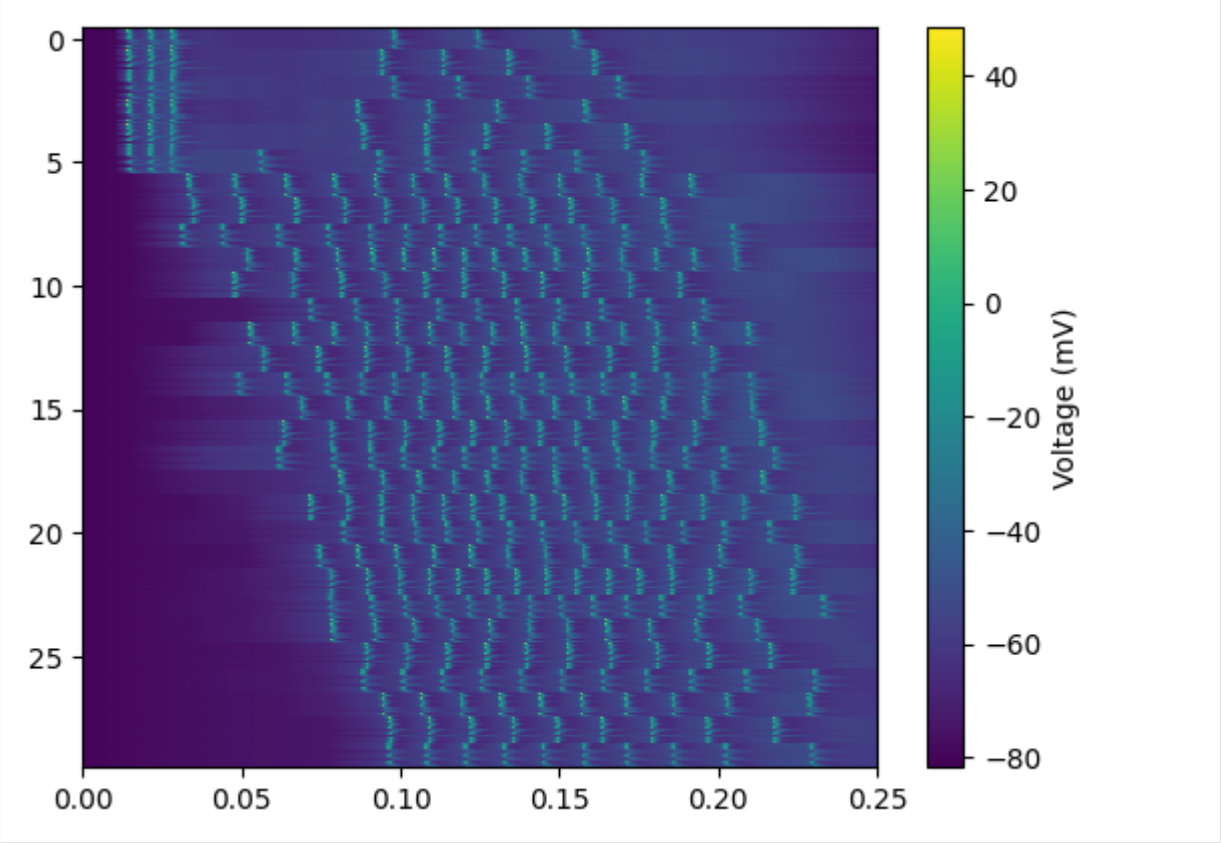
Now we have many more traces in the sim output, one per neuron per compartment!

Just in case we want differently-shaped neurons in the future, let's leave the traces list to be:

- instead of making an array of `N_cells x n_comps x timesteps`, assign an index number in the sequence of traces, from which the set of traces start (n_comps trajectories, including the indexed one)

```
from matplotlib import pyplot as plt
neuron_waveforms = np.array([moresults[x] for i,x in enumerate(traces)])
print('Waveform matrix size:', neuron_waveforms.shape)
# XXX append the offsets
timevec = moresults['t']
offset_per_neuron = np.round(np.arange(N_cells)*n_comps)
print('Row start per cell:', offset_per_neuron)
im = plt.imshow(neuron_waveforms,
    extent=[ results['t'][0], timevec[-1], N_cells-.5,-0.5 ],
    aspect='auto', interpolation='none', cmap ="viridis");cbar = plt.colorbar(im);␣
↪cbar.set_label('Voltage (mV)')
```

```
Waveform matrix size: (6060, 501)
Row start per cell: [   0  202  404  606  808 1010 1212 1414 1616 1818 2020 2222␣
↪2424 2626
 2828 3030 3232 3434 3636 3838 4040 4242 4444 4646 4848 5050 5252 5454
 5656 5858]
```



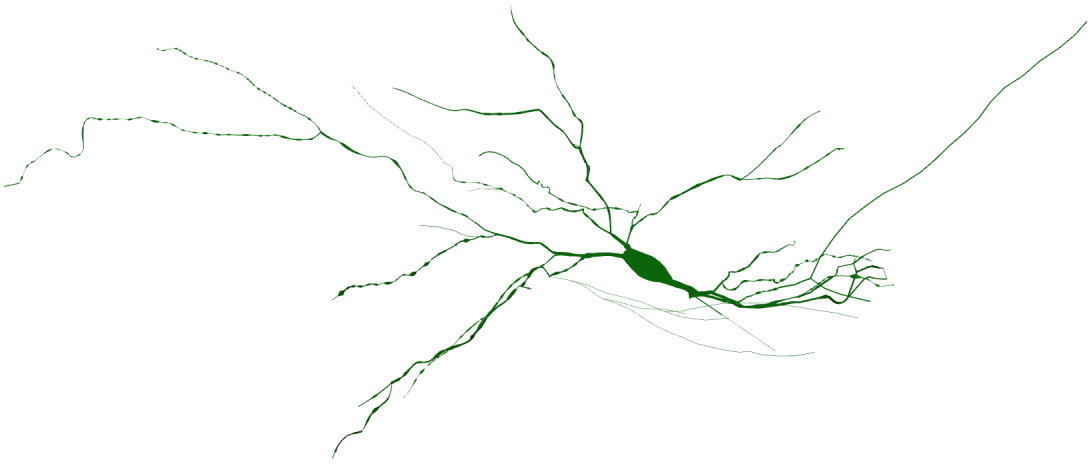
When we ran `explain_cell`, some lists `mesh_vertices`, `mesh_faces` and `mesh_comp_per_face` were also provided for the cell. These represent the solid shape (in computer graphics parlance, a *mesh*) that cells of this type have. (For unique morphology, one would make individual cell types in NeuroML.)

Meshes are made up from a set of points (known as *vertices*) in 3-D space, and a set of polygonal *faces* (typically triangles). Thus the joined flat *faces* form the shape together, just like lines connected through lines do in 2D space. To display a 3-D object, meshes may be enhanced with texture images and related attributes, to show a more detailed surface on the meshes. But for our purpose, we'll be painting the mesh explicitly, so that we can show biophysical attributes across each cell.

Here is what the neuron looks like according to the provided mesh, in plain color:
(Use the mouse over the picture to rotate/move the neuron)

```
mesh_vertices = cell_info['mesh_vertices']
mesh_faces    = cell_info['mesh_faces'   ]
import trimesh
viz = trimesh.Trimesh(
    vertices=mesh_vertices, faces=mesh_faces )
viz.visual.face_colors = (0.1,0.9,0.1)
viz.show()
```

```
<IPython.core.display.HTML object>
```



Note that that if z is up, that's a different orientation compared to the render[257] in the NeuroML-DB.

For the following, we'll assume $\overrightarrow{y}$ points to "up" and rotate the `plot.camera` accordingly, just like we did in the previous point-based animation. (For more about the meaning of each coordinate axis, refer to the article (page 38) on spatially-detailed cells.)

In addition to the `mesh`'s `vertices` and `faces` we got a list `mesh_comp_per_face`, which indicates which *compartment* is represented by each *face* of the mesh.

```
face_comp = cell_info['mesh_comp_per_face']
print(f"We have {len(mesh_faces)} faces in the mesh, and {len(face_comp)} matching␣
↪labels.")
```

```
We have 30148 faces in the mesh, and 30148 matching labels.
```

Combining this with the vertex numbers that each face is made of, we can have both a mapping from each compartment to each corresponding face, as well as from each compartment to each vertex. (Different 3D graphics utilities prefer each form.)

We'll now use this mapping to paint the neuron selectively, with a different colour for each compartment, to show how membrane potential spreads when it's initiated from the soma. This can be done immediately through the auxiliary `eden_simulator.display.spatial.k3d.plot_neuron` (page 240) routine, that comes with the `display` (page 238) section of the Python API (page 237).

Tip: When animating a spatially neuron over many time samples, use `compress_cells = True`(default) with `plot_neuron` and display it on a `eden_simulator.display.spatial.k3d.Plot` with `IPython.display.HTML` or `IPython.display.IFrame` to keep the memory and space requirements under control.

```
from eden_simulator.display.animation import subsample_trajectories
_, anim_axis, sampled_time_axis_sec, (sampled_voltage,) = subsample_trajectories(
    timevec, [neuron_waveforms.T], animation_speed=0.03, animation_frames_per_
↪second=60)

import k3d; from eden_simulator.display.spatial.k3d import Plot, plot_neuron
Multicomp_plot = plot = Plot()
k3d_label_dict = { str(real): f't = {sim:.3f} ~ s' for (real, sim) in zip(anim_
↪axis, sampled_time_axis_sec)  }
for neu in range(N_cells):
    tra_off = offset_per_neuron[neu] # some traces per neuron, starting from ...
    comp_trajes = sampled_voltage[:,tra_off:tra_off+n_comps] # there they are
    plot += plot_neuron(cell_info, comp_trajes, time_axis_sec=anim_axis,␣
↪translation=cell_positions[neu,:],
                            color_range=[-80, 0],color_map='rainbow');
```

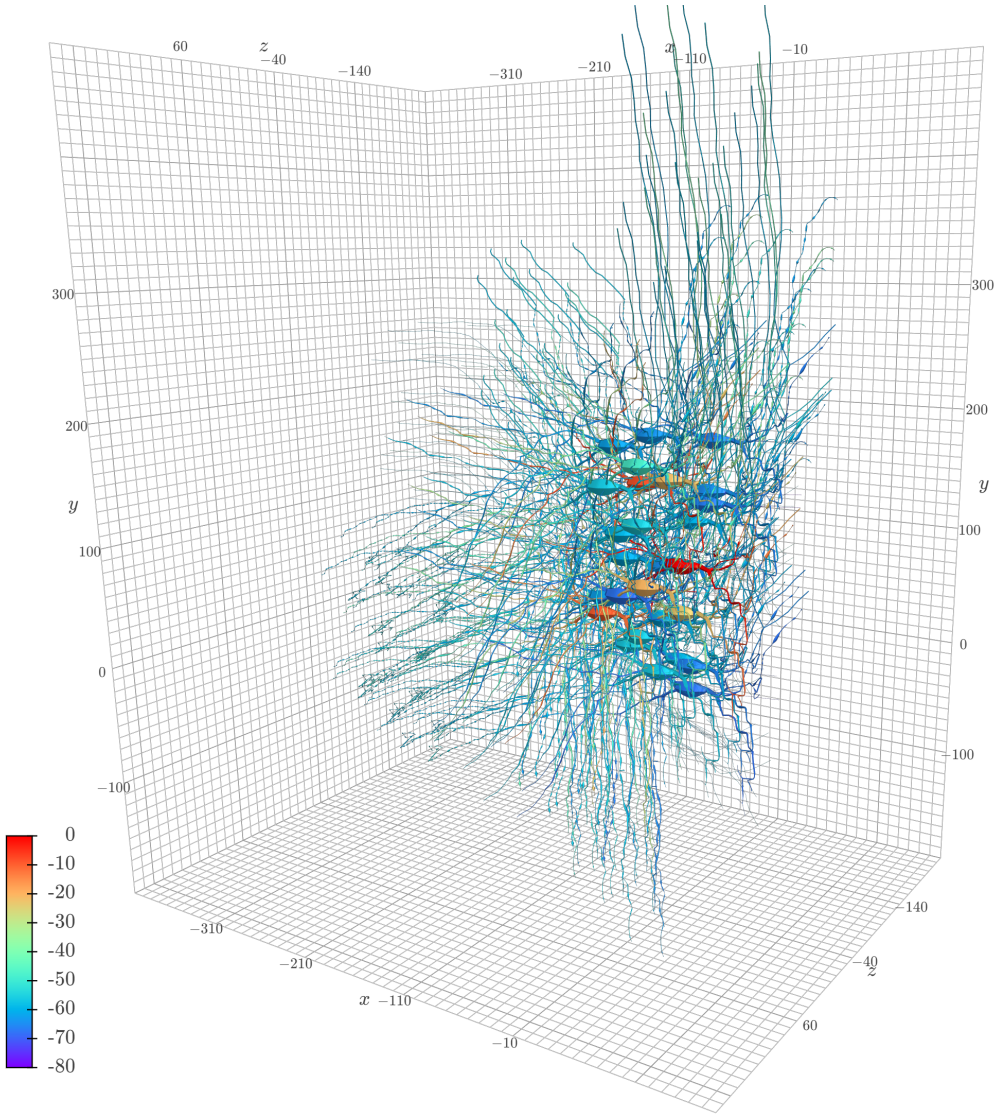
(continues on next page)

[257] https://neuroml-db.org/api/gif?id=NMLCL000625

(continued from previous page)

```
plot.camera = plot.get_auto_camera(pitch=30, yaw=10)[:6]+[0,1,0] # set 'y' to up !␣
↪ LATER zoom a bit also; vecs are pos, tgt, up
plt_label = k3d.text2d(k3d_label_dict, (0.,0.)); plot += plt_label # add 2d␣
↪elements AFTER setting auto camera

# Since the plot is quite big, include it outside of the notebook so that the␣
↪notebook renders quick.
plot.snapshot_type = 'online'; plot_html = plot.get_snapshot()
plot_file = '_static/tutorial_network_big_animation.html'
with open(plot_file, 'w') as f: f.write(plot_html)
display(IFrame('_static/tutorial_network_big_animation.html', '100%', f'{plot.
↪height} px'))
```

```
<IPython.lib.display.IFrame at 0x7401416fbbb0>
```



Observe how the spikes travel along the neurites.

Exercise: While constructing the model (before implementing the NeuroML description), there was a mistake in one of the formulae for the network parameters. This mistake makes whether the neurons' spatially reverberating waves occur, extremely sensitive to the physical parameters of the neuron model, synapses and applied probes.

- Find the mistake and re-evaluate sensitivity (also with the alternative cell types commented out above (page 133)) using the corrected, plausible formula. If a fine-tuned, *wrong* model can reproduce the expected phenomenon, what does that imply for computational neuroscience?

CHAPTER

THIRTEEN

EXAMPLE: LOCAL FIELD POTENTIAL

In this example, we show how one can sample the Local Field Potential around a neuron, by extracting the necessary data from the simulation and applying the LFP calculations for the desired sample points.

If one adds a population of neurons, does the relevant changes and focuses the sample points closer, the same methods can also yield the multi-unit activity (mind the simulation timestep though).

First of all, let's install the required software, if on certain platforms like Colab that run the bare notebooks:

```
import os; from pathlib import Path
if 'COLAB_BACKEND_VERSION' in os.environ:
  !TMP=$(mktemp -d); git clone https://eden-simulator.org/repo --depth 1 -b␣
↪development "$TMP"; cp -r "$TMP/." .; rm -rf "$TMP"
  exec(Path('.binder/install_livenb.py').read_text())
if 'DEEPNOTE_PROJECT_ID' in os.environ: exec(Path('../.binder/install_livenb.py').
↪read_text())
```

13.1 Get the model

Let's see what the LFP looks like for the L5 pyramidal cell from Mainen et al. 1995[258]:

```
# Download the zip file with the model
import urllib.request
urllib.request.urlretrieve('https://codeload.github.com/OpenSourceBrain/MainenEtAl_
↪PyramidalCell/zip/refs/heads/master', 'Mainen95.zip')
# and unpack it
from zipfile import ZipFile
with ZipFile('Mainen95.zip', 'r') as zipp: zipp.extractall()
# Check which files are present now
# from os import listdir; listdir()
```

Open the folder (through the GUI or `os.listdir`) and locate the `NeuroMLv2` folder.

```
nml_dir = 'MainenEtAl_PyramidalCell-master/neuroConstruct/generatedNeuroML2/'
# listdir(nml_dir)
```

[258] https://github.com/OpenSourceBrain/MainenEtAl_PyramidalCell

13.2 Prepare and run the simulation

We'll run the existing setup with a lone neuron and a DC pulse clamp on it, but we'll make a whole new `<Simulation>` file to capture all points of the neuron.

First analyse the cell to extract the compartments this neuron contains:

```
net_filename = nml_dir+"MainenEtAl_PyramidalCell.net.nml"
celltype_name = 'MainenNeuroML'
import eden_simulator
cells_info = eden_simulator.experimental.explain_cell(net_filename)
cell_info = cells_info[celltype_name]
```

```
comp_mid_seg = cell_info['comp_midpoint_segment']
comp_mid_fra = cell_info['comp_midpoint_fractionAlong']
n_comps = len(comp_mid_seg) # Number of actual compartments in this cell, may be␣
↪retrieved from most cell_info arrays
print(f'Compartments: {n_comps}')
```

```
Compartments: 739
```

And then generate the new simulation file with its traces:

```
network_name = 'network_MainenEtAl_PyramidalCell';
population_name = 'NeuroMLBased'; N_cells = 1
# Set a new routine for making the <Simulation> file with new recording options
def NmlSimParms(network_name, run_time, run_timestep=25e-6, sampling_period=1e-3,
            rec_lines='', href='file://results.gen.txt'):
    return f'''
        <Simulation id="sim1" length="{run_time}s" step="{run_timestep}s" target="
↪{network_name}">
            <EdenOutputFile id="first" href="{href}" format="ascii_v0" sampling_
↪interval="{sampling_period} s">
            '''+'\n\t        '.join(rec_lines)+'''
            </EdenOutputFile>
        </Simulation>
        <Target component="sim1"/>'''

# Make the traces to record each compartment, and save the file
traces = [f"{population_name}[{neu}]/{celltype_name}/{comp_mid_seg[i]}{('%.9f'
↪%comp_mid_fra[i])[1:]}/v"
    for neu in range(N_cells) for i in range(len(comp_mid_seg))]
rec_lines = [f'<OutputColumn id="v_{i}" quantity="{x}"  output_units="mV"/>' for i,
↪x in enumerate(traces) ]
print(f"recording {len(rec_lines)} waveforms this time !")
sim_filename = nml_dir+"/SimLFP.xml"
with open(sim_filename, "w") as f:
    f.write('<?xml version="1.0" encoding="UTF-8"?>\n<Lems>\n<include file=
↪"MainenEtAl_PyramidalCell.net.nml"/>\n')
    f.write(NmlSimParms(network_name, run_time=0.06, run_timestep=10e-6, sampling_
↪period=0.1e-3, href='moresults.gen.txt', rec_lines=rec_lines))
    f.write('\n</Lems>\n')
# !cat $nml_dir/SimMore.xml
```

```
recording 739 waveforms this time !
```

And run the simulation:

```
%time moresults = eden_simulator.runEden(sim_filename)
timevec = moresults['t']
```

```
CPU times: user 48.7 ms, sys: 13.9 ms, total: 62.6 ms
Wall time: 723 ms
```

Since we'll be doing physics with the simulation's results so that we can calculate the LFP, let's convert the V_m data to a physical `Quantity` with the help of the `pint` Python package[259] for unit math.

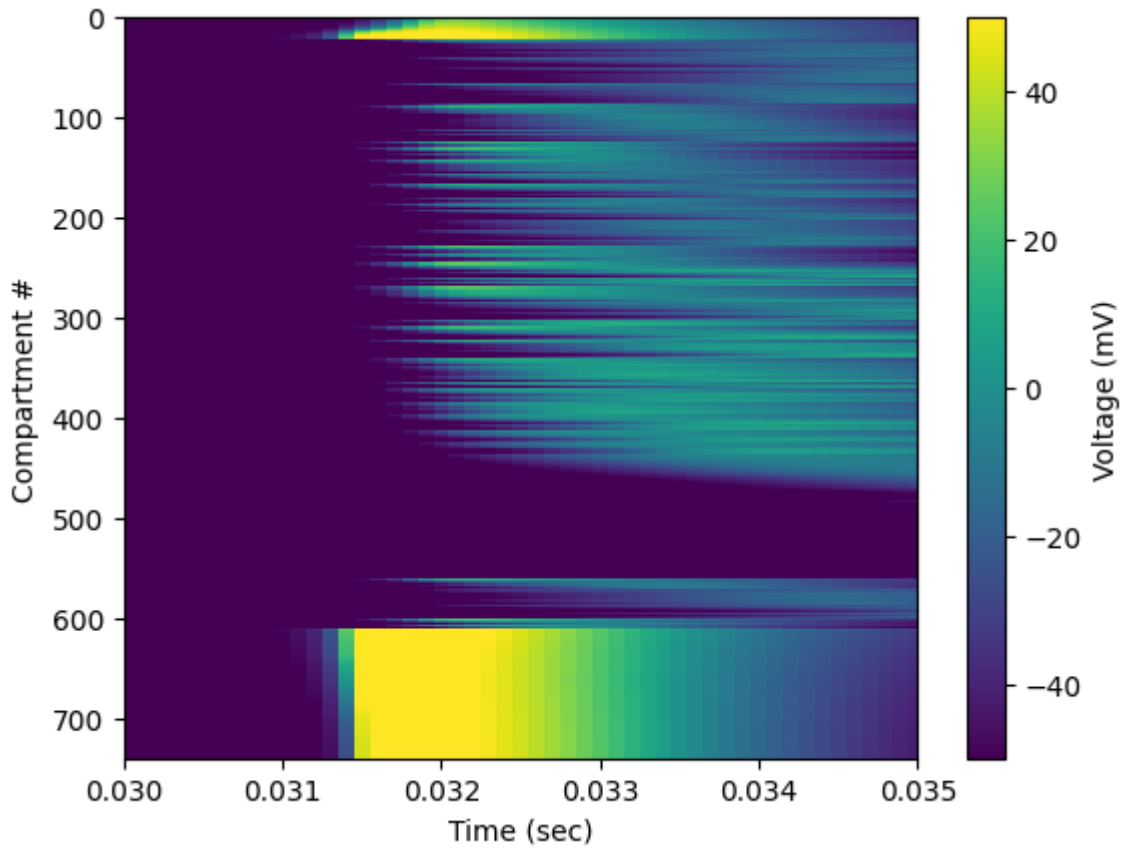
```
!pip install -q pint
```

```
import numpy as np
from pint.registry import Quantity as Q
neuron_membrane_voltage = np.array([moresults[x] for i,x in enumerate(traces)])
neuron_membrane_voltage = Q(neuron_membrane_voltage,'mV')
print("%d points in space, sampled over %d points in time." % neuron_membrane_
↪voltage.shape)
```

```
739 points in space, sampled over 601 points in time.
```

Let's see a raster of V_m per compartment:

```
from matplotlib import pyplot as plt
im = plt.imshow(neuron_membrane_voltage.m_as('mV'),
    extent=[ timevec[0], timevec[-1], n_comps, 0 ],
    aspect='auto', interpolation='none', cmap ="viridis")
cbar = plt.colorbar(im); cbar.set_label('Voltage (mV)')
plt.clim((-50, +50)); plt.ylabel('Compartment #'); plt.xlabel('Time (sec)');
plt.xlim([0.03,0.035]);
```



And an animation of said V_m:

```
from eden_simulator.display.animation import subsample_trajectories
samples_vm, anim_axis_vm, sampled_time_vm, (sampled_voltage,) = subsample_
↪trajectories(
    timevec, [neuron_membrane_voltage.m_as('mV').T], animation_speed=0.0048,␣
```

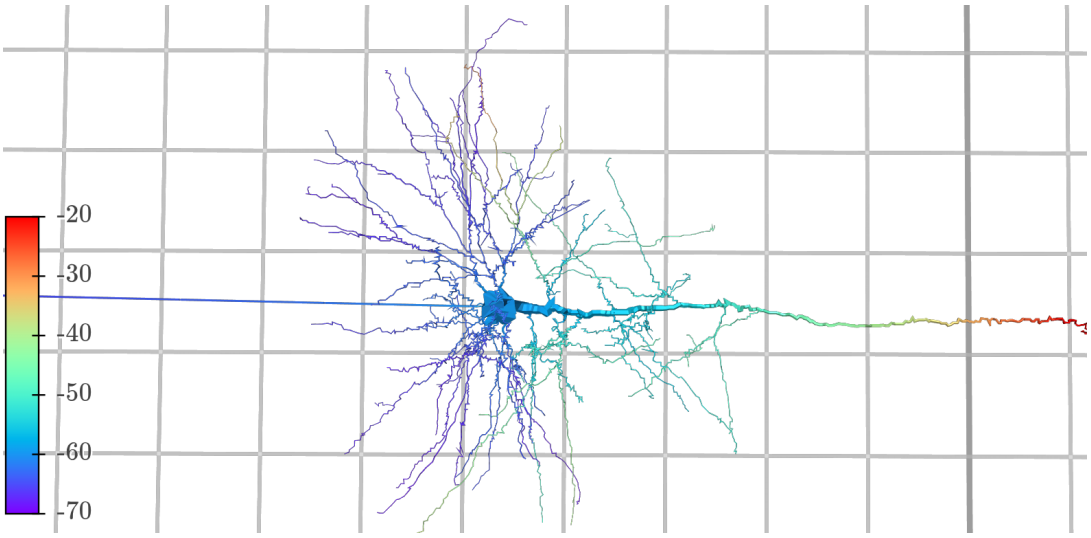
(continues on next page)

[259] https://pint.readthedocs.io/en/stable/user/defining-quantities.html

(continued from previous page)

```
↪animation_frames_per_second=24)

import k3d; from eden_simulator.display.spatial.k3d import Plot, plot_neuron
Vm_plot = plot = Plot()
plot += plot_neuron(cell_info, sampled_voltage, time_axis_sec=anim_axis_vm,
                            color_range=[-70, -20],color_map='rainbow');
plot.camera = plot.get_auto_camera(pitch=30, yaw=10)[:6]+[0,1,0] # set 'y' to up !␣
↪vecs are pos, tgt, up
k3d_label_dict = { str(real_time): f't = {sim_time*1000:.1f} ~ ms' for (real_time,␣
↪sim_time) in zip(anim_axis_vm, sampled_time_vm)  }
plt_label = k3d.text2d(k3d_label_dict, (0.,0.)); plot += plt_label # add 2d␣
↪elements AFTER setting auto camera
plot.show_html()
```

```
<IPython.core.display.HTML object>
```



We have recorded V_m all over the cell and we'll use it (along with the cell's properties) to obtain the resulting LFP.

13.3 Post-process simulation data to get I_m

In this tutorial we'll be looking at the LFP, hence we need the *trans-membrane current* I_m which flows to/from *extracellular space*. Each compartment's electric dynamics are:

$$C\dot{V}_m = I_{axial} + I_{transport} + I_{injected} \; ; \; I_m = C\dot{V}_m - I_{transport}$$

where $I_{transport}$ flows between the cytoplasm and extracellular space (e.g. ion channels and passive leaks), $I_{injected}$ flows between the cytoplasm and places *other than* extracellular space or adjacent compartments (e.g. patch clamps), and $V_m is measured from the *inner* to the *outer* side of the membrane.

To avoid supreme confusion, in the following we will consistently measure positive capacitive current $C\dot{V}_m$ as coming OUT of the compartment and into extracellular space, and all other currents coming INTO the compartment as positive. Hence capacitive current equals (is opposed to) everything else. We will also measure the total membrane current Im coming OUT of the compartment; current locally flowing OUT will cause a positive LFP in the vicinity by Ohm's law (even though voltage *drop* goes along the current flow, potential at infinity = ground = zero, remember that), and current locally flowing IN will make it negative.

Keep in mind the following **important** points:

- It might look strange but I_m includes the *capacitive* current of the membrane (dis)charging. If you're not convinced, draw a circuit diagram[260] of the compartments and circle the ion channels and capacitor in parallel.

[260] https://en.wikipedia.org/wiki/File:Cable_theory_Neuron_RC_circuit_v3.svg

What current comes into the intracellular node (axial currents and patch clamps) must equal what goes out to extracellular space, following Kirchhoff's law of currents.

- Although direct injections cross the membrane (topologically), they don't count towards the LFP because charge travels from inside a body to inside another; thus the extracellular environment is not affected. Consider: If we bring the neuron into a steady state using a current clamp, $I_{transport} + I_{\text{injected}} = 0$ (modulo spatial details), $I_m = -I_{patch}$ and the LFP is very real: the neuron itself becomes the probe we pump current through.
- Whether a stimulus from the experimental rig should count as trans-membrane current or direct injection is ultimately up to the modeller's *intention*; due caution is advised! For example: is a "virtual synapse" stimulus imposed by a patch clamp, or a simulated synapse?

Having said all that, we can measure I_m by either measuring $C\dot{V}_m - I_{transport}$, or $I_{\text{axial}} + I_{\text{injected}}$. Which one is easier?

Well, presently, NeuroML doesn't provide for directly recording any of these. But not all is lost: thanks to EDEN's cell analysis feature (page 241) we can deduce capacitive and axial current. And we should know clamp current, since we are the ones who put them (right?); even if it's a dynamic clamp that we defined with LEMS (page 61) we can record the resulting current. This leaves out the current from trans-membrane mechanisms (say, channels) which takes the bulk of computing. Let's leave it then to EDEN and work with axial + injected current.

One last wrinkle in our plans is then, where in the world are injected currents located? One could fish the exact `<segment>` out of the NeuroML file - but remember, we discretise those into compartments so we want the actual *compartment* that the clamping sites fall on. Appropriate helper routines for that will be available soon, but for now let's exploit the facts that a) most people inject in the soma, which is usually 2) a single compartment, the very first one.

13.3.1 Calculating axial current

Just multiply membrane potential by the conductance matrix joining neurons. This matrix is so generally useful (see also the imposed field (page 159) example) that a helper for Python will soon provide it.

```
# First, get the axial resistance matrix: Ia = G*V. It follows the tree structure
comp_ga = cell_info['comp_conductance_to_parent'] # provided in nS
iii = []; jjj = []; vvv = [] # Construct the lists (i,j,v)
for i,p in enumerate(cell_info['comp_parent']):
    g = comp_ga[i]
    if p >= 0:
        iii += [ i, p, i, p] # add 4 sparse elms in one go
        jjj += [ p, i, i, p]
        vvv += [+g,+g,-g,-g] # same as:
        # axial_conductance_matrix[i,p] += g
        # axial_conductance_matrix[p,i] += g
        # axial_conductance_matrix[i,i] -= g
        # axial_conductance_matrix[p,p] -= g

import scipy
G = scipy.sparse.coo_matrix((vvv,(iii,jjj))).tocsr() # needs csr for some reason
G = Q(G,'nS')

axial_current = (G.m @ neuron_membrane_voltage.m)*G.units*neuron_membrane_voltage.
↪units
```

13.3.2 Calculating injected current

As explained previously, we'll have to figure that out from our model for the time being. Oh well…

```
comp_Iclamp = Q(np.zeros_like(neuron_membrane_voltage), 'pA')
# facts of life for github.com/OpenSourceBrain/MainenEtAl_PyramidalCell - change␣
↪accordingly.
comp_Iclamp[0,(3e-3 < timevec)&(timevec <= 43e-3+11e-6)] = Q(160,'pA')
# NOTE: Check the inequalities.
# The effect of Im will appear on the timestep AFTER it nominally starts and ends,
#  as the capacitive and axial currents can't have reacted to it before it started!
# Might as well apply a highpass or median filter since the simulation timestep␣
↪should be less than 25 µs.
```

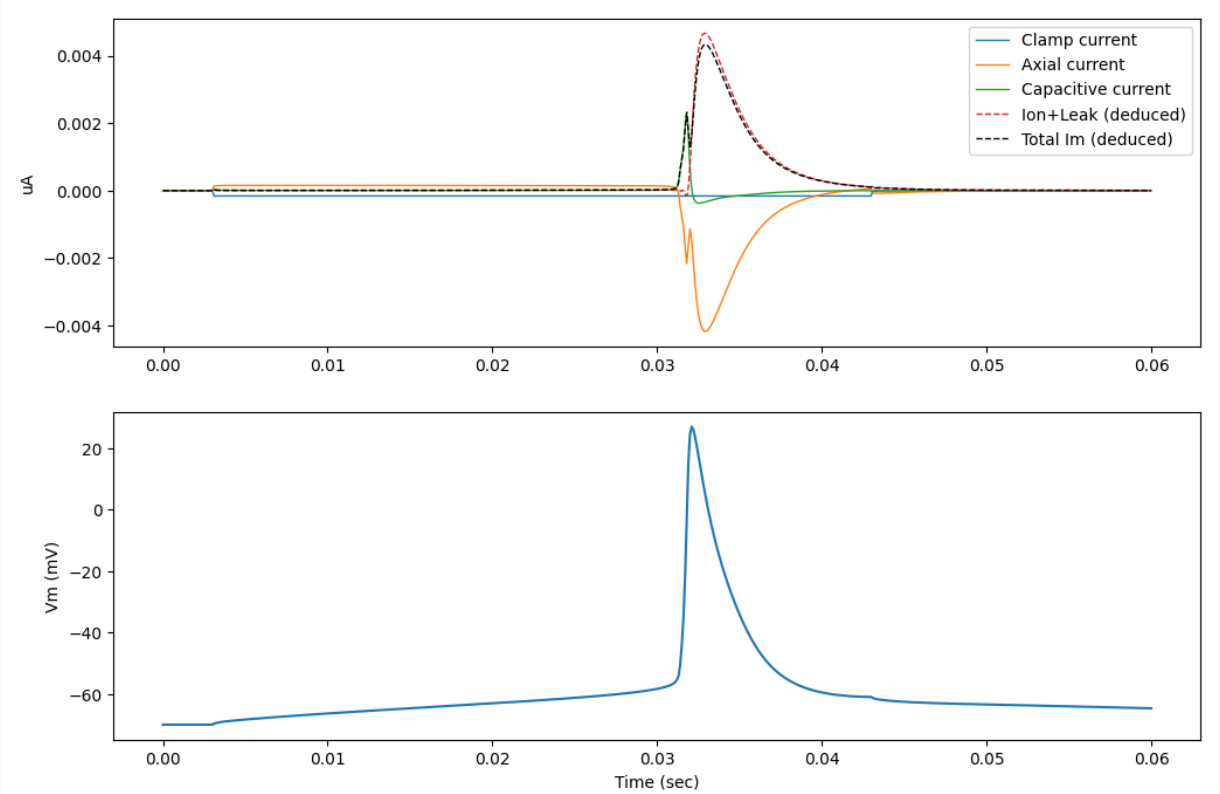
And check that the total membrane current makes sense on, say, the soma.

13.3.3 Calculating membrane current

Add them up and check that the total membrane current makes sense on, say, the soma.

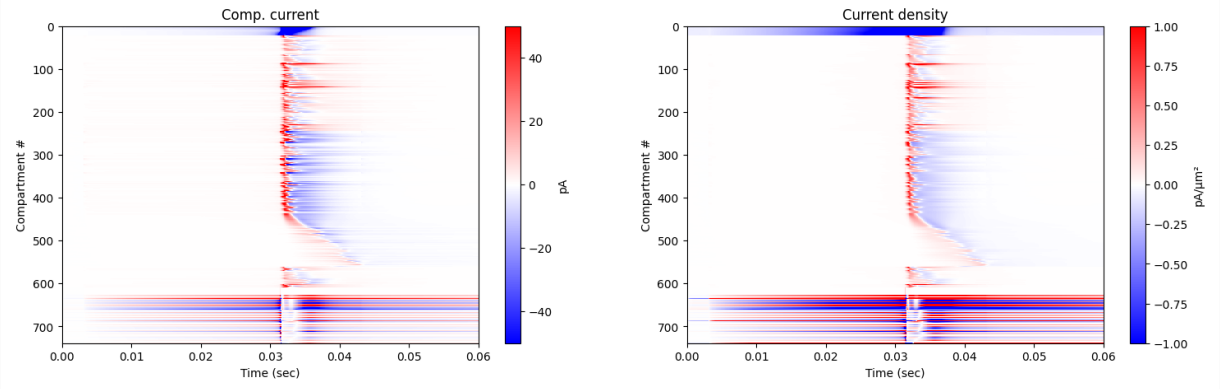
```
# Icap + Iion + Iax + Iclamp = 0 (outward); Im = Icap + Iion = -Iax -Iclamp␣
↪(outward)
neuron_membrane_current = -(-comp_Iclamp + -axial_current) # going outward!
```

```
# Let's see the breakdown
comp_capa = Q(cell_info['comp_capacitance'], 'pF')
capa_current = comp_capa[:,None] * np.diff(neuron_membrane_voltage, axis=-1)/Q(np.
↪diff(timevec),'s') # outward!
ion_current = (neuron_membrane_current[:,1:]+neuron_membrane_current[:,:-1])/2 -␣
↪capa_current # outward!

comp = 0; xlim = [0.049,0.056]
fig, (ax1, ax2) = plt.subplots(2,1,figsize=(12,8), dpi=100)
ax1.plot(timevec,-comp_Iclamp[comp].m_as('uA'), '-',lw=1,label='Clamp current');
ax1.plot(timevec,-axial_current[comp].m_as('uA'), '-',lw=1,label='Axial current');
ax1.plot((timevec[1:]+timevec[:-1])/2,capa_current[comp].m_as('uA'), '-',lw=1,␣
↪label='Capacitive current');
ax1.plot((timevec[1:]+timevec[:-1])/2,ion_current[comp].m_as('uA'), '--',lw=1,␣
↪label='Ion+Leak (deduced)');
ax1.plot(timevec,neuron_membrane_current[comp].m_as('uA'),'k--',lw=1,label='Total␣
↪Im (deduced)');
ax1.set_ylabel('uA'); ax1.legend()
# ax1.set_xlim(xlim)
ax2.plot(timevec,neuron_membrane_voltage[comp].m_as('mV')); ax2.set_ylabel('Vm (mV)
↪'); ax2.set_xlabel('Time (sec)');
# ax2.set_xlim(xlim)
```



Let's see the total current flow for each compartment… On second thought, that's absolute numbers for unevenly sized compartments. The *density* over the membrane is perhaps more informative.

```
fig, axs = plt.subplots(1,2, figsize=(18,5))
im = axs[0].imshow(neuron_membrane_current.m_as('pA'),
    extent=[ moresults['t'][0], timevec[-1], n_comps, 0 ],
    aspect='auto', interpolation='none', cmap ="bwr");
cbar = plt.colorbar(im); cbar.set_label('pA')
im.set_clim((-50, +50)); axs[0].set_ylabel('Compartment #'); axs[0].set_xlabel(
↪'Time (sec)');
axs[0].set_title("Comp. current");

comp_area = Q(cell_info['comp_area'], 'um*um')
im = axs[1].imshow((neuron_membrane_current/comp_area[:,None]).m_as('pA/um**2'),
    extent=[ moresults['t'][0], timevec[-1], n_comps, 0 ],
    aspect='auto', interpolation='none', cmap ="bwr");
cbar = plt.colorbar(im); cbar.set_label('pA/µm²')
im.set_clim((-1, +1)); axs[1].set_ylabel('Compartment #'); axs[1].set_xlabel('Time␣
↪(sec)');
axs[1].set_title("Current density");
```



Observe that some compartments are much more affected by axial currents (≈ membrane currents!) than others.
Generally, the less area they have, and the tighter they are coupled to adjacent compartments, the more affected they are by axial current. (Cytoplasmic resistivity `Ra` and membrane spec. capacitance `Cm` can also matter when non-uniform.)
Feel free to plot `comp_ga` and `comp_capa` (and the ratio thereof `1/RC`) to see the sensitivity of each component.

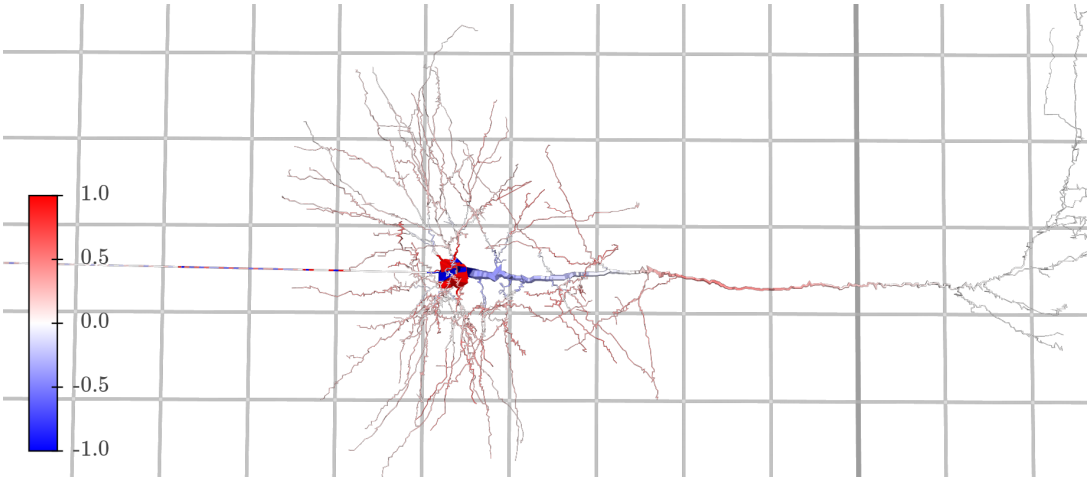
```
# intentionally left blank
```

13.3.4 Visualisation

Let's see then how the trans-membrane current evolves on the neuron. We'll use the same Red-White-Blue colormap we'll display the LFP with, in the following.

```
# resample the anim_axis because Im has 1 time point less than Vm, due to np.diff
samples_im, anim_axis_im, sampled_time_im, [sampled_current_density] = subsample_
↪trajectories(
    timevec[:-1], [(neuron_membrane_current / comp_area[:,None]).m_as('pA/um**2').
↪T], animation_speed=0.0048, animation_frames_per_second=24)

Im_plot = plot = Plot()
plot += plot_neuron(cell_info, sampled_current_density, time_axis_sec=anim_axis_im,
                            color_range=[-1, +1],color_map='bwr');
plot.camera = plot.get_auto_camera(pitch=30, yaw=10)[:6]+[0,1,0] # set 'y' to up !␣
↪vecs are pos, tgt, up
plot.show_html()
```

```
<IPython.core.display.HTML object>
```



13.4 Post-process I_m to get the LFP

The other half of calculating the LFP is how much each point outside the cell is being affected by the current flow on each part of the neuron (that is, mostly by the closest compartments, but also slightly from the more distant ones).

The usual assumptions apply for this example: The extracellular medium is uniform, highly conductive, isotropic and unobstructed until far away. If you need a more sophisticated model, apply the relevant adjustments to the following. (Perhaps a separate simulation process just for the LFP is called for, see the other articles TBD on "Pipelines" generally.)

Just for illustration, we'll use one element per compartment. It would be even better if each NeuroML `<segment>` is applied separately, if it's not too much for the computer.
Alternatively, the LFPykit[261] could also be used to calculate the coupling matrix.

```
extracell_conductivity = Q(.3, 'S/m')

source_points = cell_info['comp_midpoint'][:]
x_space = np.linspace(-360, 000, 10); y_space = np.linspace(-180, 600, 30); z_
↪space = [-30]
x, y, z = np.meshgrid(x_space, y_space, z_space, indexing='ij')
sampling_points = np.stack((x,y,z),axis=-1).reshape((-1,3))
# print("Sampling points:", sampling_points)

# Vectorize calculation of pairwise distance matrix https://jaykmody.com/blog/
↪distance-matrices-with-numpy/
distmat =  source_points @ sampling_points.T
distmat = np.sum(sampling_points**2,axis=-1) - 2*distmat + np.sum(source_points**2,
↪axis=-1)[:,None]
distmat = np.sqrt(distmat)
distmat = Q(distmat, 'um')

# LATER: Exclude current from points inside the cell, thereby avoiding␣
↪singularities as a bonus.
coupling = (1 / (4*np.pi*extracell_conductivity*distmat)).to('uV/pA')
# NB: Coupling is positive *although* potential *drops* along the electric field's␣
↪lines! Think of it this way:
#     Since potential is 0 at infinity, the potential any closer to the source has␣
↪to be positive! Thus I/4πσr

print('Coupling is calculated for %d current sources times %d sampling points.' %␣
↪coupling.shape)
def get_lfp(membrane_current): return ((coupling.m.T @ membrane_current.
↪m)*coupling.units*membrane_current.units).to('uV')
# get_lfp(neuron_membrane_current[:,0])
```

```
Coupling is calculated for 739 current sources times 300 sampling points.
```

13.5 Visualisation

13.5.1 In 4-D

Now let's show the membrane voltage and the observed LFP together in one animation.

```
# Animate the lfp, similar to label but with a vector instead of a string
lfp_uvolt = get_lfp(neuron_membrane_current).m_as('uV').T
k3d_lfp_dict = { str(real_time): x.astype(np.float32)
                for (real_time, x) in zip(anim_axis_vm, lfp_uvolt[samples_im])  }
```

```
LFP_plot = plot = Plot()

lfp_scale_range = [-10, 10] #other options to see transitions vs. absolute range:␣
↪[-.2, .2], [-.5, +.5] ...
plt_points = k3d.points(positions=sampling_points.astype(np.float32),attribute=k3d_
↪lfp_dict,
        color_range=lfp_scale_range,point_size=10,color_map=k3d.matplotlib_color_
↪maps.Bwr); plot += plt_points
```

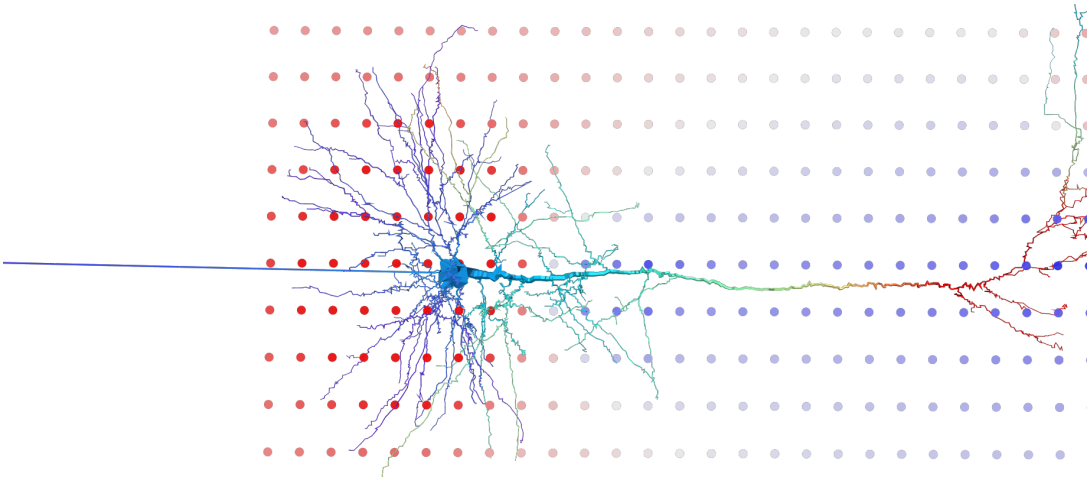
(continues on next page)

[261] https://github.com/LFPy/LFPykit

(continued from previous page)

```
plot.camera = plot.get_auto_camera(pitch=30, yaw=10)[:6]+[0,1,0] # set 'y' to up !␣
↪vecs are pos, tgt, up
plt_mesh = plot_neuron(cell_info, sampled_voltage, time_axis_sec=anim_axis_vm,
                       color_range=[-70, -20],color_map='rainbow', compress_
↪cells=False);
plot += plt_mesh
plt_label = k3d.text2d(k3d_label_dict, (0.,0.)); plot += plt_label # add 2d␣
↪elements AFTER setting auto camera
plot.show_html()
```

```
<IPython.core.display.HTML object>
```



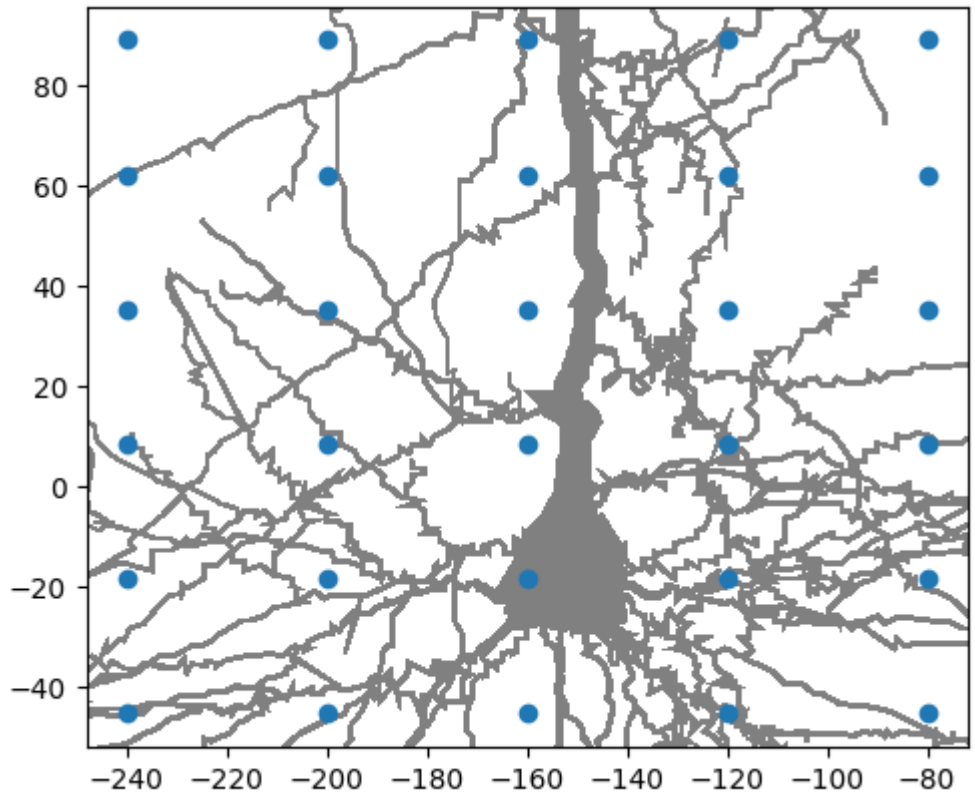
The dots around the neuron are the sampling points for the LFP, they don't necessarily have to be a regular grid (but it's easier to understand this way).
They are coloured with red ❤️ for positive potential, blue 💙 for negative, and white 🤍 for zero.

13.5.2 Another popular way

The geekier among us may appreciate a 2-D grid of sampled LFP over time, with the morphology in the background for reference. As seen in Holt and Koch (1999)[262] figures 3 and 4, for a similar neuron type.

```
# Grab the sampling points in an area near the soma
s = sampling_points
marked = (-240 <= s[:,0]) & (s[:,0] <= -60) & (-50 <= s[:,1]) & (s[:,1] <= 100)
cs = s[marked]

# What are we looking at ?
import matplotlib as mpl; from matplotlib import pyplot as plt
fig,ax = plt.subplots()
ax.scatter(cs[:,0], cs[:,1], zorder=10)
ax.set_aspect('equal'); plt.autoscale()
xlim = ax.get_xlim(); ylim = ax.get_ylim();
plt.tripcolor(cell_info['mesh_vertices'][:,0], cell_info['mesh_vertices'][:,1],␣
↪cell_info['mesh_faces'], facecolors=[0]*len(cell_info['mesh_faces']), cmap='gray
↪', clim=[-1,+1],zorder=0)
ax.set_xlim(xlim);ax.set_ylim(ylim);
```

[262] http://doi.org/10.1023/A:1008832702585



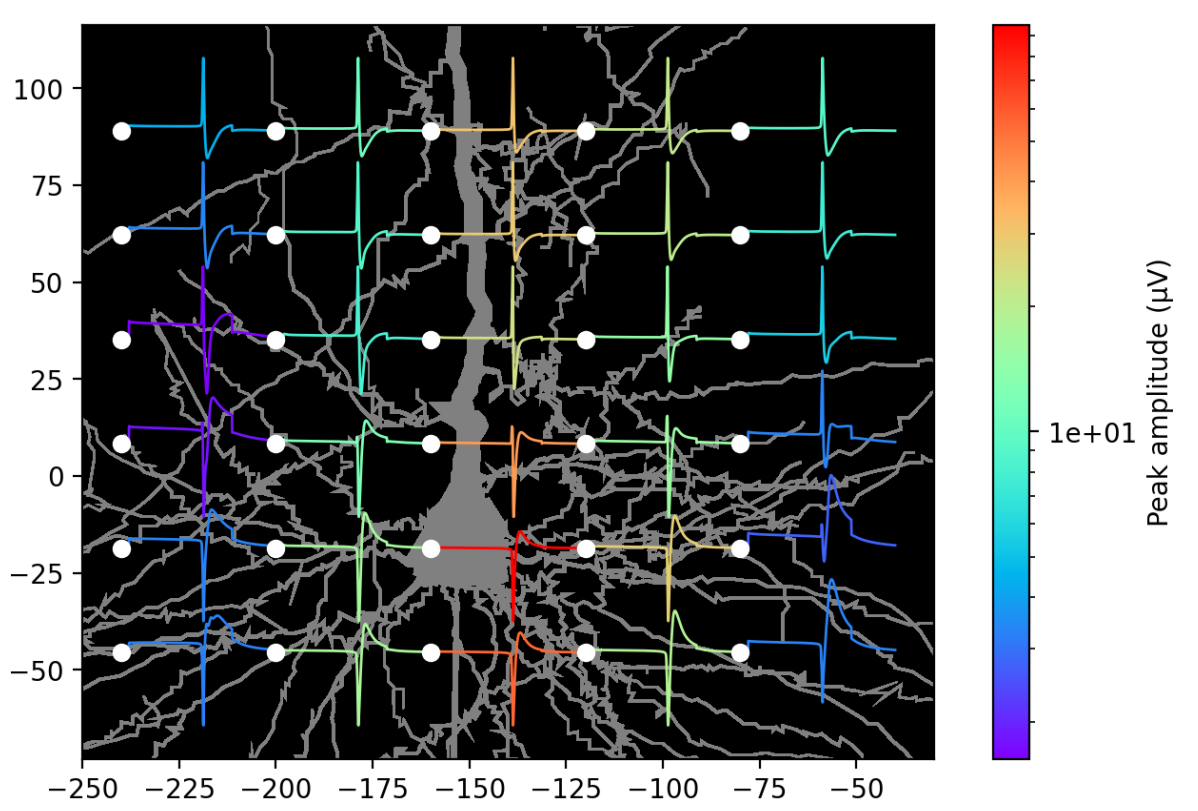
Now we're using an orthographic projection, we can see clearer that the morphology a bit jaggy and rectangular. Likely so because it was traced in the early 1990s. One can hope that the excess tortuosity was accounted for in the biophysics. Anyway, it's still more realistic than a stick figure so there.

```
# Keep track of the 2D sampling space
xoff_step = np.diff(x_space)[0]; yoff_step = np.diff(y_space)[0]
xoff, yoff = cs[:,0], cs[:,1]

# Normalise by peak magnitude, to show spikes in any mangitude. NB: aliasing can␣
↪be an issue if the sampling period is more (e.g. ~msec) than the peak's duration.
lfp_peak_mag_uV = np.max(np.abs(lfp_uvolt[:,marked]), axis=0)
minmag_uV = np.min(lfp_peak_mag_uV); maxmag_uV = np.max(lfp_peak_mag_uV)

# Set up logarithmic colormapping (or not)
max_range_db = 60 # 1:1000 amplitude
mynorm = mpl.colors.LogNorm(vmin=max(minmag_uV,maxmag_uV/(10**(max_range_db/20))),␣
↪vmax=maxmag_uV)
cmap = mpl.colormaps['rainbow']
lfp_peakcols = cmap(mynorm(lfp_peak_mag_uV))

# Draw the dot positions, to also cue the colorbar (see also visible=false)
fig,ax = plt.subplots(dpi=200, figsize=np.array([8,5])*1); ax.set_facecolor("k")
dots = ax.scatter(xoff, yoff, c=lfp_peak_mag_uV, norm=mynorm, cmap=cmap,
↪visible=False);
_ = ax.scatter(xoff, yoff, c='w',zorder=10);  plt.colorbar(dots, format='%.00g',␣
↪label='Peak amplitude (µV)')

ax.set_prop_cycle(color=lfp_peakcols)
ax.plot(xoff[None]+(xoff_step*timevec[:,None]/timevec[-1]), lfp_uvolt[:,
↪marked]*yoff_step*.7/lfp_peak_mag_uV + yoff[None], lw=1); # Draw normalised␣
↪lines...
# plt.plot(xoff[None]+(xoff_step*timevec[:,None]/timevec[-1]), lfp_uvolt[:,marked]␣
```

(continues on next page)

(continued from previous page)

```
↪+ yoff[None],lw=1); # ... or not
ax.set_aspect('equal'); plt.autoscale()
xlim = ax.get_xlim(); ylim = ax.get_ylim(); # don't autoscale
ax.tripcolor(cell_info['mesh_vertices'][:,0], cell_info['mesh_vertices'][:,1],␣
↪cell_info['mesh_faces'], facecolors=[0]*len(cell_info['mesh_faces']), cmap='gray
↪', clim=[-1,+1],zorder=0)
ax.set_xlim(xlim);ax.set_ylim(ylim);
# Exercise: show only the time span near the spike.
```



13.5.3 More ideas

The enterprising reader may find more nuances and improved ways of calculating the LFP in the work of, among others, Einevoll and his lab. (Neuron users: This is the `rec` part of the popular extracellular_stim_and_rec[263] scripts.)

For higher detail, consider using non-uniform sampling grids, centered around where the potential changes most sharply and where the (both stimulation and recording) electrodes are located.

If storing all voltage traces is becoming a burden, one can calculate the coupling matrix as they prefer, and add up the numbers using virtual "gap junctions" that point (page 107) to membrane current and insert a custom flux (page 121) to a "readout" fake point neuron. (See also pretty much the same being done on Brian[264]). For really large numbers of current sources, it might become more efficient to stream values with a pipeline to the Barnes-Hut algorithm or the fast multipole method.

[263] https://web.archive.org/web/20100730180342/http://neuron.yale.edu/phpBB/viewtopic.php?f=28&t=168

[264] https://brian2.readthedocs.io/en/stable/examples/compartmental.lfp.html

CHAPTER

FOURTEEN

EXAMPLE: IMPOSED ELECTRIC AND MAGNETIC FIELDS

This article explains how to model the effect of an external, quasi-static electric field $\vec{E}$ on a spatially-detailed neuron model (page 33). The field is assumed to be perfectly *imposed* outside the neuron, as is the case for high-intensity fields and high-conductivity surrounding media (such as a lonely neuron on a dish, or whenever the effects of the surroundings can be just simplified away without much loss).

First of all, let's install the required software, if on certain platforms like Colab that run the bare notebooks:

```
import os; from pathlib import Path
if 'COLAB_BACKEND_VERSION' in os.environ:
  !TMP=$(mktemp -d); git clone https://eden-simulator.org/repo --depth 1 -b␣
↪development "$TMP"; cp -r "$TMP/." .; rm -rf "$TMP"
  exec(Path('.binder/install_livenb.py').read_text())
if 'DEEPNOTE_PROJECT_ID' in os.environ: exec(Path('../.binder/install_livenb.py').
↪read_text())
```

14.1 Get a cell model

For example, from Mainen et al. 1995[265]:

```
# Download and unpack
import urllib.request; from zipfile import ZipFile
urllib.request.urlretrieve('https://codeload.github.com/OpenSourceBrain/MainenEtAl_
↪PyramidalCell/zip/refs/heads/master', 'Mainen95.zip')
with ZipFile('Mainen95.zip', 'r') as zipp: zipp.extractall()
nml_dir = 'MainenEtAl_PyramidalCell-master/neuroConstruct/generatedNeuroML2/'
cell_filename = nml_dir+"MainenNeuroML.cell.nml"
```

And analyse it with EDEN:

```
celltype_name = 'MainenNeuroML'
import eden_simulator
cells_info = eden_simulator.experimental.explain_cell(cell_filename)
cell_info = cells_info[celltype_name]
comp_mid_seg = cell_info['comp_midpoint_segment']
n_comps = len(comp_mid_seg)
print(f'Compartments: {n_comps}')
xyz = cell_info['comp_midpoint'].T # break down by coordinate
```

```
Compartments: 739
```

[265] https://github.com/OpenSourceBrain/MainenEtAl_PyramidalCell

14.2 Prepare to run simulations

We'll re-use the following code for a single-neuron model to apply the equivalent I_{ext} of the imposed field (as seen in the following), and record and display the resulting V_m, over all parts of the neuron.

```
cell_pop_name = 'Pop'
tabline = '\n        '
comp_locators = eden_simulator.experimental.GetLemsLocatorsForCell(cell_info)
cell_voltage_paths = [ f'{cell_pop_name}[0]/{loc}/v' for loc in comp_locators ]
def MakeNmlModel(model_name, cell_info, Iext_nA, pulse_sec):
    assert(pulse_sec[1]>pulse_sec[0])
    model_file = f'''<neuroml>
        <include href="{cell_filename}"/>
        <pulseGenerator id="pulseNamp" delay="{pulse_sec[0]} s" duration="{pulse_
↪sec[1] - pulse_sec[0]} s" amplitude="1 nA"/>
        <network id="Net_{model_name}">
            <population id="{cell_pop_name}" component="{celltype_name}" size="1" /
↪>

            <inputList id="ElectrodeStim" component="pulseNamp" population="{cell_
↪pop_name}">
            {tabline.join([ f'<inputW id="{comp}" target="{cell_pop_name}[0]"␣
↪weight="{curr}"'
                           +f' segmentId="{cell_info["comp_midpoint_segment
↪"][comp]}"'
                           +f' fractionAlong="{cell_info["comp_midpoint_
↪fractionAlong"][comp]}" destination="synapses"/>'
                           for comp, curr in enumerate(Iext_nA)])}
            </inputList>
        </network>
    </neuroml>'''
    with open(f'Net_{model_name}.nml', 'wt') as f: f.write(model_file)

def MakeNmlSimFile(model_name, sim_duration_sec):
    tabline = '\n        '
    sampling_period_sec = 0.1e-3
    sim_file = f'''<Lems>
    <Include href="Net_{model_name}.nml"/>
    <Simulation id="MySim" length="{sim_duration_sec} sec" step="10 usec" target=
↪"Net_{model_name}">
        <EdenOutputFile id="OutFile" href="file://Results_{model_name}.gen.txt"␣
↪format="ascii_v0" sampling_interval="{sampling_period_sec} s">
            {tabline.join([ f'<OutputColumn id="v_{loc}"  quantity= "{path}"␣
↪output_units="V"/>' for loc, path in zip(comp_locators, cell_voltage_paths)])}
        </EdenOutputFile>
    </Simulation>
    <Target component="MySim"/></Lems>'''
    sim_filename = f'LEMS_Sim_{model_name}.xml'
    with open(sim_filename, 'wt') as f: f.write(sim_file)
    return sim_filename
```

Since we'll be doing physics with the simulation's results, let's add physical units with `pint`.

```
#!pip install -q pint
def ResultsToVm(results):
    import numpy as np
    from pint.registry import Quantity as Q
    neuron_membrane_voltage = np.array([results[path] for path in cell_voltage_
↪paths])
    neuron_membrane_voltage = Q(neuron_membrane_voltage,'V')
    return neuron_membrane_voltage
```

Let's make a 2D plot of V_m per compartment:

```
def ShowVmRaster(neuron_membrane_voltage):
    from matplotlib import pyplot as plt
    im = plt.imshow(neuron_membrane_voltage.m_as('mV'),
        extent=[ *rec_time_axis_sec[[0,-1]], n_comps, 0 ],
        aspect='auto', interpolation='none', cmap ="viridis")
    cbar = plt.colorbar(im); cbar.set_label('Voltage (mV)')
    plt.ylabel('Compartment #'); plt.xlabel('Time (sec)');
```

And cool 3D animations in general:

```
import numpy as np; import k3d; from eden_simulator.display.animation import␣
↪subsample_trajectories
def TimeLabel(anim_axis,sampled_time):
    k3d_label_dict = { str(real_time): f't = {sim_time*1000:.1f} ~ ms' for (real_
↪time, sim_time) in zip(anim_axis, sampled_time)  }
    return k3d.text2d(k3d_label_dict, (0.,0.))
animation_speed=0.0048
def makeplot(cell_info, values, rec_time_axis_sec, color_range=[-70, -20], color_
↪map='rainbow', plot=None):
    from eden_simulator.display.animation import subsample_trajectories
    samples_vm, anim_axis, sampled_time, (sampled_voltage,) = subsample_
↪trajectories(
        rec_time_axis_sec, [neuron_membrane_voltage.m_as('mV').T], animation_
↪speed=animation_speed, animation_frames_per_second=24)
    plot = Plot()
    plot += plot_neuron(cell_info, sampled_voltage, time_axis_sec=anim_axis,
                                color_range=color_range, color_map=color_map);
    plot.camera = plot.get_auto_camera(pitch=30, yaw=10)[:6]+[0,1,0] # set 'y' to␣
↪up ! vecs are pos, tgt, up
    plot += TimeLabel(anim_axis,sampled_time) # add 2d elements AFTER setting auto␣
↪camera
    return plot, (anim_axis, sampled_time)
def PointUp(plot):
    plot.camera = plot.get_auto_camera(pitch=30, yaw=10)[:6]+[0,1,0] # set 'y' to␣
↪up ! vecs are pos, tgt, up
```

Now, to the experiment.

14.3 Physics

Let's introduce some electro-physical terms and laws that will help understand the following (or skip them if they're well known):

- A *vector* is, for the purposes of this topic and Euclidean space, a "longer" or "shorter", generally directional property drawn as an arrow. Arrows of "zero" length that are *ineffectual* and *directionless* can exist, we just don't draw them. As mathematical symbols we write vectors down as lower-case names e.g. r but either with an arrow on top $\overrightarrow{r}$, or in boldface $\mathbf{r}$.
  - For instance: Points in space can be expressed as arrows from an arbitrary point of reference or *origin*, to them. Velocity $\mathbf{v}$ can be expressed as its direction, with length in corresponding units of measure.
  - If we (very conveniently) break down a vector to our three accessible dimensions or *axes* of movement (i.e. left-right, up-down and front-back), we can break it down into x, y, z components parallel to each dimension. We can then name *unit vectors* $\mathbf{x}, \mathbf{y}, \mathbf{z}$ for the associated dimensions, and express x, y, z as *real numbers*, that represent multiples of the aforementioned unit vectors. Thus $\forall\, \mathbf{r}\ \exists\, x, y, z \in \mathbb{R}^3 : \mathbf{r} = x\mathbf{x} + y\mathbf{y} + z\mathbf{z}$.
- A field[266] $\overrightarrow{F}(\overrightarrow{r})$ is a mathematical function that maps *locations* to *vectors*. We generally use capital letters

[266] https://en.wikipedia.org/wiki/Vector_field

to name those, to avoid confusion. Fields can represent directional attributes (of e.g. physical phenomena) that exist over places. For every location, there is one particular vector; we draw a few of them to get a rough picture. An easy to understand example of a field is the velocity of a flock, crowd or stampede of animals; nearby animals have about the same velocity lest they bump on each other.

- Electrically conductive materials such as the intracellular (and normally extracellular) environment are full of freely moving charges (of electrons, ions and such). These charges are pushed around by an electric *force field* $\overrightarrow{E}$; its force $\overrightarrow{F}$ is amount of electric charge q times the field's intensity. We can write this down as $\overrightarrow{F} = q\overrightarrow{E}$; the field is hence measured in *Newtons over Coulombs*. (Another better known force field is gravity acting likewise on mass; we just have never observed negative mass.) Within neurons, this field is caused by the electric charge of the membrane (on both sides) and relative differences thereof over space and time. But in this chapter, we'll also add the influence of *external fields* to the mix.
- Movement through a force field imparts or deducts kinetic energy K; movement of a positive charge in the direction of the electric field imparts, movement in the opposite direction deduces, movement sideways does neither. This all is proportional to charge amount and field intensity, and reversed for negative charges (and zeroed for zero charges, field, movement). We can write that down as $dK = q\overrightarrow{E} \cdot d\mathbf{r}$ for small movements $d\mathbf{r}$ (remember that $\mathbf{E}(\mathbf{r})$ changes with $\mathbf{r}$). We can then define the *energy per charge* gained along a certain path L as the $\frac{\Delta}{q} = \int_L \overrightarrow{E} \cdot d\mathbf{r}$. This is what we call the *electric tension* or *voltage* along a path! *Volts = Energy (Joules) over Charge (Coulombs)*. Considering that *Energy* is *Force times Length*, we also measure $\overrightarrow{E}$ more commonly in *Volts over Metres*. Note that the opposite voltage applies over the same path in the opposite direction.
- In some cases, it doesn't matter which *path* a particle makes to move from **a** to **b**; they all will impart the same *energy per charge*. Then we can define a scalar *electric potential* $U(\mathbf{r})$ or $U(x, y, z)$ which represents the voltage between locations and some arbitrary point of reference. Keep in mind that the potential *decreases* in value along the lines of the field! The point of reference doesn't really matter because we should only consider the *difference* from point to point, much like with integrals. Note that the effect of such fields or their "potentials" can be summed linearly when they are present together, as long as they don't interfere with each other (significantly).
- In other cases, the path *does* matter, which complicates things. In fact, a charged particle could get more and more energy by following two different paths from **a** to **b**, one forwards than one backwards in an endless loop! This happens when we're dealing with a *time-varying magnetic field* and the electro-magnetic induction[267] that it causes. In such cases, we can't define an unambiguous $U(x, y, z)$ over space, we'll just have to trace the path being taken. Remember that the charges' path is constrained by the conductors they can pass through, and otherwise driven by the forces applied on them, that can't just go however they please. Hence there is a well defined voltage between two points, along a certain path. But when, say, a wire does go in loops it can accumulate voltage multiple times over (roughly) the same path.
- Even when the path does matter, this doesn't mean that "membrane voltage" and other are meaningless. They remain part of the total field and energy differential, we just add the effect of the field's rotational component[268] on top. The part with a "potential" is due to distributions of electric charge (for instance, between the cell membrane's sides) and the rotational part is due to a changing magnetic field (page 169), as per Maxwell's equations.
- The actual quantity charge *moving* because of the applied forces is usually determined by Ohm's law[269]: the more conductive the material is the more the charges move. We can have an electric field even when nothing moves, observe an unplugged mains socket. As the socket gets plugged sparks may fly because as the conductors get to touch, the difference in potential is across a smaller and smaller distance before it is "moved" to the load after the plug.

[267] https://en.wikipedia.org/wiki/Electromagnetic_induction
[268] https://en.wikipedia.org/wiki/Helmholtz_decomposition#Electrodynamics
[269] https://en.wikipedia.org/wiki/Ohm%27s_law

14.3.1 Classic cable equation

Having said all that, we can apply fields on neurons as follows.

Neural cable theory[270] assumes for each compartment of a neuron:

- two electric nodes, one for inside the membrane and one for outside the membrane. The voltage between them is $V_m = U_{int} - U_{ext}$;
- membrane capacitance between these nodes, and a bunch of trans-membrane channels with reversal-potentials (which can be lumped into a Thévenin equivalent);
- axial resistance between the *inner nodes* of adjacent compartments. $I_{a,ij} = \frac{U_{int,i} - U_{int,j}}{R_{a,ij}}$

Now, the simplest model (shown here[271]) assumes that the exterior side of *all* compartments shares the same potential i.e. the outside of the cell is all shorted together. Hence $U_{int,i} - U_{int,j} = V_{m,i} - V_{m,j}$. That could be true in the case of an isolated neuron with highly-conductive, non-polarised surroundings. In practice, more interesting things can also happen.

In the more general case, we can consider voltages $V_{ext,ij}$ between each pair i, j of adjacent compartments. Keep in mind that these *do not* matter locally across a membrane; if the exterior of a compartment is at a certain potential then the interior side will only differ by V_m regardless. Thus the effect of external voltage differences only appears as *voltage gradients* from compartment to compartment. Mathematically, this means that from the whole cable equation, only the axial current equation has to include the change $U_{int,i} - U_{int, j} $ $= (V_{m,i} + U_{ext,i}) - (V_{m,j} + U_{ext,j}) $ $= (V_{m,i} - V_{m,j}) + (U_{ext,i} - U_{ext,j}) = V_{m,ij} + V_{ext,ij}$. Thus axial current between two adjacent compartments is the difference in their V_m divided by axial resistance, *plus* the voltage between their exteriors divided by the same.

Since they are not part of the NeuroML specification, EDEN doesn't offer a way to *directly* specify extracellular electric fields; the current assumption is that U_{ext} is the same everywhere. Nonetheless, fields can be modelled accurately by modelling the *equivalent current flows* that they cause inside the neuron, as explained above. We can do this in our NeuroML model by injecting in each compartment i, a *pseudo-current* that is the algebraic sum of all equivalent flows from this compartment to adjacent ones, caused by external voltages: $I_{eq,i} = \sum_j \frac{V_{ext,ji}}{R_{a,ij}}$. In linear algebra terms, $\mathbf{I}_{eq} = \mathbf{G}\mathbf{U}_{ext}$ where $\mathbf{G}$ is the conductance matrix between adjacent compartments. This technique is also presented in McNeal (1976)[272] and related literature.

Observe that the total pseudo-current injected to the neuron is zero. At the same time, *do note* that the real current flowing *through* a neuron because of the field is accounted for; as alterations in conductance-based trans-membrane current, that follow the applied voltage gradient from one cell to the other. For instance, a resting cell slightly polarised from side to side will have differential changes in resting potential, and thereby differential leak current.

```
def GetAxialConductanceMatrix(cell_info):
    '''Form the axial resistance matrix $I_a = G*V$ connecting compartments.

    Parameters
    ---
    cell_info : dict[str, object]
        The cell information retrieved by :py:`eden_simulator.experimental.explain_
↪cell`.
    Returns
    ---
    m : scipy.sparse.sparray
        A sparse symmetric matrix of inter-compartment conductance, in the same␣
↪units as `cell_info['comp_conductance_to_parent']`.
        TODO
    '''
    comp_ga = cell_info['comp_conductance_to_parent']
    iii = []; jjj = []; vvv = [] # Construct the lists (i,j,v)
```

(continues on next page)

[270] https://en.wikipedia.org/wiki/Cable_theory
[271] https://en.wikipedia.org/wiki/File:Cable_theory_Neuron_RC_circuit_v3.svg
[272] https://doi.org/10.1109/TBME.1976.324593

(continued from previous page)

```
    for i,p in enumerate(cell_info['comp_parent']):
        g = comp_ga[i]
        if p >= 0:
            iii += [ i, p, i, p] # add 4 sparse elms in one go
            jjj += [ p, i, i, p]
            vvv += [+g,+g,-g,-g] # same as:
            # G[i,p] += g
            # G[p,i] += g
            # G[i,i] -= g
            # G[p,p] -= g

    import scipy
    G = scipy.sparse.coo_matrix((vvv,(iii,jjj))).tocsr() # needs csr for some␣
↪reason
    return G
G_nS = GetAxialConductanceMatrix(cell_info)
```

14.4 Steady field shape

If the field varies spatially *only in magnitude* (its "pulse shape" remains the same, so to speak), we need to apply the above calculations just once for arbitrary units, to find the relative weights. Then at simulation time, we can vary the amplitude of all pseudo-current injection points by this weight factor.

Tip: Such simulations can be modelled in pure NeuroML, without any EDEN-specific extensions - albeit with the help of explain_cell (page 241) to tell us where the actual compartments are located in the neuron model and how they are electrically linked.

14.4.1 Uniform electric field

Let's say the electric field of amplitude a points to the $+y$ axis: $\overrightarrow{E} = a\,\hat{y}\ \frac{V}{m}$. Then we can define an electric potential $U(x,y,z) = -ay$ all over the place. Note that as long as we care about the *difference* i.e. *voltage*, we can assume the point where $U = 0$ wherever (in this case where $y = 0$ for simplicity).

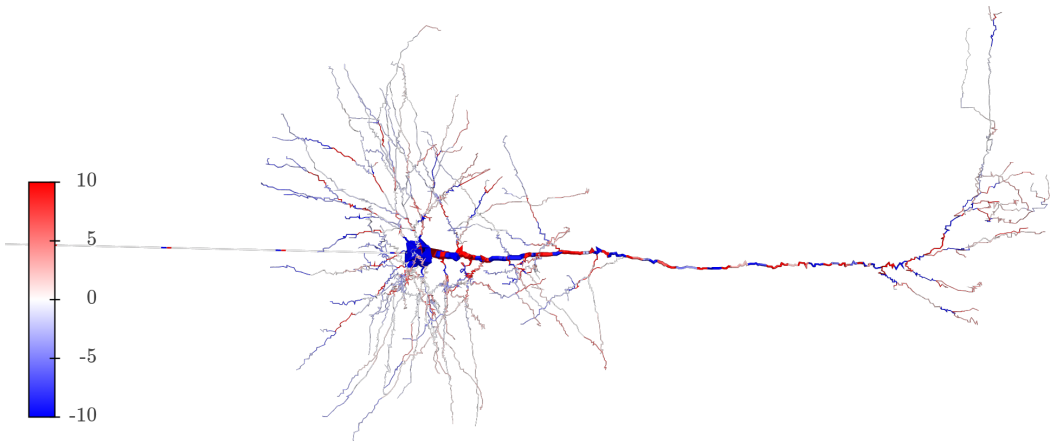
To restate: Since the mathematical expression is the same really, we will re-use the axial conductance matrix but apply it on the extracellular potential instead of membrane voltage, to calculate additional currents to those if U_{ext} was zero.
We could also derive inter-compartment voltages from the field's values. This is a bit more involved to get accurately, but some times we must (page 169).

```
def ElectricPotential_Uniform(x,y,z):
    "Get the extracellular potential for a uniform field of 1 V/m in volts for x,y,
↪z in microns."
    return -y * 1 * 1e-6 # 1 µV/µm
Vext_V = 50 * ElectricPotential_Uniform(*xyz) # let's say 50 V/m or mV/mm
Iext_nA = G_nS @ Vext_V # nS * V
```

Let's look at the pseudo-currents we're about to inject in each compartment, to apply the externally imposed field.

```
import k3d; from eden_simulator.display.spatial.k3d import Plot, plot_neuron
Im_uniform_plot = plot = Plot()
plot += plot_neuron(cell_info, Iext_nA*1e3, color_map='bwr', color_range=[-10,␣
↪+10]);
PointUp(plot); plot.show_html()
```

```
<IPython.core.display.HTML object>
```



Plot the following graph at different scales of magnitude, to appreciate the equivalent currents in both the thick and the thin compartments.

One may be surprised to see that adjacent compartments along a straight line have a non-zero injected current! *How can this be true, surely all currents add up to zero?* Well, the equation is pretty clear about it: The quantities $\frac{Vi-1-V_i}{R_{i-1,i}}$ and $\frac{V_i-V_{i+1}}{R_{i,i+1}}$ are, in any way, unequal. In fact the only place where injected current is precisely zero along the field's direction, is between electrically identical compartments on the model's stylized axon (between the nodes of Ranvier).

Well then, how exactly does it make sense that the current balance in an intermadiate compartment would be nonzero? where would it draw the current from? Remember, we are calculating a shift in axial currents here. If the shift indicates a divergence in the flow of current, this simply means that ceteris paribus the compartment has more or less of a V_m than its surroundings. Which is also what with an external point electrode, or in our case, when the neurites curve in the direction of the field and then back against it.

Let's run the simulation now:

```
model_name = 'ImposedField_Simple'; pulse_sec = [0.005,0.040]
MakeNmlModel(model_name, cell_info, Iext_nA, pulse_sec=pulse_sec)
sim_filename = MakeNmlSimFile(model_name, sim_duration_sec=0.1)
```

```
results = eden_simulator.runEden(sim_filename)
rec_time_axis_sec = results['t']; neuron_membrane_voltage = ResultsToVm(results)
```

And see what happened:

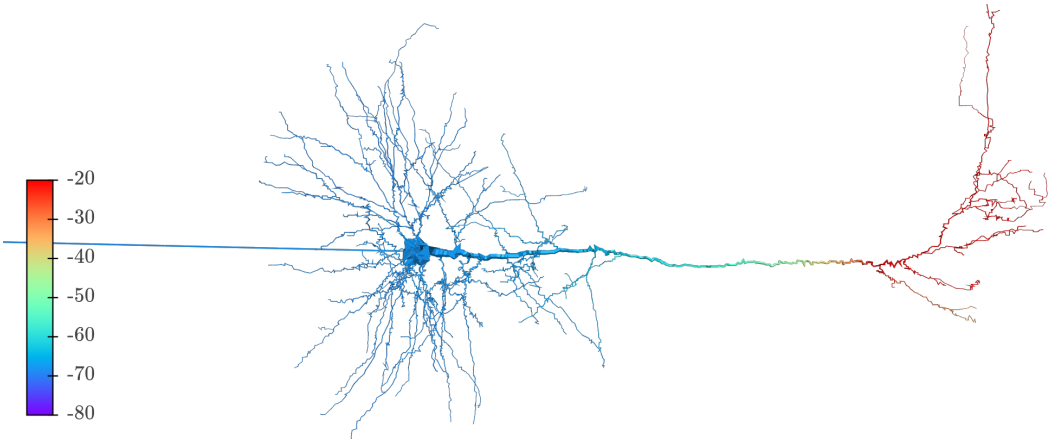
```
# Add a cute arrow to the uniform field's direction
def ArrowMesh(p0,p1,stem_radius=5,tip_radius=10,tip_length=20):
    d = np.array(p1) - p0
    import trimesh
    cone = trimesh.creation.cone(radius=tip_radius, height=tip_length,
↪transform=trimesh.geometry.align_vectors([0, 0, 1], d))
    tube = trimesh.creation.cylinder(radius=stem_radius, segment=[p0,p1])
    return  k3d.mesh(np.r_[cone.vertices+p1, tube.vertices].astype('float32'), np.
↪r_[cone.faces,len(cone.vertices)+tube.faces].astype('uint32'), color=0xffc700)
def ColorByPulse(anim_axis,sampled_time,on_off,colors):
    return { str(real_time): (colors[1] if on_off[0] < sim_time < on_off[1] else␣
↪colors[0] ) for (real_time, sim_time) in zip(anim_axis, sampled_time)  }

Vm_uniform_plot,_ = plot,(anim_axis,sampled_time) = makeplot(cell_info, neuron_
↪membrane_voltage, rec_time_axis_sec, color_range = [-80,-20])
arrow = ArrowMesh([50,-50,0], [50,+100,0])
arrow.color = ColorByPulse(anim_axis, sampled_time, pulse_sec, [0xc0c0c0,0xffc700])
# arrow.color = { f'{(5e-3)/animation_speed-.001}': 0xc0c0c0, f'{(5e-3)/animation_
↪speed}': 0xffc700, f'{(205e-3)/animation_speed-.001}': 0xffc700, f'{(205e-3)/
```

(continues on next page)

(continued from previous page)

```
↪animation_speed}': 0xc0c0c0,  }
plot += arrow
plot += k3d.text('\\overrightarrow{E}',position=[50, 120, 0], label_box=False,␣
↪reference_point='cb', size=2, color=0xffb700)
plot.show_html()
```

```
<IPython.core.display.HTML object>
```



Try the opposite field. What happens? When and where did the neuron fire first this time and why? Now try the original field but applied for 30 msec (`pulse_sec = [0.005, 0.040]`), and watch again. How did *this* happen?

Having shown how a strong enough electric field can itself make a neuron fire, in the following simulations we will use weaker sub-threshold fields to just show the spatial polarisation of the neuron.

14.4.2 Disc electrode

Now, let's simulate the effect of a flat disc electrode. The electric potential it induces follows the formula:

$$U(r,z) = \frac{2V}{\pi} sin^{-1}\left(\frac{2a}{\sqrt{(r-a)^2+z^2}+\sqrt{(r+a)^2+z^2}}\right)$$

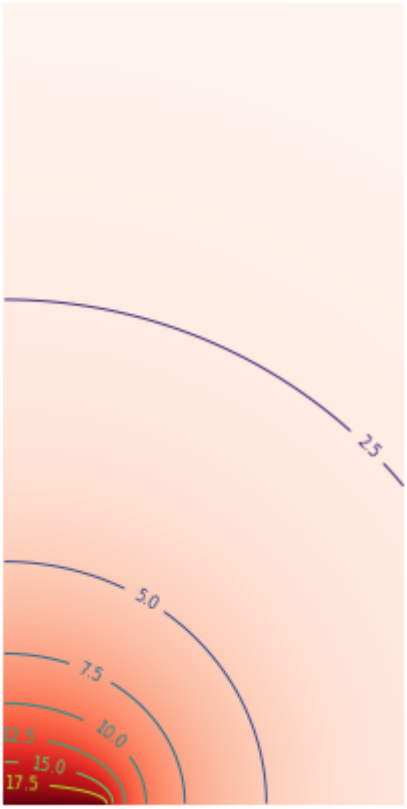
where a is the radius of the electrode and r, z are the distances parallel and perpendicular to the disc, and with $U = 0$ far far away. From Wiley and Webster (1982)[273].

```
import numpy as np
def ElectricPotential_ConductiveDisc(x,y,z, middle, radius):
    "Get the relative extracellular potential for a disc electrode on the x-z␣
↪plane field, for x,y,z in microns."
    x,y,z = x - middle[0], y - middle[1], z - middle[2]
    a = radius; norm = np.linalg.norm
    r = norm([x,z], axis=0);
    return (2/np.pi)*np.arcsin(2*a/(norm([r-a,y], axis=0) + norm([r+a,y], axis=0)))

disc_middle = np.array([-162.88156, -150, -4.9890413]); disc_radius = 50
Vext_V = 0.02 * ElectricPotential_ConductiveDisc(*xyz, disc_middle, disc_radius)
# plt.plot(Vext_V)
Iext_nA = G_nS @ Vext_V # nS * V
```

This time, we apply a *localised* field, with the electrode at a potential $+20\,mV$. It is prudent to visualise the electrode's location and computed potential, to fix mistakes at this stage before interpreting the simulation's results.

[273] https://doi.org/10.1109/TBME.1982.324910

```
from matplotlib import pyplot as plt
image_size_microns = [200,400]; imgsiz = image_size_microns
X,Y = np.meshgrid(np.arange(0,imgsiz[0])+disc_middle[0],np.arange(0,
↪imgsiz[1])+disc_middle[1]); Z = 0*X+disc_middle[2]
sampUext = 0.02*ElectricPotential_ConductiveDisc(X,Y,Z,disc_middle, disc_radius)
dpi = 144
fig,ax = plt.subplots(figsize=[2*imgsiz[0]/dpi,2*imgsiz[1]/dpi],dpi=dpi)
ax.pcolormesh(X,Y,sampUext,cmap='Reds')
CS = ax.contour(X, Y, sampUext*1e3,linewidths=1); ax.clabel(CS, fontsize=8)
ax.set_axis_off();ax.axis('equal')
fig.subplots_adjust(top=1.0, bottom=0, right=1.0, left=0, hspace=0, wspace=0)
fig.savefig('uext_disc.png', bbox_inches='tight', pad_inches=0)
fig.dpi=fig.dpi/2 # just for publishing this big vertical image...
```



To illustrate the effect, the proximity and tininess of the electrode have been exaggerated. Please also disregard for a moment that the artificial axon goes through the electrode, it won't affect the result.

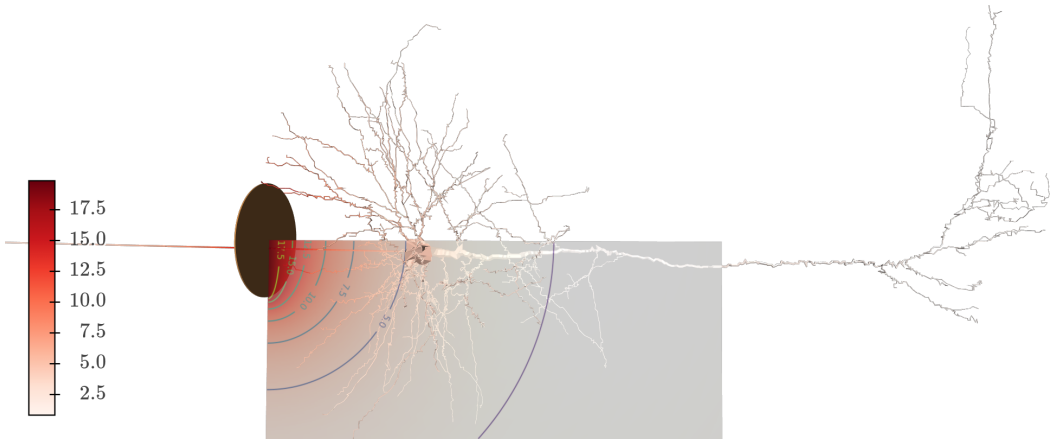
```
def AddBillboardXY(filename,off,ext,name=None):
    return k3d.mesh(
        vertices=[off+[    0,      0,0],off+[ext[0],     0,0],
                  off+[ext[0],ext[1],0],off+[     0,ext[1],0]],
        indices=[[0,1,2,],[2,3,0]],uvs=[[0,1],[1,1],[1,0],[0,0]],
        side='double',
        texture=open(filename,'rb').read(),
        texture_file_format=filename.split('.')[-1],
        color=0xffffff,name=name)
def DiscMesh():
    import pyvista as pv
    import logging; k3d.helpers.logger.setLevel(logging.WARN) # silence whine for␣
↪float64 numbers
    mesh = pv.Cylinder(center=disc_middle, direction=[0, 1, 0], radius=disc_radius,
↪ height=1);
    return k3d.vtk_poly_data(mesh, color=0xc6884b);
```

(continues on next page)

(continued from previous page)

```
Uext_disc_plot = plot = Plot()
plot += plot_neuron(cell_info, Vext_V*1e3,color_range=None, color_map='reds');
# Add a graph of the field
graph = AddBillboardXY('uext_disc.png',disc_middle,imgsiz,name='Field graph')
graph.opacity = 0.75; plot += graph
plot += DiscMesh() # Add the electrode
PointUp(plot); plot.show_html()
```

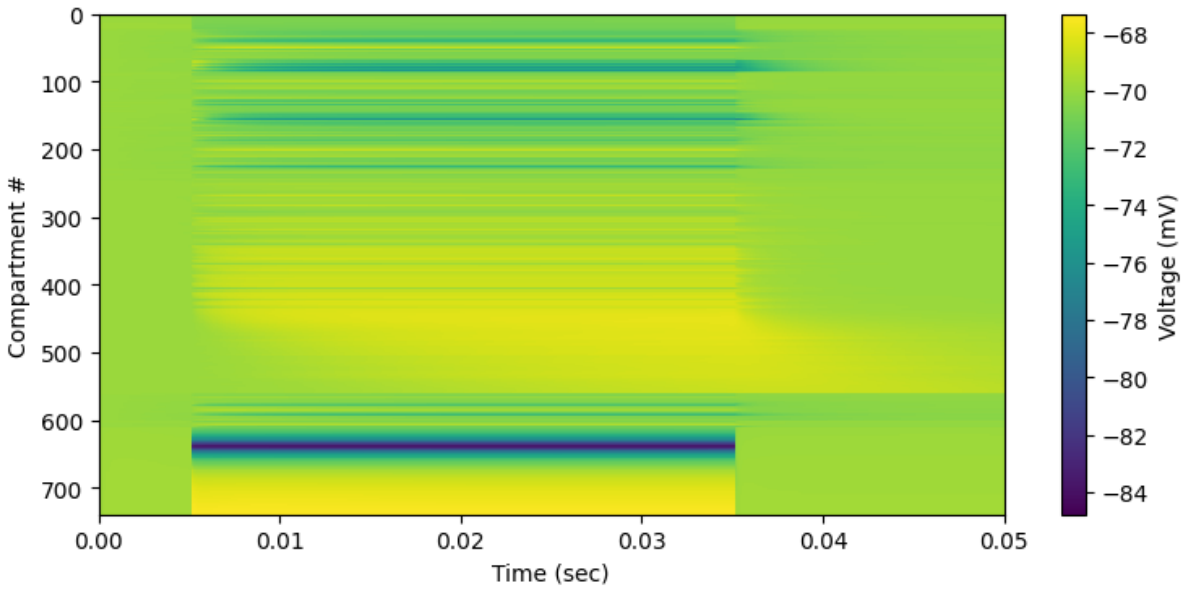
```
<IPython.core.display.HTML object>
```



Looks good, let's run the simulation then.

```
model_name = 'ImposedField_Disc'; pulse_sec = [0.005,0.035]
MakeNmlModel(model_name, cell_info, Iext_nA, pulse_sec=pulse_sec)
sim_filename = MakeNmlSimFile(model_name, sim_duration_sec = 0.050)
```

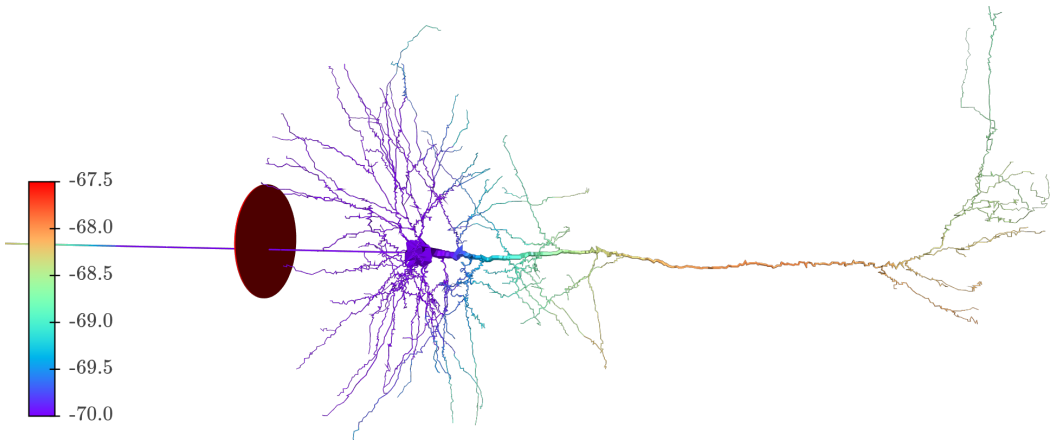
```
results = eden_simulator.runEden(sim_filename)
rec_time_axis_sec = results['t']; neuron_membrane_voltage = ResultsToVm(results)
plt.figure(figsize=[9,4]);ShowVmRaster(neuron_membrane_voltage); #plt.clim([-70, -
↪67.5])
```



The cell has been slightly polarised but not past rheobase. Note that the shift in charges completed in some 16 msec, but even faster for most parts. Let's see how that looks in 3-D:

```
Vm_disc_plot,_ = plot,(anim_axis,sampled_time) = makeplot(cell_info, neuron_
↪membrane_voltage, rec_time_axis_sec, color_range = [-70,-67.5])
plate = DiscMesh()
plate.color = ColorByPulse(anim_axis,sampled_time,pulse_sec,[0xc6884b,0xff0000])
plot += plate
plot.show_html()
```

```
<IPython.core.display.HTML object>
```



14.5 Magnetic field by a coil

14.5.1 More physics

Note: The following analysis is called *quasi-static* since it neglects EM wave dynamics. If radio-frequency effects are is significant, well … perhaps a timestep in the order of µsec is too big, and this level of modelling too coarse to be honest. Or the effect could be lumped at this level, with an empirical formula.

When a magnetic field changes, it induces[274] a *curl* on the electrical field: $\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$. The electric potential is then ill defined, because one could go from point A to point B following however many loops and gain an arbitrary amount of energy. *However!* The potential through a conductor (neural cable) is still well defined, and so is the the induced *electromotive force* i.e. the *line integral* of the field, from compartment A to compartment B. Which we can calculate just fine, we'll just have to (in theory) trace the exact path taken. If the compartments are chopped finely enough and the field is smooth enough, we can sample the field at the midpoints x_i of the compartments only, and approximate $V_{ij} \approx -\Delta \vec{x}_{ij} \cdot \frac{1}{2}(\vec{E}(x_i) + \vec{E}(x_j))$.

An example about it will be provided on popular demand. The interested reader is kindly requested to contact us (page 3) *with an use case to make us go faster.*

14.6 General time-varying field

In the more general case where the shape of the field does change over time, the virtual currents also need to be re-sampled along the time axis as well; hence, we need to feed each pseudo-source with an arbitrary waveform. EDEN can easily handle this through its NeuroML extension for input time series (page 116).

[274] https://en.wikipedia.org/wiki/Maxwell's_equations

14.6.1 Magnetic field by a moving coil

An example about it will be provided on popular demand. The interested reader is kindly requested to contact us (page 3) *with an use case to make us go faster.*

14.7 Further considerations

- Since cells are laid out as simple trees rather than coils, we don't expect to see too drastic effects over just one neuron; consider the coupling between neurons at the tissue scale. In this case, besides the electric synapses obviously connecting adjacent cells, modellers have to consider more indirect (but at scale, significant) paths to electric coupling.
- When simulating multiple neurons, or when the field's coordinates are different from the neuron's origin: Remember to translate/rotate by the neuron's pose[275]!

14.7.1 Feature size of the neurons and the fields

An important property of the field is its "feature size", or the smallest distance over which it may change significantly. For a good approximation of the field, sampling at points must be at least as fine-detailed as the field is in the vicinity.

- If the (radius of) curvature of the field is tighter than the distance between compartments, don't just sample the midpoints; consider the whole path through the morphology's `<segment>`s, with a tool like LFPykit. On the other hand, in this case the compartments should be fine enough that local effects are fully captured. Which brings us to …
- If the morphology's segments are too coarse … well, get a better morphology, or somehow simulate a more "likely" one. For instance, if some segments are stylised and unrealistically long, subdivide them (page 52) and add some realistic curvature as applicable.
- Be especially wary of the neuron's *axons*: modellers often simplify them to huge straight sticks, which then catch a disproportionate lot of induced field and hence apply a very large and very *wrong* influence on the cell's behaviour. As a great mind once put it, if something is a fiction and it drives the cell, then the cell is driven by a fiction.

14.7.2 Detail of the surroundings

- In the above we assumed a linear, uniform, isotropic, totally dull extracellular environment. But the real tissue may have anatomical features which constrain electrical flow, the dura[276] just to name one. Be aware and creative while negotiating brain effects, and contact us (page 3) if you have questions. If the effect is really local, it might make sense to constrain simulation to that point and crank up the resolution (see also other finite-element and molecule-based programs).

14.7.3 Ephaptic coupling

Sometimes instead of a blunt and unnaturally powerful electrode, we want to model the interactions between cells (with less power to give and more 'internal resistance' so to speak). In the finer-detailed case, there are lots of compartment-compartment interactions: to maintain numerical stability, the whole tissue gets coupled into a huge linearised system that we'd have to solve on each timestep(!). Simulators like ELFENN[277] try to take care of this but can handle a limited number of compartments; we are also developing an EDEN-beased solution even for this tricky subject, stay tuned.

[275] `https://en.wikipedia.org/wiki/Pose_(computer_vision)`
[276] `https://en.wikipedia.org/wiki/Dura_mater`
[277] `https://doi.org/10.3389/fninf.2019.00035`

There are other cases though, such as within certain axon bundles, where we can see the potentials flowing in a synchronised wave; we can then assume this stampede immovable by any one neuron, and study the dynamics of that one neuron as forced by the *imposed* neural wave around it. As seen in this article.

14.8 Related resources

Many relevant discussions and pitfalls to watch out for can be found on the NEURON forum[278]. Search for "extracellular", "field" or so.

- For NEURON users: This is the `stim` part of the popular extracellular_stim_and_rec[279] scripts.

[278] https://neuron.yale.edu/phpBB
[279] https://web.archive.org/web/20100730180342/http://neuron.yale.edu/phpBB/viewtopic.php?f=28&t=168

CHAPTER

FIFTEEN

EXAMPLE: PLAYING PONG

A useful way to study behaviour is simple small "toy" environments. Within these environments, if an actor is not yet capable of behaving, it may *learn* to behave through interaction and some gentle prodding to the right direction. This process is called reinforcement learning[280].

In this article, we'll see how a simple table-top game can be modelled within EDEN, to be used in closed-loop interaction experiments.

The game has a rectangular two-dimensional field, and a ball (or puck) bouncing around it. There are two "player" entities that can move horizontally in the field and try to deflect the ball by tracking the paddle.

The actors provided by this example are not neural networks but hard-wired linear control law. Arguably, this is good enough for this purpose, so adding neural networks is not that impressive as an AI task; nonetheless, it may be useful to study neurons.

15.1 Modelling the play environment

15.1.1 The field

As a form of "tennis" in the wide sense, the playing field is a rectangle of a certain *width* and *height*. To simplify math, we'll be using the *half* width (`hw`) and height(`hh`) as measures, and take (0,0) as the coordinates of the field's middle. This is where we'll serve the balls from as well. (A variant is serving from a paddle, which would give more time to the players to react.)

A ball of a certain *radius* will be bouncing around this field. If it touches ±`hw` it is reflected, if it touches ±`hh` one of the player scores and the game continues with a serve.

15.1.2 The paddles

The paddles are also shaped as rectangles, which are moved around on *horizontal* rails, by a *control force* that a *player* suggests for each of them.
Since the players are not part of the playing field, they interact with the display controls through `<VariableReference>`s, as we'll see in the following.

So then, the constants for all the above are:

```
import numpy as np
field_hw = 1.0 # half width
field_hh = 1.5 # half height

paddle_hw = 0.20 # likewise for paddle
paddle_hh = 0.05
```

(continues on next page)

[280] https://en.wikipedia.org/wiki/Reinforcement_learning

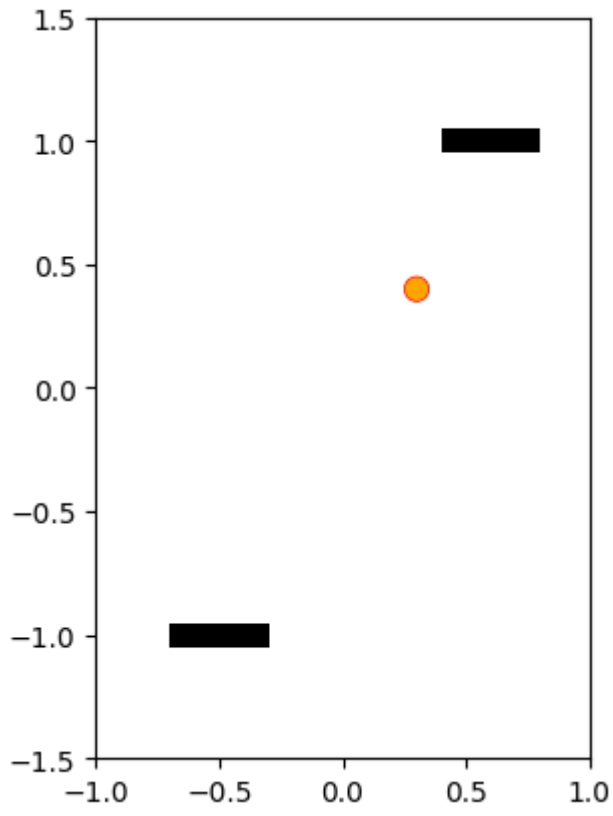
(continued from previous page)

```
paddle_y = 1 # distance from middle at whose +- the paddles lie

ball_r = 0.05 # ball radius
```

15.2 Showing the playing field

Here's how the field will look like, following the above.

```
import matplotlib.pyplot as plt
def draw_field(ax, px, py, p1x, p2x, s1=None, s2=None, time=None):
    ax.set_xlim(-field_hw, +field_hw)
    ax.set_ylim(-field_hh, +field_hh)
    ax.set_aspect('equal') # don't distort the playing field
    from matplotlib.patches import Rectangle, Circle
    ax.add_patch(Rectangle((p1x-paddle_hw,-paddle_y-paddle_hh), paddle_hw*2,␣
↪paddle_hh*2, linewidth=0, color='k'))
    ax.add_patch(Rectangle((p2x-paddle_hw,+paddle_y-paddle_hh), paddle_hw*2,␣
↪paddle_hh*2, linewidth=0, color='k'))
    ax.add_patch(Circle((px, py), ball_r, ec="red", facecolor='orange', linewidth=.
↪5))
    # ax.axhline(-paddle_y); ax.axhline(paddle_y) to taste
    if time is not None: ax.set_xlabel(f'Time: {time:.3f} sec')
    if s1 is not None: ax.text(0.5, 0, f'Score: {s1:.0f}', ha='center', va='bottom
↪', transform=ax.transAxes)
    if s2 is not None: ax.text(0.5, 1, f'Score: {s2:.0f}', ha='center', va='top',␣
↪transform=ax.transAxes)

# Show an example
draw_field(plt.gca(), 0.3, 0.4, -0.5, 0.6)
```

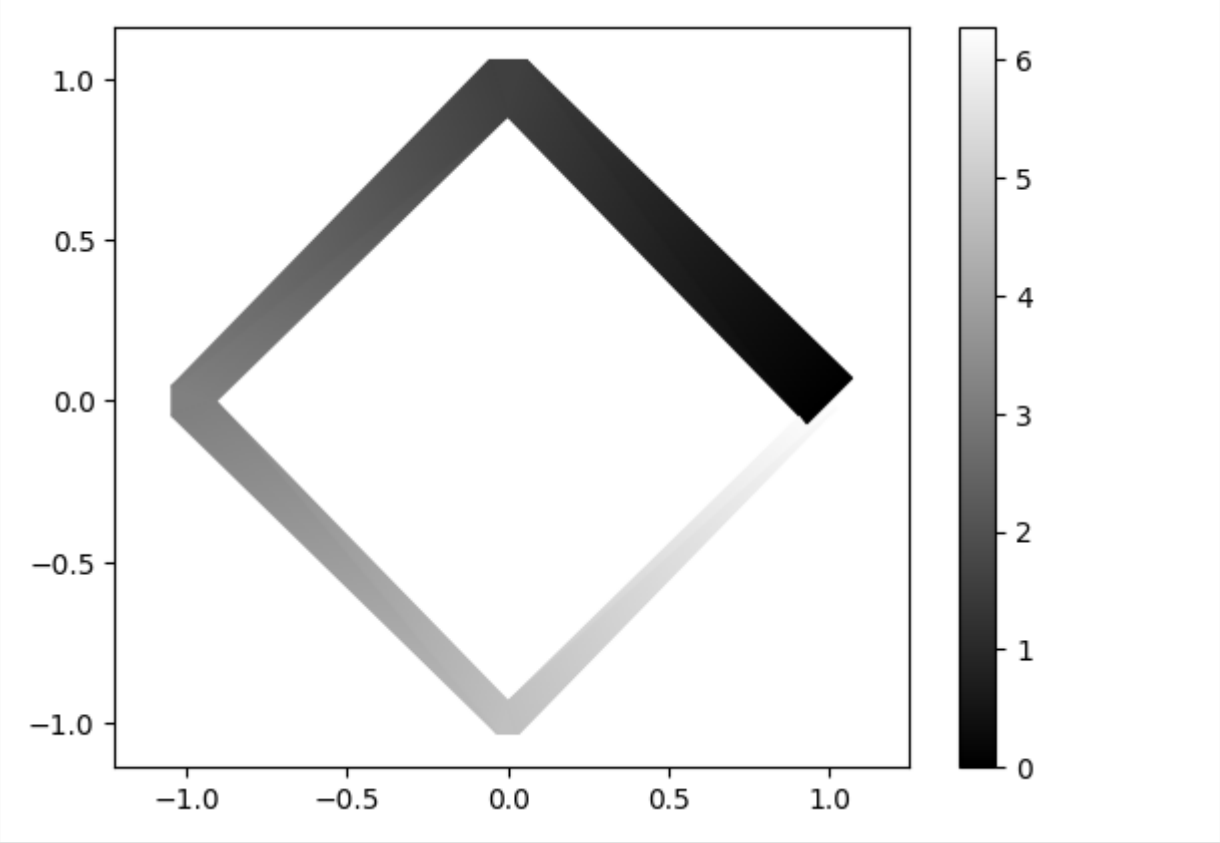


15.2.1 Showing the ball's trace through the field

It is easier to perceive the fast moving ball if we add the trail of its previous recent positions, with fading colour and width going back to time.

However, all easily accessible graphics tools can't handle this concept without nasty artifacts; we'll have to construct the ribbon from polygonal vertices and edges instead. The algorithm goes as follows:

```
N = 4 # segments
t = np.linspace(0,2*np.pi,N+1)
xy = np.vstack([np.cos(t), np.sin(t)]).T
wmax = .1; wmin = .03
width = (wmax-wmin)*(1 - t/(2*np.pi)) + wmin
def intersect_2d(p00, p01, p10, p11):
    '''Get the lerp factors to intersect 2d segs'''
    den= ((p00[0]-p01[0])*(p10[1]-p11[1]) - (p00[1]-p01[1])*(p10[0]-p11[0]))
    if np.any(np.abs(den) < 1e-9): return 1,0 # and hope for the best
    f0 = ((p00[0]-p10[0])*(p10[1]-p11[1]) - (p00[1]-p10[1])*(p10[0]-p11[0])) / den
    f1 = ((p00[0]-p10[0])*(p00[1]-p01[1]) - (p00[1]-p10[1])*(p00[0]-p01[0])) / den
    return f0,f1
def fancy_streamline(ax, xy, width, value, **kwargs):
    '''Draw a thick ribbon of variable width and value.  By Sotirios Panagiotou'''
    N = len(width)
    if N < 2: return None # can't draw a line like this
    def segify(xy):
        ''' Duplicate the elements except for the first and the last.'''
        return xy.repeat(2, axis=0)[1:-1]

    d = np.diff(xy, axis=0)
    n = d / (np.linalg.norm(d,axis=-1)[:,None] + 1e-20) # normals
    ww = np.vstack([width[:-1], width[1:]]).T.flatten() # width replicated
    tt = np.vstack([value[:-1], value[1:]]).T.flatten() # value replicated
    nn = (n.repeat(2, axis=0)*ww[:,None])[:,::-1]*[-1, 1] # binormals
    xyxy = segify(xy) # points replicated for start+end
    pp = np.vstack([(xyxy+nn),(xyxy-nn)])

    #replace inner-side points with intersection point of segs
    for i in range(0,len(pp)-3,2):
        if i + 2 == len(pp)/2: continue
        f0, f1 =  intersect_2d(pp[i], pp[i + 1], pp[i + 2], pp[i + 3])
        if 0 <= f0 <= 1 and 0 <= f1 <= 1 :
            pp[i+1] , pp[i+2] = 2*[(1-f0)*pp[i]+f0*pp[i+1]]
    # ax.plot(*xy.T); ax.plot(*pp[0:len(pp)//2].T); ax.plot(*pp[len(pp)//2:].T)
    # Gouraud shading will still show mach bands (why?) along the strip (try with␣
↪hsv), oh well.
    tris = [ [xyxy.shape[0]+i+1, i, i+1, ] for i in range((N-1)*2-1)]
    tris+= [ [xyxy.shape[0]+i, xyxy.shape[0]+i+1, i, ] for i in range((N-1)*2-1)]
    im = ax.tripcolor(pp[:,0], pp[:,1],  np.hstack([tt,tt]), triangles=tris,␣
↪shading="gouraud", **kwargs)
    return im

im = fancy_streamline(plt.gca(), xy, width, t, cmap='gray', edgecolor='#0f0f0f00')
plt.axis('equal');plt.colorbar(im);#plt.show()
```



15.3 Game Dynamics

It wouldn't be fun if things didn't change, here is how things change.

15.3.1 Walls

Whenever the ball hits one of the horizontal walls, its velocity (and position) is reflected along the wall.

15.3.2 Paddle movement

Each paddle receives an input force from a player (with a game-specific cap for the maximum amplitude) which leads to accelerating toward one direction or the other. The paddles are, constrained between the field's two horizontal walls, and each's movement speed is set to zero whenever it hits a wall.

15.3.3 Paddle hits

When the ball hits a paddle, its vertical speed is reflected to the opposite direction. Its horizontal speed gets more interesting in that case, depending on which part of the paddle it bounced on: it becomes a fraction of what it was the closer it gets to the middle, but at the same time is incremented by a fraction of total speed the further away it gets from the paddle's center.

15.3.4 Serving

When the ball reaches the vertical end of the field behind one of the paddles, a point is awarded to the opposite side's player and the ball is put back in the middle of the playing field. The ball stays still for a small period, after which it is launched toward one or the other paddle at a random direction and play resumes.

15.4 Implementing the playing field

Let's cast all those play rules into a LEMS description with the various `<Parameter>`s and `<StateVariable>`s:

```
%%writefile PongField.nml
<neuroml>
<!-- Let's make a new ComponentType ! it's a big one because it runs a whole pong-
↪tennis field. -->
<ComponentType name="PongField">
    <Constant name="sec" dimension="time" value="1 sec"/>
    <Constant name="per_sec" dimension="per_time" value="1 per_s"/> <!-- LATER␣
↪meters per second -->
    <Parameter name="pi" dimension="none" value="3.14159"/> <!-- Add the␣
↪dimensionless constant π -->

    <Parameter name="hw" dimension="none" description="half-width"/>
    <Parameter name="hl" dimension="none" description="Standard deviation of␣
↪current"/>

    <Property name="ball_r"   dimension="none" defaultValue="0.05" description=
↪"Radius of the ball"/>
    <Property name="max_ball_speed" dimension="per_time" defaultValue="100 per_s"␣
↪description="Don't let the ball run too fast, or it might clip through the paddle
↪"/>
    <Property name="min_ball_vertical_speed" dimension="per_time" defaultValue="1␣
↪per_s" description="Don't let the ball run too slow, or it might take ages to␣
↪bounce through the field"/>

    <Property name="paddle_y" dimension="none" defaultValue="1" description=
↪"Location of the paddle"/>
    <Property name="paddle_hw" dimension="none" defaultValue="0.20" description=
↪"Width  of the paddle"/>
    <Property name="paddle_hh" dimension="none" defaultValue="0.05" description=
↪"Height of the paddle"/>

    <Property name="paddle_h_gain_factor"   dimension="none" defaultValue="0.40"␣
↪description="Add some of total speed to H-speed when bouncing on the paddle's␣
↪sides."/>
    <Property name="paddle_h_loss_factor"   dimension="none" defaultValue="0.5"␣
↪description="Deciamte H-speed then bouncing on the paddle's middle."/>
    <Property name="max_paddle_force_per_s2" dimension="none" defaultValue="600"␣
↪description="Maximum possible paddle acceleration"/>

    <Property name="serve_speed" dimension="per_time" defaultValue="4 per_s"␣
↪description="Speed to serve at"/>
    <Property name="serve_duration" dimension="time" defaultValue="100 ms"␣
↪description="Time betwen point and serve"/>

    <VariableRequirement name="paddle_1_a_per_s2" dimension="none" description=
↪"Requested paddle force from player 1."/>
    <VariableRequirement name="paddle_2_a_per_s2" dimension="none" description=
↪"Requested paddle force from player 2."/>
```

(continues on next page)

(continued from previous page)

```
    <Dynamics>
        <StateVariable name="px" exposure="bx" dimension="none"  description="H-
↪Location of ball"/>
        <StateVariable name="py" exposure="by" dimension="none"  description="V-
↪Location of ball"/>
        <StateVariable name="vx" exposure="vx" dimension="per_time" description="H-
↪speed of ball"/>
        <StateVariable name="vy" exposure="vy" dimension="per_time" description="V-
↪speed of ball"/>
        <StateVariable name="paddle_1_x" dimension="none"  description="H-Location␣
↪of paddle 1 (bottom)"/>
        <StateVariable name="paddle_2_x" dimension="none"  description="H-Location␣
↪of paddle 2 (top)   "/>
        <StateVariable name="paddle_1_v" dimension="per_time" description="H-Speed␣
↪of paddle 1 (bottom)"/>
        <StateVariable name="paddle_2_v" dimension="per_time" description="H-Speed␣
↪of paddle 2 (top)   "/>

        <StateVariable name="score_1" exposure="score_1" dimension="none"␣
↪description="Score for player 1 (bottom)"/>
        <StateVariable name="score_2" exposure="score_2" dimension="none"␣
↪description="Score for player 2 (top)   "/>
        <StateVariable name="serve_clock" exposure="serve_clock" dimension="time"␣
↪description="Timer for serve (negative to disable)"/>
        <!--
        <DerivedVariable name="paddle_1_a_per_s2" dimension="none" value="40*(px-
↪paddle_1_x) + 900*(2*random(1)-1)/2" description="Requested control force for␣
↪player 1"/>
        <DerivedVariable name="paddle_2_a_per_s2" dimension="none" value="30*(px-
↪paddle_2_x) + 900*(2*random(1)-1)/2" description="Requested control force for␣
↪player 2"/>-->
        <!-- Limit paddle force to what's "feasible", LATER dimensional abs␣
↪function? -->
        <ConditionalDerivedVariable name="paddle_1_aeff_per_s2" dimension="none"␣
↪description="Effective control force for player 1">
            <Case condition="paddle_1_a_per_s2 .gt. +max_paddle_force_per_s2"␣
↪value="+max_paddle_force_per_s2"/>
            <Case condition="paddle_1_a_per_s2 .lt. -max_paddle_force_per_s2"␣
↪value="-max_paddle_force_per_s2"/>
            <Case value="paddle_1_a_per_s2"/>
        </ConditionalDerivedVariable>
        <ConditionalDerivedVariable name="paddle_2_aeff_per_s2" dimension="none"␣
↪description="Effective control force for player 2">
            <Case condition="paddle_2_a_per_s2 .gt. +max_paddle_force_per_s2"␣
↪value="+max_paddle_force_per_s2"/>
            <Case condition="paddle_2_a_per_s2 .lt. -max_paddle_force_per_s2"␣
↪value="-max_paddle_force_per_s2"/>
            <Case value="paddle_2_a_per_s2"/>
        </ConditionalDerivedVariable>

        <!-- Keep track of 2-D speed -->
        <DerivedVariable name="ball_speed" dimension="per_time" value="sqrt((vx/
↪per_sec)^2+(vy/per_sec)^2)*per_sec" description="Magnitude of speed vector"/>

        <!-- And sundry -->
        <DerivedVariable name="random_phase" dimension="none" value="random(2*pi)"␣
↪description="Have a named random variable to use the same value in two places"/>
        <ConditionalDerivedVariable name="serve_clock_rate" dimension="per_time"␣
↪description="Advance only when it's not elapsed">
            <Case condition="serve_clock >= 0*sec" value="1 * per_sec"/>
            <Case value="0"/>
```

(continues on next page)

(continued from previous page)

```
        </ConditionalDerivedVariable>

        <!-- Move the ball and paddles around, and occasionally count time -->
        <TimeDerivative variable="px" value="vx"/> <TimeDerivative variable="py"␣
↪value="vy"/>
        <TimeDerivative variable="paddle_1_x" value="paddle_1_v"/> <TimeDerivative␣
↪variable="paddle_1_v" value="paddle_1_aeff_per_s2 * per_sec * per_sec"/>
        <TimeDerivative variable="paddle_2_x" value="paddle_2_v"/> <TimeDerivative␣
↪variable="paddle_2_v" value="paddle_2_aeff_per_s2 * per_sec * per_sec"/>
        <TimeDerivative variable="serve_clock" value="serve_clock_rate"/>

        <!-- bounce on walls -->
        <OnCondition test="px + ball_r .gt. hw">
            <StateAssignment variable="px" value="2*(hw - ball_r) - px"/>
            <StateAssignment variable="vx" value="-vx"/>
        </OnCondition>
        <OnCondition test="px .lt. -hw"> <!-- note: greater-than is allowed in ␣
↪https://www.w3.org/TR/xml/#syntax -->
            <StateAssignment variable="px" value="-hw - (px + hw)"/>
            <StateAssignment variable="vx" value="-vx"/>
        </OnCondition>

        <!-- score on non-walls -->
        <OnCondition test="py > hl">
            <StateAssignment variable="score_1" value="score_1 + 1"/>
            <StateAssignment variable="serve_clock" value="0*sec"/>
            <StateAssignment variable="px" value="0"/> <StateAssignment variable=
↪"py" value="0"/>
            <StateAssignment variable="vx" value="0"/> <StateAssignment variable=
↪"vy" value="0"/>
        </OnCondition>
        <OnCondition test="-hl > py">
            <StateAssignment variable="score_2" value="score_2 + 1"/>
            <StateAssignment variable="serve_clock" value="0*sec"/>
            <StateAssignment variable="px" value="0"/> <StateAssignment variable=
↪"py" value="0"/>
            <StateAssignment variable="vx" value="0"/> <StateAssignment variable=
↪"vy" value="0"/>
        </OnCondition>

        <!-- bounce on paddles. Could also add up some of the paddle's speed -->
        <!-- fix the hitbox to be perfectly circular if you want -->
        <OnCondition test="( (paddle_hw + ball_r) > abs( px - paddle_1_x)  ) .and.␣
↪( (paddle_hh + ball_r) > abs(py + paddle_y) )">
            <StateAssignment variable="vy" value="-vy"/>
            <StateAssignment variable="py" value="-(paddle_y - ball_r - paddle_hh)
↪"/>
            <!-- funny math: give a way to manipulate the ball's direction based␣
↪on distance form the center of the paddle -->
            <StateAssignment variable="vx" value="vx*((1-paddle_h_loss_factor)+␣
↪(paddle_h_loss_factor)*abs((px - paddle_1_x) / paddle_hw)) + (ball_speed *␣
↪paddle_h_gain_factor * (px - paddle_1_x) / paddle_hw)"/>
        </OnCondition>
        <OnCondition test="( (paddle_hw + ball_r) > abs( px - paddle_2_x)  ) .and.␣
↪( (paddle_hh + ball_r) > abs(py - paddle_y) )">
            <StateAssignment variable="vy" value="-vy"/>
            <StateAssignment variable="py" value="+(paddle_y - ball_r - paddle_hh)
↪"/>
            <!-- funny math: give a way to manipulate the ball's direction based␣
↪on distance form the center of the paddle -->
            <StateAssignment variable="vx" value="vx*((1-paddle_h_loss_factor)+␣
```

(continues on next page)

(continued from previous page)

```
↪(paddle_h_loss_factor)*abs((px - paddle_2_x) / paddle_hw)) + (ball_speed *␣
↪paddle_h_gain_factor * (px - paddle_2_x) / paddle_hw)"/>
        </OnCondition>

        <!-- constrain paddles -->
        <OnCondition test="(paddle_hw + paddle_1_x) > hw">
            <StateAssignment variable="paddle_1_v" value="0*paddle_1_v"/>
            <StateAssignment variable="paddle_1_x" value="hw - paddle_hw"/> <!--␣
↪or to taste, eg reflect -->
        </OnCondition>
        <OnCondition test="-hw > (-paddle_hw + paddle_1_x)">
            <StateAssignment variable="paddle_1_v" value="0*paddle_1_v"/>
            <StateAssignment variable="paddle_1_x" value="-hw + paddle_hw"/> <!--␣
↪or to taste, eg reflect -->
        </OnCondition>
        <OnCondition test="(paddle_hw + paddle_2_x) > hw">
            <StateAssignment variable="paddle_2_v" value="0*paddle_2_v"/>
            <StateAssignment variable="paddle_2_x" value="hw - paddle_hw"/> <!--␣
↪or to taste, eg reflect -->
        </OnCondition>
        <OnCondition test="-hw > (-paddle_hw + paddle_2_x)">
            <StateAssignment variable="paddle_2_v" value="0*paddle_2_v"/>
            <StateAssignment variable="paddle_2_x" value="-hw + paddle_hw"/> <!--␣
↪or to taste, eg reflect -->
        </OnCondition>

        <!--  serve mechanics -->
        <OnCondition test="serve_clock > serve_duration">
            <StateAssignment variable="vx" value="serve_speed * cos(random_phase)"/
↪>
            <StateAssignment variable="vy" value="serve_speed * (sin(random_phase)␣
↪+ 0.4*(2*H(sin(random_phase))-1))"/>
            <StateAssignment variable="serve_clock" value="-1 * sec"/> <!--␣
↪disable -->
        </OnCondition>
        <OnCondition test="ball_speed > max_ball_speed">
            <StateAssignment variable="vx" value="vx * max_ball_speed/ball_speed"/>
            <StateAssignment variable="vy" value="vy * max_ball_speed/ball_speed"/>
        </OnCondition>
        <OnCondition test="(abs(vy/per_sec) .lt. min_ball_vertical_speed/per_sec) .
↪and. (serve_clock .lt. 0)">
            <StateAssignment variable="vy" value="min_ball_vertical_speed *␣
↪(2*H(vy/per_sec)-1)"/> <!-- a sign function would look nicer, if it was in lems -
↪->
        </OnCondition>

        <!--  and set initial conditions -->
        <OnStart>
            <StateAssignment variable="px" value="+0.7"/>
            <StateAssignment variable="py" value="-0.5"/>
            <StateAssignment variable="vx" value="3*per_sec"/> <!-- or to random5 -
↪->
            <StateAssignment variable="vy" value="5*per_sec"/> <!-- or to random3 -
↪->
            <StateAssignment variable="paddle_1_x" value="0"/> <StateAssignment␣
↪variable="paddle_1_v" value="0 * per_sec"/>
            <StateAssignment variable="paddle_2_x" value="-0.5"/> <StateAssignment␣
↪variable="paddle_2_v" value="0 * per_sec"/>
            <StateAssignment variable="serve_clock" value="-1 * sec"/> <!-- serve␣
↪right away -->
        </OnStart>
```

(continues on next page)

(continued from previous page)

```
        <!-- done! -->
    </Dynamics>
</ComponentType>
</neuroml>
```

```
Writing PongField.nml
```

15.5 Implementing the players

For illustration purposes (and since setting a SNN to competently control the ball appears *tricky* even when any living creature could manage), we'll implement the players as *proportional control* governors which chase after where the ball is now. Keep in mind that the effective command is capped in the playing field, to simulate real-life constraints. And because that would be too perfect, we'll introduce *noise* (simulating "tremor" and general uncertainty) that perturbates the control enough to make it interesting.

We could have implemented all this inside one LEMS component, but were we'll break it down since we want our own non-trivial system (a *spiking neural network* perhaps?) to work the paddles. The playing field (or *environment*) is separated from the *controller*, with the only communication flows being observation (in whichever form) and control command (in whichever form).

Here is then, the *proportional control* rule expressed in LEMS. Note that the same `<ComponentType>` can be used for both players.

Note: Even though the component is static (the paddle's command force is simply where it isn't minus where it is[281]), we continuously assign a `StateVariable` with an always-holding `<OnCondition>` so that the `<VariableRequirement>` can observe it.

```
%%writefile PongPlayer.nml
<neuroml>
<!-- Let's make a new ComponentType ! for the player. -->
<ComponentType name="PongPlayer">

    <Property name="force_per_deviation" dimension="none" defaultValue="100"␣
↪description="Weight of (ball to paddle center) on force command"/>
    <Property name="noise_intensity"     dimension="none" defaultValue="900"␣
↪description="Weight of noise on force command"/>

    <VariableRequirement name="paddle_x" dimension="none" description="The␣
↪horizontal coordinate of the paddle (where the paddle is)."/>
    <VariableRequirement name="ball_x"   dimension="none" description="The␣
↪horizontal coordinate of the ball (where the paddle isn't), to blidly track."/>
    <Dynamics>
        <!-- NOTE: should be a derived variable, but eden doesn't allow␣
↪VariableReferences to non-state variables yet. LATER make that happen. -->
        <StateVariable name="request_a_per_s2" exposure="request_a_per_s2"␣
↪dimension="none"  description="H-Location of ball"/>
        <OnCondition test="2 + 2 == 4">
            <!-- NOTE: the effect of this formula is sensitive to timestep ! See␣
↪the Ornstein-Uhlenbeck noise example for a dt-corrected random source. -->
            <StateAssignment variable="request_a_per_s2" value="force_per_
↪deviation*(ball_x-paddle_x) + noise_intensity*(2*random(1)-1)/2"/>
        </OnCondition>
    </Dynamics>
```

(continues on next page)

[281] https://web.archive.org/web/20030813200740/http://www.afmissileers.org/AFMissileers/pdf/NL1997/Dec97.pdf#page=5

(continued from previous page)

```
</ComponentType>
</neuroml>
```

```
Writing PongPlayer.nml
```

15.6 Putting it all together

The next step is to add the playing field and the players in a simulation. Since there is no hard <Requirement> or dependency on other entities, for "cells" we'll just instantiate 1 field and 2 players as <population>s. A creative modeller could also have formulated them as synapses or input sources attached to a random cell, but the way shown here is closer to the intention.

```
%%writefile Pong.nml
<neuroml>
<include file="PongField.nml" />
<include file="PongPlayer.nml" />

<!-- Add a pong field type, and players -->
<PongField id="MyFirstWhat" hw="1" hl="1.5" />
<PongPlayer id="MyPlayer" noise_intensity="900" />

<network id="Net" type="networkWithTemperature" temperature="37degC" >
    <!-- Add said enviromnent and players -->
    <population id="Pong" component="MyFirstWhat" size="1"/>
    <population id="Players" component="MyPlayer" size="2"/>
</network>

<Simulation id="Sim" length="10 s" step="0.1 ms" target="Net" seed="20190109" > <!-
↪- seed used -->

    <!-- Use EdenCustomSetup ! -->
    <EdenCustomSetup filename="Pong_CustomSetup.gen.txt"/>

    <!-- Use EdenOutputFile to sample less frequently than on every timestep -->
    <EdenOutputFile id="MyFirstOutputFile" href="results.gen.txt" format="ascii_v0
↪" sampling_interval="5 msec">
        <OutputColumn id="vx" quantity="Pong[0]/px"/>
        <OutputColumn id="vy" quantity="Pong[0]/py"/>
        <OutputColumn id="p1" quantity="Pong[0]/paddle_1_x"/>
        <OutputColumn id="p2" quantity="Pong[0]/paddle_2_x"/>
        <OutputColumn id="s1" quantity="Pong[0]/score_1"/>
        <OutputColumn id="s2" quantity="Pong[0]/score_2"/>
    </EdenOutputFile>
</Simulation>
<Target component="Sim"/>
</neuroml>
```

```
Writing Pong.nml
```

We'll be using <EdenCustomSetup> to set up what we cannot through pure NeuroML.

Since we are using <VariableRequirement>s, for them to work we'll have to *point* their target for each instance. That means to point the field's paddles to the players' requested control force, and also the players to the ball's horizontal position, and also the current position of the paddle for each player.

While at it, add some variation between the two players' control stiffness, otherwise they would both be moving both paddles identically (even under noise interference).

```
%%writefile Pong_CustomSetup.gen.txt
```

(continues on next page)

(continued from previous page)

```
set cell Pong all all paddle_1_a_per_s2 Players[0]/request_a_per_s2 # paddles need␣
↪to know ...
set cell Pong all all paddle_2_a_per_s2 Players[1]/request_a_per_s2 # what the␣
↪players want

set cell Players all all ball_x Pong[0]/px # they both track px
set cell Players all all paddle_x multi # they use different paddles
values Pong[0]/paddle_1_x Pong[0]/paddle_2_x
set cell Players all all force_per_deviation multi  # different behaviours␣
↪otherwise they'll track exactly the same way. NEXT: allow values line with␣
↪comment.
values 40 30
```

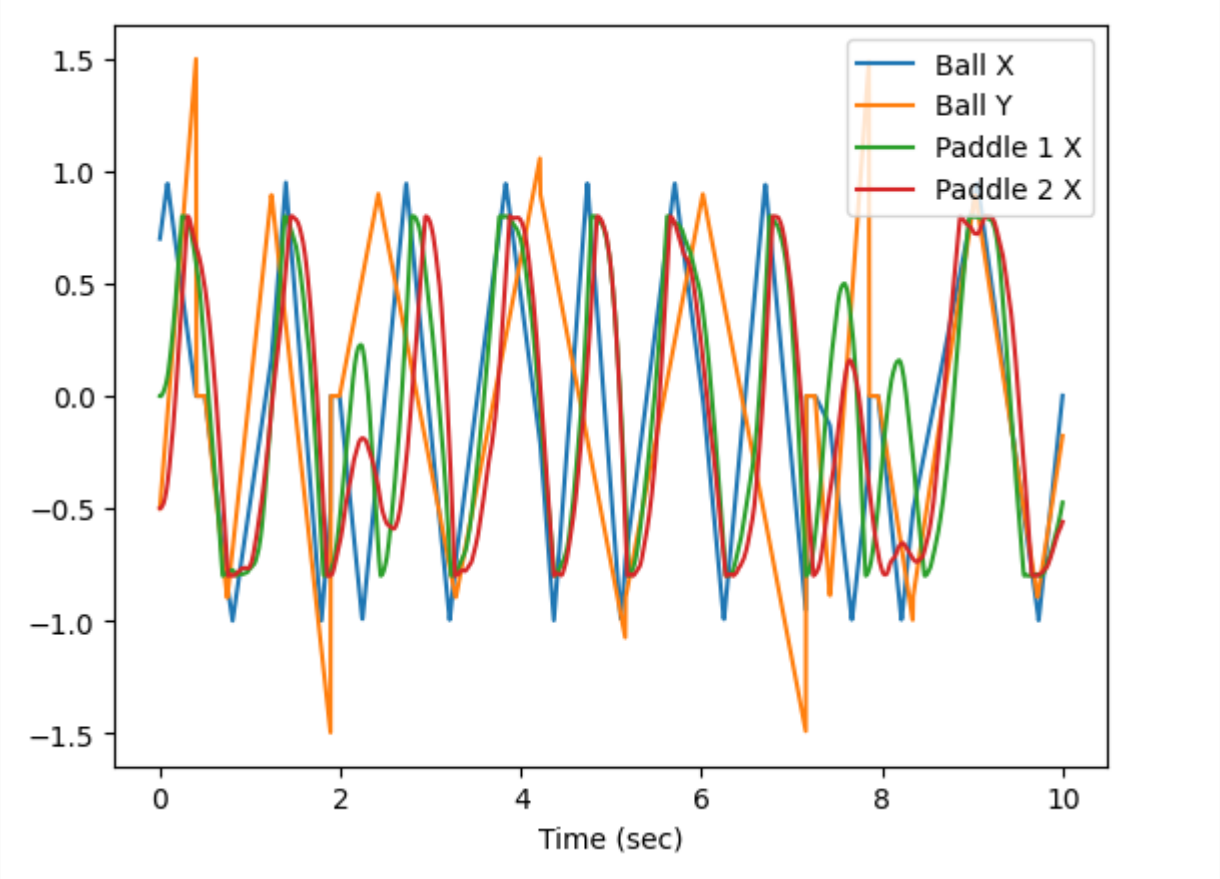
```
Writing Pong_CustomSetup.gen.txt
```

15.7 Displaying results

Let's run this simulation:

```
!pip install -q eden-simulator
```

```
from eden_simulator import runEden; from matplotlib import pyplot as plt
results = runEden('Pong.nml')
t = results['t']; px, py, p1, p2, s1, s2 = (results[f'Pong[0]/{x}'] for x in ('px',
↪ 'py', 'paddle_1_x', 'paddle_2_x', 'score_1', 'score_2'))
plt.plot(results['t'],px, label='Ball X')
plt.plot(results['t'],py, label='Ball Y')
plt.plot(results['t'],p1, label='Paddle 1 X')
plt.plot(results['t'],p2, label='Paddle 2 X')
# plt.plot(results['t'],s1, label='Player 1 score')
# plt.plot(results['t'],s2, label='Player 2 score')
plt.xlabel('Time (sec)'); plt.legend();
# plt.xlim((0.16, 0.17))
# plt.xlim((0.95, 1))
```



Each line helps us visualise the game:

- The `Ball X` line shows how the ball bounces between walls. Notice that the "bouncing period" may change when the ball hits a paddle and gets a different `vx`.
- The `Ball Y` line shows how the ball bounces between paddles. If it exceeds the usual range in either direction, it's probably a point!
- `Paddle 1` and `Paddle 2` show how the two paddles try to anticipate `Ball X`. It's not clear when it's a hit or not, but we have `Ball Y` for that.

Let's use the routines made previously (page 174) to better visualise the game's progression:

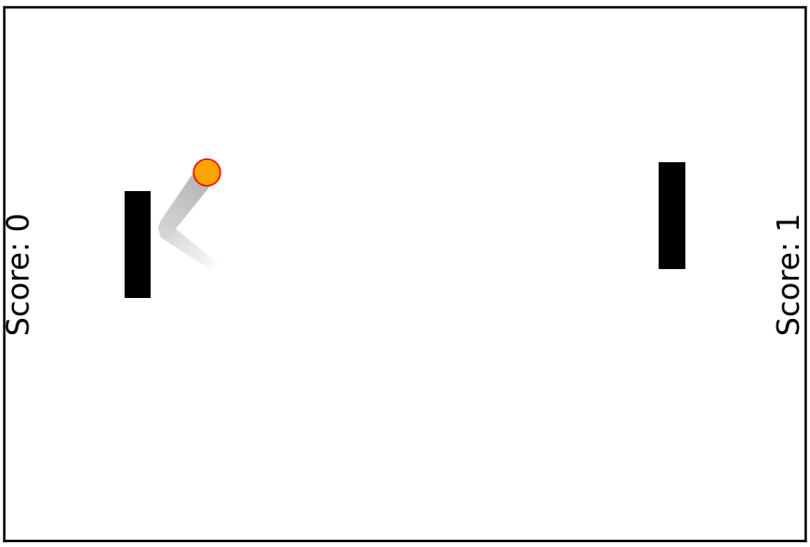
```
fig, ax = plt.subplots()
time_fade = 0.1 # how long (in time) should the ball's trace be?
def animate(time):
    ax.clear()
    # Get the recent trace for the ball
    ii = np.logical_and((time - time_fade) < t , t <= time) # could also do binary␣
↪search
    ii, = np.where(ii)
    # Cut off instant large transitions caused by scoring, keep only after then␣
↪last big one (if it exists)
    cut_locations = abs(np.diff(py[ii])) > field_hh/2.1
    cut_locations = [1]+list(cut_locations) # add guard value to keep whole␣
↪sequence, if there's no reset
    back_idx = -1-(np.argmax(cut_locations[::-1])) # search from end to start for␣
↪the latest cut
    ii = ii[back_idx:] #
    # Set up the motion line
    tt = t[ii]; xy = np.vstack( (px[ii], py[ii])).T
    wmax = ball_r*.8; wmin = ball_r*.3
```

(continues on next page)

(continued from previous page)

```
    weight = (tt - (time- time_fade))/time_fade
    width = (wmin) + (wmax-wmin)*weight
    fancy_streamline(ax, xy, width, weight*.3, cmap='gray_r', clim=(0,1), alpha=1)
    draw_field(ax, *xy[-1], p1[ii][-1], p2[ii][-1], s1[ii][-1], s2[ii][-1], time)

if 0 == 1: animate(0.13) # useful for debugging
else:
    import matplotlib.animation as animation
    frames_per_sec = 20
    sim_per_real = 1/2
    sim_duration = 10
    ani = animation.FuncAnimation(fig, animate, frames=np.linspace(0, sim_duration,
↪ int(sim_duration*frames_per_sec/sim_per_real))[:], interval = (1000/frames_per_
↪sec), repeat_delay=1000)

    plt.close() #to avoid the redundant pyplot of only the first frame coming out?
    from IPython.display import display, HTML
    display(HTML(ani.to_jshtml()))
    # Consider also:
    # writer = animation.FFMpegWriter( fps=frames_per_sec, bitrate=1800)
    # ani.save("movie.mp4", writer=writer)
```

```
<IPython.core.display.HTML object>
```



Seems like it's really working. 🏓

15.8 Next steps

The Pong game is set up, but the players follow a simple linear control rule. Think of neural networks that can learn to play the game competitively (also using hints like score and the opponent's paddle position) and see how they perform. Don't despair at that point; *persevere* and study the factors that lead to skill.

Consider how to *anticipate* the ball's course (including walls!), use the paddle's edges to build up speed, possibly even (if the paddles' locations are available) fake out the strike and/or adapt to the opponent's weaknesses.

Also think of rendering the image into pixels (how would one go around it? hint: sample the x,y with different activation thresholds provided through `<EdenCustomSetup>`…), and stimulate a visual-motor-cortex circuit like in Anwar et al. 2022[282] (code[283]).

[282] https://doi.org/10.1371/journal.pone.0265808
[283] https://github.com/NathanKlineInstitute/SMARTAgent

See also a similar experiment[284] using the NEST simulator.

Remember to compare performance with a at least better linear controller (say, PID) as well. Was all this effort and computational intensity to run competent neural networks worth it? How so?

[284] https://nest-simulator.readthedocs.io/en/stable/auto_examples/pong/run_simulations.html

CHAPTER

SIXTEEN

TUTORIAL: A BALANCED EXCITATORY-INHIBITORY NETWORK WITH DELTA SYNAPSES

A popular network dynamics model involves two interconnected populations of neurons: those in the first "excitatory" population stimulate (depolarise) all post-synaptic neurons, while neurons of the "inhibitory" population suppress (repolarise) all their post synaptic neurons, whether they are excitatory or inhibitory. Various interesting patterns can emerge out of such systems, depending on the particular parameters of the excitation-inhibition balance. A macro-scale, rate-based formulation is the Wilson-Cowan[285] model. A more detailed formulation with LIF point neurons, impulse-based synapses, and stochastic external input has been studied by Brunel[286]: on paper regarding general statistics and equilibria, but also with numerical simulations for interesting parameter combinations.

This chapter shows how to model such a network using EDEN. The network can be expressed in conventional NeuroML – save for the post-synaptic effect, which is modelled as instantaneous *jumps* in membrane potential or *impulses* of current. This effect could be approximated with short enough current pulses (as the numerical schemes used allow); but now it can also be declared precisely, thanks to EDEN's <WritableRequirement> (page 123) extension to NeuroML.

Note that the same synapse model can be used for both inter-neuron synaptic `<projection>`s, as well as for the excitatory <poissonFiringSynapse>[287]s driving the cell; the semantics are preserved.

First, let's set up the network's parameters. We'll opt for asynchronous firing for this example:

```
# Adapted from the Brian adaptation: https://brian2.readthedocs.io/en/stable/
↪examples/frompapers.Brunel_2000.html
sim_duration = 0.1 # just for demonstration
# network parameters
N_exc = 4000 # NB: the original paper shows for 10000, the behaviour doesn't␣
↪change that much
N_inh = int(N_exc/4)
p_connections = 0.1
N_ext_inputs = int(round(p_connections * N_exc))

# Cell parameters in SI units
cell_tau = 20e-3 # sec
cell_vth = 20e-3 # volt
cell_vr  = 10e-3 # volt
cell_ref =  2e-3 # sec

# synapse parameters
syn_delta = 0.1e-3 # volt
conn_delay = 1.5e-3 # sec

# system state parameters
# for asynchronous spiking
g = 5 # relative inhibitory to excitatory synaptic strength
nu_ext_over_nu_thr = 2 # ratio of external stimulus rate to threshold rate
```

(continues on next page)

[285] https://doi.org/10.1016/S0006-3495(72)86068-5
[286] https://doi.org/10.1023/A:1008925309027
[287] https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#poissonfiringsynapse

(continued from previous page)

```
# derived properties
nu_thr = cell_vth / (syn_delta * N_ext_inputs * cell_tau) # hertz
# external stimulus
nu_ext = nu_ext_over_nu_thr * nu_thr
# N_ext_inputs at a rate of nu_ext and weight of syn_delta for every cell
```

Then let's define the cell and synapse models:

```
cell_defs = f'''
<ComponentType name="VoltTauCell" extends="baseCellMembPotCap"> <!-- not really␣
↪cap but since baseSynapse exposes current... -->
    <Parameter name="v_reset" dimension="voltage"/>
    <Parameter name="v_thresh" dimension="voltage"/>
    <Parameter name="tau_mem" dimension="time"/>
    <Dynamics>
            <StateVariable name="lastSpikeTime" dimension="time"/>
        <StateVariable name="v" dimension="voltage" exposure="v"/>

        <ConditionalDerivedVariable name="voltage_rate" dimension="voltage_per_time
↪">
                    <Case condition="t .gt. lastSpikeTime + tau_refrac" value="(-
↪v) / tau_mem"/>
                    <Case value="0"/>
            </ConditionalDerivedVariable>
            <TimeDerivative variable="v" value="voltage_rate" />

        <OnStart> <!-- Start at resting state -->
            <StateAssignment variable="v" value="v_reset"/> <!-- TODO or zero? -->
            <StateAssignment variable="lastSpikeTime" value="t - tau_refrac" />
        </OnStart>

            <OnCondition test="(v .gt. v_thresh) .and. (t .gt. lastSpikeTime + tau_
↪refrac)">
                    <EventOut port="spike"/>
                    <StateAssignment variable="lastSpikeTime" value="t" />
                    <StateAssignment variable="v" value="v_reset" />
            </OnCondition>
    </Dynamics>
</ComponentType>
<VoltTauCell id="IafTauRefNeuron" tau_mem="{cell_tau} s" tau_refrac="{cell_ref} s"␣
↪v_thresh="{cell_vth} V" v_reset="{cell_vr} V"  C="0.2nF" /> <!-- C is unused,␣
↪see above -->
'''
syns_defs = f'''
<ComponentType name="DeltaSyn" extends="baseSynapse" description="A voltage-
↪impulse synapse model. In case cell capacitance is variable, consider a current-
↪impulse model instead.">
    <Constant name="AMP" dimension="current" value="1 A"/>
    <Parameter name="delta" dimension="voltage" description="Weight really"/> <!--␣
↪TODO access the weight of the synapse as defined on the projection, somehow! -->
    <WritableRequirement name="v" dimension="voltage"/>
    <Dynamics>
            <DerivedVariable name="i" exposure="i" dimension="current" value="0 *␣
↪AMP" />
            <OnEvent port="in">
                    <StateAssignment variable="v" value="v+delta"/>
            </OnEvent>
    </Dynamics>
</ComponentType> <!-- TODO updating., init weight, etc. -->
```

(continues on next page)

(continued from previous page)

```
<DeltaSyn id="SynExc" delta="{syn_delta} V"/>
<DeltaSyn id="SynInh" delta="{-g*syn_delta} V"/>
'''
```

Then generate the synaptic projections between populations. (Note that the XML format starts to become a nuisance for several million synapses…)

```
import numpy as np
rng = np.random.default_rng(seed=5)
def RandomUniformProjection(rng, N_from, N_to, prob, symmetric):
    M = rng.uniform(size=(N_from,N_to))
    if symmetric: M += np.eye(N_from, N_to) # exclude diagonal
    ei, ej = np.where(M < prob)
    return ei, ej

popsizes = {'Exc': N_exc, 'Inh': N_inh}
def get_synapses(projname, source, target, syntype, weight=1, delay_sec=0, prob=0.
↪1):
    nml_lines = [f'''\t<projection id="{projname}" presynapticPopulation="{source}
↪" postsynapticPopulation="{target}" synapse="{syntype}">''']
    syn_i, syn_j = RandomUniformProjection(rng, popsizes[source], popsizes[target],
↪ prob, (source == target))

    nml_lines +=  [ f'\t\t<connectionWD id="{ii}" preCellId="{pre}" postCellId="
↪{post}" weight="{weight}" delay="{delay_sec} s"/>' for ii, (pre, post) in␣
↪enumerate(zip(syn_i, syn_j))]
    nml_lines += ['\t\t</projection>']
    return nml_lines, len(syn_i)
net_lines = []
net_ee, n_ee = get_synapses("ee_synapses", 'Exc', 'Exc', 'SynExc', weight=+1,␣
↪delay_sec=conn_delay, prob=p_connections); net_lines += net_ee
net_ei, n_ei = get_synapses("ei_synapses", 'Exc', 'Inh', 'SynExc', weight=+1,␣
↪delay_sec=conn_delay, prob=p_connections); net_lines += net_ei
net_ie, n_ie = get_synapses("ie_synapses", 'Inh', 'Exc', 'SynInh', weight=-g,␣
↪delay_sec=conn_delay, prob=p_connections); net_lines += net_ie
net_ii, n_ii = get_synapses("ii_synapses", 'Inh', 'Inh', 'SynInh', weight=-g,␣
↪delay_sec=conn_delay, prob=p_connections); net_lines += net_ii
```

Then assemble the NeuroML components, the synaptic projections, and many many (`N_ext_inputs`) independently firing external synapses attached to each neuron, into the model file:

```
tabline = '\n    ' # annoyingly enough, \ may not exist at all in a f string, not␣
↪even in brackets. Fixed in Python 3.12+
nml_file=''
nml_file=f'''
<neuroml>

{cell_defs}
{syns_defs}
<poissonFiringSynapse id="PoissonInput" averageRate="{nu_ext} Hz" synapse="SynExc"␣
↪spikeTarget="./GenericDeltaSynExc"/>

<network id="Net">
    <population id="Exc" component="IafTauRefNeuron" size="{N_exc}"/>
    <population id="Inh" component="IafTauRefNeuron" size="{N_inh}"/>

    {tabline.join(net_lines)}

    <inputList id="InpExc" population="Exc" component="PoissonInput">
        {tabline.join([ f'<input id="{i}" target="Exc[{idx}]" destination="synapses
↪"/>' for i, idx in enumerate([cell for cell in range(N_exc) for jjj in range(N_
```

(continues on next page)

(continued from previous page)

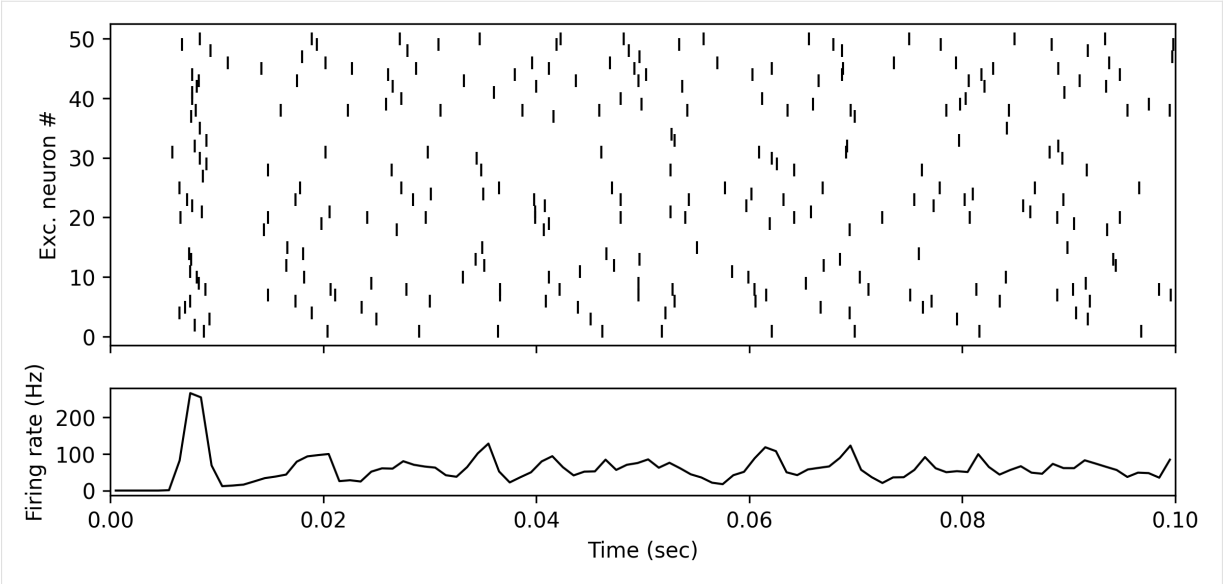
```
↪ext_inputs)]) ])}
    </inputList>
    <inputList id="InpInh" population="Inh" component="PoissonInput">
        {tabline.join([ f'<inputW id="{i}" target="Inh[{idx}]" destination=
↪"synapses" weight="1"/>' for i, idx in enumerate([cell for cell in range(N_inh)␣
↪for jjj in range(N_ext_inputs)]) ])}
    </inputList>
</network>
</neuroml>
'''
with open('Model.nml', 'wt') as f: f.write(nml_file)

sim_file = f'''<Lems><include href="Model.nml"/><Simulation id="MySim" length="
↪{sim_duration} s" step="100 us" target="Net">
    <EventOutputFile id="SpikesExc" fileName="spikes_exc.gen.txt" format="TIME_ID">
        {tabline.join([ f'<EventSelection id="{i}"  select="Exc[{i}]" eventPort=
↪"spike"/>' for i in range(N_exc)])}
    </EventOutputFile>
    <EventOutputFile id="SpikesInh" fileName="spikes_inh.gen.txt" format="TIME_ID">
        {tabline.join([ f'<EventSelection id="{i}"  select="Inh[{i}]" eventPort=
↪"spike"/>' for i in range(N_inh)])}
    </EventOutputFile>
</Simulation>
<Target component="MySim"/></Lems>'''
with open('Sim.nml', 'wt') as f: f.write(sim_file)

# it's quite big until we use h5 so let's save space
del nml_file, net_lines, net_ee, net_ii, net_ei, net_ie
import gc; gc.collect();
```

And run the simulation:

```
import eden_simulator as eden
tra, eve = eden.runEden('Sim.nml', reload_events=True, threads=1)
```

Let's see the raster plot of the first 50 excitatory neurons. Since connectivity is the same over the whole population, they should represent a random sample. And also the firing rate of the whole population, using a histogram. (Could also use sliding window or other convolutions but it's also good with enough bins)

```
import matplotlib; from matplotlib import pyplot as plt
fig, axs = plt.subplots(2,1,sharex=True,figsize=[9,4],height_ratios=[3,1], dpi=300)
for i in range(50):
    spikes=eve[f'Exc[{i}]']
    axs[0].plot(spikes, [i+1]*len(spikes), "|k")
axs[0].set_xlim([0,sim_duration]); axs[0].set_ylabel('Exc. neuron #');
binsize = 0.001 # seconds
counts, _ = np.histogram(
    [t for i in range(N_exc) for t in eve[f'Exc[{i}]']],
    range=[0,sim_duration], bins = round(sim_duration/binsize))
axs[1].plot((np.arange(len(counts))+.5)*binsize, counts/(N_exc*binsize),'-k',lw=1)
# axs[1].locator_params(axis='y', nbins=6)# axs[1].set_ylim([0, 130]);
axs[1].set_xlim(0, sim_duration)
axs[1].set_ylabel('Firing rate (Hz)'); axs[1].set_xlabel('Time (sec)');
axs[1].yaxis.set_major_locator(matplotlib.ticker.MultipleLocator(base=100))
# axs[1].axhline((1+nu_ext_over_nu_thr)*nu_thr,ls='--',c='orange')
```


Exc. neuron #
0
10
20
30
40
50
Firing rate (Hz)
0
100
200
0.00
0.02
0.04
0.06
0.08
0.10
Time (sec)

CHAPTER

SEVENTEEN

EXAMPLE: TSODYKS, URIEL, MARKRAM (2000)

Here we see how to implement the network synchrony generation model from Tsodyks, Uziel, Markram (2000)[288]. This network exhibits *network-wide synchronised firing* (along with other activity), using only event-based synapses!

This example requires and demonstrates the use of two of EDEN's extensions to NeuroML:

- Parameter variability (page 89) among point neurons and synapses;
- Multiflux (page 121): Since the cell models the *time constant* rather than passive leak and capacitance of the membrane, stimuli come in terms of *voltage!* Such that $\dot{V}_m = \frac{-V_m + I_{ext}}{\tau_m}$. To be fair, in this case we *could* introduce physical leak and (arbitrary) capacitance to the model, and *carefully* adjust most parameters to the same current-based effect … but this starts being quite the burden, and we can't always stay within NeuroML2 anyway.

Often we encounter physical parameters, whose variability is modelled by a "bell curve" … but not really: All real numbers, even unrealistic or physically impossible ones, can come out of the true $\mathcal{N}$ distribution. This can prove most inconvenient!

- A simple workaround, taken for several parameters in the featured model, is to follow a *truncated* normal distribution: samples which would fall "too far" are simply rejected and resampled. Good luck finding *this* in your favourite "general-purpose" model generator!

```
import numpy as np; import matplotlib.pyplot as plt
# set seed for reproducible figures
np.random.seed(123)#43

import scipy; from scipy import stats # for truncated normal
def truncated_normal(loc, scale, bounds, size):
    """Normal distribution truncated within bounds

    loc -- mean ("centre") of the distribution
    scale -- standard deviation (spread or "width") of the distribution
    bounds -- list of min and maximum
    size -- number of samples
    """
    bounds = np.array([bounds] * size)

    s = scipy.stats.truncnorm.rvs(
        (bounds[:, 0] - loc) / scale, (bounds[:, 1] - loc) / scale, loc=loc,␣
↪scale=scale
    )
    return s
```

The point-neuron model is almost the same as <iafTauRefCell>[289] except:

1. it admits stimulus current as *voltage*;
2. it includes a bias-current term which is different for each one neuron.

[288] https://doi.org/10.1523/JNEUROSCI.20-01-j0003.2000
[289] https://docs.neuroml.org/Userdocs/Schemas/Cells.html#iaftaurefcell

We handle the former with a `<DerivedVariable>` aggregating inbound `V_syn`, and the latter with `<EdenCustomSetup>`.

```
cell_defs = '''
<ComponentType name="VoltTauCell" extends="baseCellMembPotCap"> <!-- not really␣
↪*Cap* but since baseSynapse exposes current... -->

    <Parameter name="v_reset" dimension="voltage"/>
    <Parameter name="v_thresh" dimension="voltage"/>
    <Parameter name="V_b" dimension="voltage"/>
    <Parameter name="tau_mem" dimension="time"/>
    <Parameter name="tau_refrac" dimension="time"/>

    <Dynamics>
            <StateVariable name="lastSpikeTime" dimension="time"/>
        <StateVariable name="v" dimension="voltage" exposure="v"/>
        <DerivedVariable name="V_syn" dimension="voltage" select="synapses[*]/V_syn
↪" reduce="add" />

        <ConditionalDerivedVariable name="voltage_rate" dimension="voltage_per_time
↪">
                    <Case condition="t .gt. lastSpikeTime + tau_refrac" value="(-v␣
↪+ V_syn + V_b) / tau_mem"/>
                    <Case value="0"/>
            </ConditionalDerivedVariable>

            <TimeDerivative variable="v" value="voltage_rate" />

        <OnStart> <!-- Start at resting state -->
            <StateAssignment variable="v" value="v_reset"/>
            <StateAssignment variable="lastSpikeTime" value="t - tau_refrac" />
        </OnStart>

            <OnCondition test="(v .gt. v_thresh) .and. (t .gt. lastSpikeTime + tau_
↪refrac)">
                    <EventOut port="spike"/>
                    <StateAssignment variable="lastSpikeTime" value="t" />
                    <StateAssignment variable="v" value="v_reset" />
            </OnCondition>
    </Dynamics>
</ComponentType>

<!-- NB: V_b may be overriden with EdenCustomSetup -->
<VoltTauCell id="ExcNeuron" tau_mem="30ms" tau_refrac="3 ms" v_thresh="15 mV" v_
↪reset="13.5 mV" V_b="0mV"  C="0.2nF" /> <!-- C is unused, see above -->
<VoltTauCell id="InhNeuron" tau_mem="30ms" tau_refrac="2 ms" v_thresh="15 mV" v_
↪reset="13.5 mV" V_b="0mV"  C="0.2nF" />
'''
```

Let's fill in the values for `V_b`:

```
rng = np.random.default_rng(seed=5)
def GetRandomIbMillivolt(rng,N,ib_mean_mvolt):
    # paper gives range of 0.05 mV but population bursts are not visible with that␣
↪value
    # -> increased to 1 mV range
    return np.random.uniform(ib_mean_mvolt - 0.5, ib_mean_mvolt + 0.5, size=N)

eden_setup_lines = []
N_exc = 400; N_inh = 100

ib_vals = GetRandomIbMillivolt(rng, N=N_exc, ib_mean_mvolt=15)
```

(continues on next page)

(continued from previous page)

```
# ib_vals = exc_neurons.I_b / mV # override with Brian's
eden_setup_lines += [
    f'set cell Exc all all V_b multi mV',
    'values '+' '.join([f'{v}' for v in ib_vals]) ]
ib_vals = GetRandomIbMillivolt(rng, N=N_inh, ib_mean_mvolt=15)
# ib_vals = inh_neurons.I_b / mV # override with Brian's
eden_setup_lines += [
    f'set cell Inh all all V_b multi mV',
    'values '+' '.join([f'{v}' for v in ib_vals]) ]
```

The synapse model is unique but not so special, other than randomised `U`, `tau_rec`, `tau_facil`. Until EDEN supports LEMS inheritance we'll have to duplicate some lines:

```
syns_defs = '''
<ComponentType name="DynSyn" extends="baseSynapse">
    <Parameter name="A" dimension="voltage"/>
    <Parameter name="U" dimension="none"/>
    <Parameter name="tau_I" dimension="time"/>
    <Parameter name="tau_rec" dimension="time"/>
    <Constant name="AMP" dimension="current" value="1 A"/>

    <Dynamics>
        <StateVariable name="x" dimension="none"/>
        <StateVariable name="y" dimension="none"/>
        <DerivedVariable name="z" dimension="none" value="1 - x - y"/>
        <DerivedVariable name="i" exposure="i" dimension="current" value="0 * AMP"␣
↪/> <!-- because baseSynapse requires it -->
        <DerivedVariable name="V_syn" exposure="V_syn" dimension="voltage" value=
↪"A*y" />

        <TimeDerivative variable="x" value="+z / tau_rec"/>
        <TimeDerivative variable="y" value="-y / tau_I"/>

        <OnStart>
            <StateAssignment variable="x" value="1"/>
            <StateAssignment variable="y" value="0"/>
        </OnStart>

        <OnEvent port="in">
            <StateAssignment variable="y" value="y + U * x"/>
            <StateAssignment variable="x" value="x - U * x"/>
        </OnEvent>

    </Dynamics>
</ComponentType>
<ComponentType name="DynSynFacil" extends="baseSynapse">
    <Parameter name="A" dimension="voltage"/>
    <Parameter name="U" dimension="none"/>
    <Parameter name="tau_I" dimension="time"/>
    <Parameter name="tau_rec" dimension="time"/>
    <Parameter name="tau_facil" dimension="time"/>
    <Constant name="AMP" dimension="current" value="1 A"/>

    <Dynamics>
        <StateVariable name="x" dimension="none"/>
        <StateVariable name="y" dimension="none"/>
        <StateVariable name="u" dimension="none"/>
        <DerivedVariable name="z" dimension="none" value="1 - x - y"/>
        <DerivedVariable name="i" exposure="i" dimension="current" value="0 * AMP"␣
↪/> <!-- because baseSynapse requires it -->
        <DerivedVariable name="V_syn" exposure="V_syn" dimension="voltage" value=
```

(continues on next page)

(continued from previous page)

```
↪"A*y" />

        <TimeDerivative variable="x" value="+z / tau_rec"/>
        <TimeDerivative variable="y" value="-y / tau_I"/>
        <TimeDerivative variable="u" value="-u / tau_facil"/>

        <OnStart>
            <StateAssignment variable="x" value="1"/>
            <StateAssignment variable="y" value="0"/>
            <StateAssignment variable="u" value="0"/>
        </OnStart>

        <OnEvent port="in">
            <StateAssignment variable="u" value="u + U*(1-u)"/>
            <StateAssignment variable="y" value="y + u * x"/>
            <StateAssignment variable="x" value="x - u * x"/>
        </OnEvent>
    </Dynamics>
</ComponentType>

<DynSyn      id="ToExcSyn" A="0 mV" U="0" tau_I="3 ms" tau_rec="800 ms"/>
<DynSynFacil id="ToInhSyn" A="0 mV" U="0" tau_I="3 ms" tau_rec="100 ms" tau_facil=
↪"1000 ms" />
'''
```

Now, let's create all these synapses: determine pre-synaptic and post-synaptic cells, create the stochastic distributions of the various parameters, and create the `set` lines for `<EdenCustomSetup>`. (Or better, generate them while writing to the final file, to save on RAM.)

```
# A lot of parameters to randomise....
def GetSynsSetupDict(N_syn, tau_I, A, U, tau_rec, tau_facil=None, override=None):
    ret = {}
    ret['tau_I'] = {'vals': tau_I, 'unit': 'ms'}

    A_min = min(0.2 * A, 2 * A)
    A_max = max(0.2 * A, 2 * A)
    ret['A'] = {
        'vals': truncated_normal(
            A, 0.5 * abs(A), [A_min, A_max], size=N_syn
        ), 'unit':'mV' }
    assert not any(ret['A']['vals'] < A_min)
    assert not any(ret['A']['vals'] > A_max)
    if override: ret['A']['vals'] = override.A / mV

    U_mean, U_min, U_max = U
    ret['U'] = {'vals': truncated_normal(U_mean, 0.5 * U_mean, [U_min, U_max],␣
↪size=N_syn), 'unit':''}
    assert not any(ret['U']['vals'] <= U_min)
    assert not any(ret['U']['vals'] > U_max)
    if override: ret['U']['vals'] = override.U

    tau_min = 5
    ret['tau_rec'] = {
        'vals': truncated_normal(
            tau_rec, 0.5 * tau_rec, [tau_min, np.inf], size=N_syn
        ), 'unit': 'ms' }
    assert not any(ret['tau_rec']['vals'] <= tau_min)
    if override: ret['tau_rec']['vals'] = override.tau_rec / ms

    if tau_facil:
        ret['tau_facil'] = {
```

(continues on next page)

(continued from previous page)

```
            'vals': truncated_normal(
                tau_facil, 0.5 * tau_facil, [tau_min, np.inf], size=N_syn
            ), 'unit': 'ms' }
        assert not any(ret['tau_facil']['vals'] <= tau_min)
        if override: ret['tau_facil']['vals'] = override.tau_facil / ms

    return ret
```

```
# sample sparse... then randomise
def RandomUniformProjection(rng, N_from, N_to, prob, symmetric):
    M = rng.uniform(size=(N_from,N_to))
    if symmetric: M += np.eye(N_from, N_to) # exclude diagonal
    ei, ej = np.where(M < prob)
    return ei, ej

def GetSynsSetupLines(projname, property, vals, unit):
    # check if iterable https://stackoverflow.com/a/1952655
    iter8 = None
    try:
        iter8 = iter(vals)
    except TypeError: pass

    lines = [ f"set synapse {projname} all post {property} {'multi' if iter8 else␣
↪vals} {unit}" ]
    if iter8: lines += [ "values "+" ".join([ str(v) for v in vals]) ]
    return lines

popsizes = {'Exc': N_exc, 'Inh': N_inh}
def get_synapses(projname, source, target, tau_I, A, U, tau_rec, tau_facil=None,␣
↪override=None):
    nml_lines = [f'''\t<projection id="{projname}" presynapticPopulation="{source}
↪" postsynapticPopulation="{target}" synapse="{'ToExcSyn' if tau_facil is None␣
↪else 'ToInhSyn'}">''']
    syn_i, syn_j = RandomUniformProjection(rng, popsizes[source], popsizes[target],
↪ 0.1, (source == target))
    if override: syn_i, syn_j = override.i, override.j

    nml_lines +=  [ f'\t\t<connection id="{ii}" preCellId="{pre}" postCellId="
↪{post}"/>' for ii, (pre, post) in enumerate(zip(syn_i, syn_j))]
    nml_lines += ['\t\t</projection>']


    setup_dict = GetSynsSetupDict(len(syn_i), tau_I, A, U, tau_rec, tau_facil,␣
↪override)
    setup_lines = []
    for propname, d in setup_dict.items():
        setup_lines += GetSynsSetupLines(projname, propname, **d)
    return nml_lines, setup_lines, len(syn_i)
net_lines = []
net_ee, set_ee, n_ee = get_synapses("ee_synapses", 'Exc', 'Exc', tau_I=3, A= 1.8,␣
↪U=[0.5 , 0.1  , 0.9 ], tau_rec=800                ); net_lines += net_ee; eden_
↪setup_lines += set_ee
net_ei, set_ei, n_ei = get_synapses("ei_synapses", 'Exc', 'Inh', tau_I=3, A= 7.2,␣
↪U=[0.04, 0.001, 0.07], tau_rec=100, tau_facil=1000); net_lines += net_ei; eden_
↪setup_lines += set_ei
net_ie, set_ie, n_ie = get_synapses("ie_synapses", 'Inh', 'Exc', tau_I=3, A=-5.4,␣
↪U=[0.5 , 0.1  , 0.9 ], tau_rec=800                ); net_lines += net_ie; eden_
↪setup_lines += set_ie
net_ii, set_ii, n_ii = get_synapses("ii_synapses", 'Inh', 'Inh', tau_I=3, A=-7.2,␣
↪U=[0.04, 0.001, 0.07], tau_rec=100, tau_facil=1000); net_lines += net_ii; eden_
↪setup_lines += set_ii
```

Finally, let's put it all together and run the simulation:

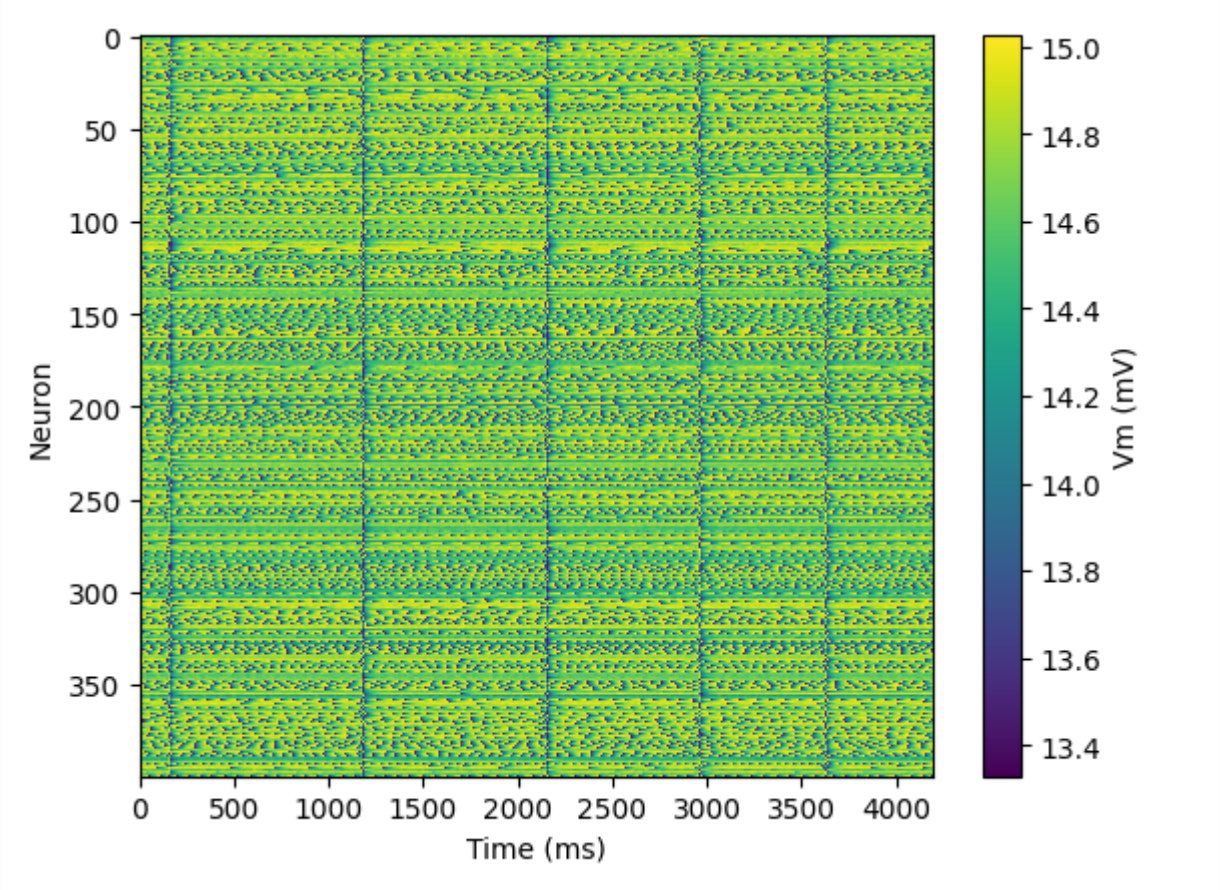
```
tabline = '\n    ' # annoyingly enough, \ may not exist at all in a f string, not␣
↪even in brackets. Fixed in Python 3.12+
nml_file=f'''
<neuroml>

{cell_defs}
{syns_defs}

<!-- Here comes the network -->
<network id="Net">
    <!-- Add a population with a single cell -->
    <population id="Exc" component="ExcNeuron" size="400"/>
    <population id="Inh" component="InhNeuron" size="100"/>
    {tabline.join(net_lines)}

</network>
<Simulation id="MySim" length="5.1 s" step="100 us" target="Net">
    <EdenOutputFile id="MyOutFile" href="results.gen.txt" format="ascii_v0"␣
↪sampling_interval="1 msec">
        {tabline.join([ f'<OutputColumn id="v_{i}"  quantity= "Exc[{i}]/v" output_
↪units="mV"/>' for i in range(N_exc)])}
    </EdenOutputFile>
    <EdenOutputFile id="MyOutSyns" href="file://syns.gen.txt" format="ascii_v0"␣
↪sampling_interval="1 msec">
        {tabline.join([ f'<OutputColumn id="x_{i}"  quantity= "ee_synapses[{i}]/
↪post/x"/>' for i in range(n_ee)])}
    </EdenOutputFile>
    <EventOutputFile id="SpikesExc" fileName="spikes_exc.gen.txt" format="TIME_ID">
        {tabline.join([ f'<EventSelection id="{i}"  select="Exc[{i}]" eventPort=
↪"spike"/>' for i in range(N_exc)])}
    </EventOutputFile>
    <EventOutputFile id="SpikesInh" fileName="spikes_inh.gen.txt" format="TIME_ID">
        {tabline.join([ f'<EventSelection id="{i}"  select="Inh[{i}]" eventPort=
↪"spike"/>' for i in range(N_inh)])}
    </EventOutputFile>

    <!-- Use EdenCustomSetup ! -->
    <EdenCustomSetup filename="CustomSetup.txt"/>
</Simulation>
<Target component="MySim"/></neuroml>
'''
with open('Model.nml', 'wt') as f: f.write(nml_file)
setup_file = '\n'.join(eden_setup_lines)
with open('CustomSetup.txt', 'wt') as f: f.write(setup_file)
```

Note that we are recording the `x` variable of every synapse just to show their average… `400 * 400 neurons * 0.1 density * 5000 msec * 18ish ASCII bytes per number` adds up to a lot! Having binary instead of ASCII output (TBD) would cut that down to 1/2 or 1/4, but complete solutions would be to either specify the *average itself* for recording, or to make a monstrosity with `<VariableReference>` powered pseudo-gap junctions into pseudo-cells for accumulation (not ideal).

```
import eden_simulator as eden
tra, eve = eden.runEden('Model.nml', reload_events=True, threads=1)
```

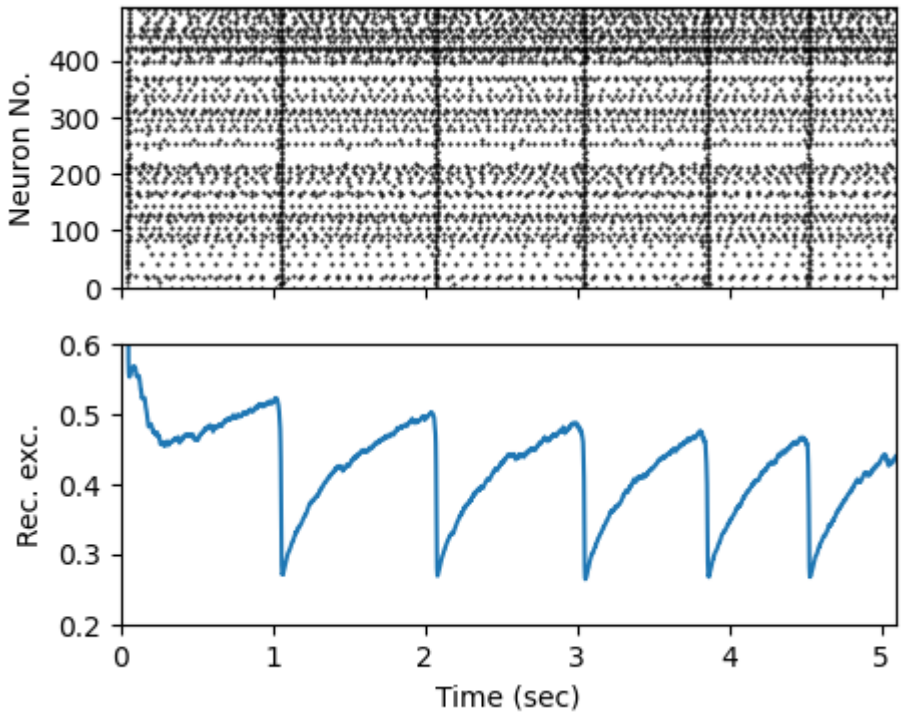
And sanity-check the network and see if there is synchronised firing:

```
v = np.array([tra[f'Exc[{i}]/v'] for i in range(N_exc)])
plt.imshow(v[:,900:], aspect='auto', interpolation='none'); plt.colorbar(label='Vm␣
↪(mV)')
plt.xlabel('Time (ms)'); plt.ylabel('Neuron');
```



And finally reproduce Figures 1a, 1c from the paper.

```
fig, axs = plt.subplots(2,1,sharex=True,figsize=[5,4],dpi=100) # or figsize=[11,4],
↪dpi=200
for i in range(0,N_exc,5): # every fifth neuron for decluttering, see paper
    spikes=eve[f'Exc[{i}]']
    axs[0].plot(spikes, [i]*len(spikes), ".k", ms=1)
for i in range(0,N_inh,5):
    spikes=eve[f'Inh[{i}]']
    axs[0].plot(spikes, [i+N_exc]*len(spikes), ".k", ms=1)
axs[0].autoscale(axis='x', tight=True)
axs[0].set_ylabel("Neuron No."); axs[0].autoscale(axis='y', tight=True)
xx = np.array([tra[f'ee_synapses[{i}]/post/x'] for i in range(n_ee)])
zz = np.mean(xx,axis=0)
axs[1].plot(tra['t'],zz)
axs[1].set_ylim([0.2, 0.6]);axs[1].autoscale(axis='x', tight=True)
axs[1].set_ylabel("Rec. exc.");axs[1].set_xlabel("Time (sec)");
```


Neuron No.
400
300
200
100
0
Rec. exc.
0.6
0.5
0.4
0.3
0.2
0
1
2
3
4
5
Time (sec)

CHAPTER

EIGHTEEN

EXAMPLE: REWARD-MODULATED STDP IN A VIRTUAL ENVIRONMENT - BRZOSKO, ZANNONE ET AL. (2017)

The brain is said to have smart neural structures which somehow learn, or at least adapt, to their surroundings and lead to – however basic – intelligent behaviours.

Can we see this in action? The neural network in Brzosko, Zannone et al. (2017)[290] (code)[291] learns to navigate toward a target, and then unlearn and relearn to reach a different target, with only *whether the target has been reached* as feedback.

This model learns to navigate with *sequential reward-modulated* STDP. Time is segmented into trials of the navigation task, and synapses are reinforced according to a time-symmetric "fire together, wire together" rule – but only at the end of each trial, and only if they succeeded. Meanwhile, spikes in general depress the involved synapses regardless of timing: this is how the SNN can *un*learn unproductive behaviours, which will prove handy once the target location *changes* without warning after several trials. Finally, weights are constrained between a maximum and minimum positive value.

Navigation happens in a (simulated, for now) physical environment. Its dynamics are so simple that it could be implemented in LEMS, but here we'll demonstrate how a SNN simulation can be plugged *into any arbitrary environment:* think of OpenAI Gym[292], the Metaverse[293] or even real life[294].

In the following, we'll get to (show how to) employ the following extensions to NeuroML:

- Per-cell variability (page 89): so that place cells are selective to different locations;
- <VariableReference> (page 107): to plug the agent's location into the place cells' activity;
- Streaming I/O (page 109): to interact with another external process, which runs the mechanical simulation in tandem;
- Accessing a point neuron's state variables from a synapse (page 123): for both delta synapses and as a workaround, so that post-synapses can observe non-standard state variables of their neuron.

18.1 Architecture of the model and experimental rig

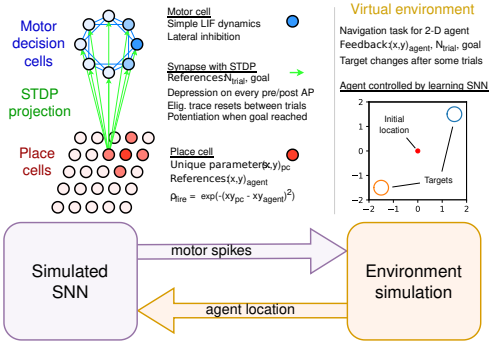
The model is basically an embodied A.I. agent[295], in terms of buzzwords: its physical presence or *body* is modelled as a freely-moving point in 2-dimensional space, which is guided by a spiking neural network (SNN) which is the "brains" for the body.

The SNN has two layers of neurons: The input layer has "place cells" which fire selectively to the body's location, and the output layer has "action cells" which each select a direction of travel for the body. All input cells project to all action cells, with synaptic weights adjusted during simulation. Action cells project to other cells; their influence is mutually inhibitory for dissimilar directions, and excitatory for similar ones.

[290] https://doi.org/10.7554/eLife.27756
[291] https://modeldb.science/233396
[292] https://en.wikipedia.org/wiki/Products_and_applications_of_OpenAI#Gym
[293] https://en.wikipedia.org/wiki/Metaverse
[294] https://en.wikipedia.org/wiki/Real_life
[295] https://en.wikipedia.org/wiki/AI_agent

The body follows the direction pointed to by the SNN's action cells (weighted by firing rate), while being physically limited to its rectangular-shaped playing field. Synapses from place cells associated with walls, to directions toward said walls, are excluded from the SNN, to prevent our agent from getting stuck. (Here's an assignment for later, relax this crutch.)

Our experimental setup looks like this:



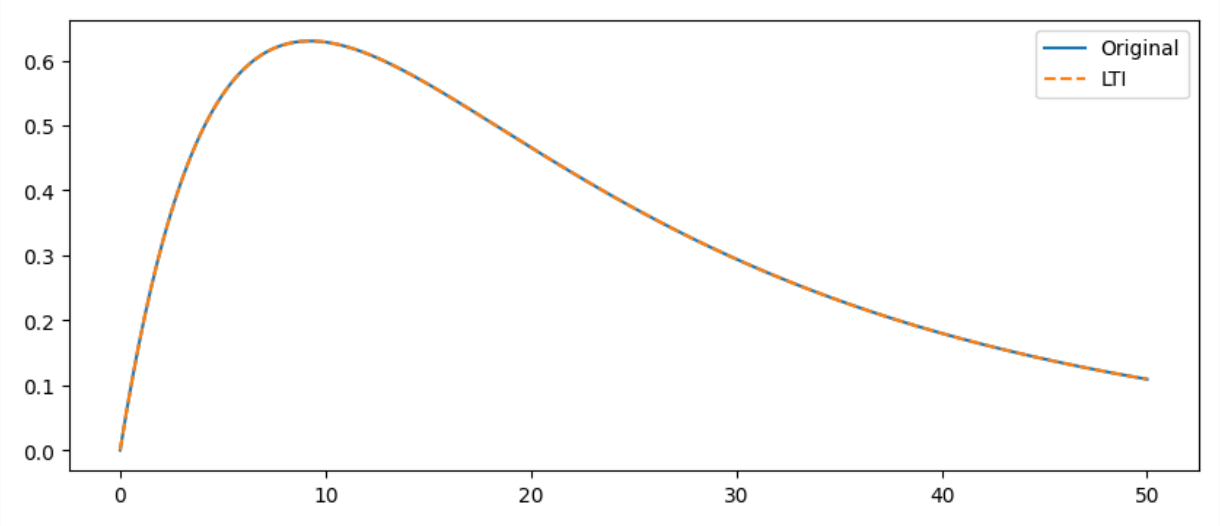
18.2 Implementing the model

To begin with, the synapses connecting the input layer to the output layer inject bi-exponentially-decaying current - but there is a quirk in the original code: when an output neuron fires, all inbound synapses are *silenced* instantly! We could have the synapse models peek into the cells to detect such spikes (as we'll have to later on for STDP). But we'll take this chance to show how exponential-current synapse models can be converted to delta synapses (page 123) changing the aggregate current, which we can then simply reset along with membrane potential in the cell's equations.

The synapse model was originally written as an explicit function of time since last spike. We know the expression represents a constant-coefficient LTI dynamical system[296] with two state variables, we'll solve its evolution over time and match parameters with the closed-form expression.

```
import numpy as np; import scipy
from matplotlib import pyplot as plt
tau_m=20; #membrane time constant
tau_s=5; #synaptic time rise epsp
eps0=20; #scaling constant epsp

t = np.linspace(0, 50, 500)
epsp = (eps0/(tau_m - tau_s))*(np.exp(-t/tau_m) - np.exp(-t/tau_s))

a = 1/tau_m
b = 1/tau_s
k = 1
dynmat = np.array([[-a, k],[0, -b]])
# Solution of above system is: x[0] = c1*exp(-at) + c2*k*(exp(-at) - exp(-bt))/(b -
↪ a); x[1] = c2*exp(bt)
# https://www.wolframalpha.com/input?i=solve+%28v%27%28t%29+%3D+kx+-av%2C++x%27%28t
↪%29+%3D+-bx%29
# then epsp(0) = 0 => c1 = 0 and c2*k/(b-a) = e0/(tau_m - tau_s) => c2*k = e0*a*b
c = np.array([0,eps0*a*b])
eps0_corrected = eps0*a*b # to inject cells at the secondary state variable with
# Let's compare with the original solution
y = np.array([scipy.linalg.expm(dynmat * ti) @ c for ti in t])
plt.figure(figsize=[10,4]);
plt.plot(t, epsp); plt.plot(t,y[:,0], '--');
plt.legend(['Original','LTI']);
```

[296] https://en.wikipedia.org/wiki/Linear_time-invariant_system



Let's convert the models for the place and action cells from the original MATLAB code. Note that both the place cells and the action cells fire *stochastically* as a function of stimulation level:

```
## Action neurons - neuron model

# rho0=60*10**(-3); #scaling rate
# theta=16; #threshold
# delta_u=2; #escape noise
cell_defs = '''
<ComponentType name="DiscreteRandomishSpiker" extends="baseCell">
    <EventPort name="spike" direction="out"/>
    <Parameter name="rho_peak" dimension="per_time" description="Peak firing rate␣
↪when mouse is right on the PC"/>
    <Parameter name="dt" dimension="time" description="Time step of the discrete␣
↪update, TODO make more continuous and smooth"/>

    <Parameter name="pc_x" dimension="none" description="Horizontal component of␣
↪this position cell's location"/>
    <Parameter name="pc_y" dimension="none" description="Vertical   component of␣
↪this position cell's location"/>
    <Parameter name="pc_sigma" dimension="none" description="Spatial constant of␣
↪this position cell's sensitivity"/>

    <VariableRequirement name="mouse_x" dimension="none" description="Horizontal␣
↪component of agent's location"/>
    <VariableRequirement name="mouse_y" dimension="none" description="Vertical  ␣
↪component of agent's location"/>
    <Dynamics>
        <StateVariable name="lastUpdate" dimension="time"/>
        <DerivedVariable name="distance" dimension="none" value="sqrt((mouse_x-pc_
↪x)^2+(mouse_y-pc_y)^2)" description="Distance of mouse from this PC"/>
        <DerivedVariable name="rho" dimension="per_time" value="rho_peak*exp(-
↪((distance/pc_sigma)^2))"/>

        <OnCondition test="(1 == 1) .and. (random(1) .lt. (rho*dt))"><!-- TODO␣
↪update explicitly and also stop when reward is reached -->
                    <EventOut port="spike"/>
            </OnCondition>
    </Dynamics>
</ComponentType>
<ComponentType name="SrmExpTwoCell" extends="baseCellMembPotCap"> <!-- not really␣
↪cap but since baseSynapse exposes current... -->
```

(continues on next page)

(continued from previous page)

```
    <Parameter name="v_rho_base" dimension="voltage" desc="Voltage midpoint for␣
↪base spiking activity"/>
    <Parameter name="rho_base" dimension="per_time" desc="Scale of base spiking␣
↪activity"/>
    <Parameter name="rho_scale" dimension="voltage" desc="Scale of exponential␣
↪spiking activity growth"/>
    <Parameter name="dt" dimension="time" description="Time step of the discrete␣
↪update, TODO make more continuous and smooth"/>

    <Parameter name="chi" dimension="voltage" description="Reset potential"/>
    <Parameter name="tau_m" dimension="time" description="Decay time constant for v
↪"/>
    <Parameter name="tau_s" dimension="time" description="Decay time constant for I
↪"/>
    <Parameter name="tau_gamma" dimension="time" description="Rise  time constant␣
↪for rho"/>
    <Parameter name="v_gamma" dimension="time" description="Decay time constant␣
↪for rho"/>
    <Constant name="msec" dimension="time" value="1 msec"/>

    <Dynamics>
            <StateVariable name="lastSpikeTime" dimension="time" exposure=
↪"lastSpikeTime"/>
        <StateVariable name="v" dimension="voltage" exposure="v"/>
        <StateVariable name="I" dimension="voltage" description="Exp decaying␣
↪current stim, modelled as voltage bc this is a tau cell"/>
        <StateVariable name="spike_count" dimension="none"/>

        <StateVariable name="rho_decay" dimension="none"/> <!-- TODO express rho␣
↪as one aggregate state variable and one latent -->
        <StateVariable name="rho_rise"  dimension="none"/>

        <DerivedVariable name="rho_tilda" dimension="per_time" value="rho_
↪base*exp((v - v_rho_base)/rho_scale)" description="Instantaneous firing rate"/>

            <TimeDerivative variable="v" value="(-v / tau_m) + I/msec" />
            <TimeDerivative variable="I" value="(-I) / tau_s" />

            <TimeDerivative variable="rho_decay" value="-rho_decay / tau_gamma" />
            <TimeDerivative variable="rho_rise" value="-rho_rise / v_gamma" />
        <DerivedVariable name="rho_smooth" dimension="none" value="rho_decay-rho_
↪rise" description="Smoothened firing rate"/>


        <OnStart> <!-- Start at resting state -->
            <StateAssignment variable="v" value="0*chi"/> <!-- TODO or zero? -->
            <StateAssignment variable="lastSpikeTime" value="t - tau_m*100" />
            <StateAssignment variable="spike_count" value="0" />
        </OnStart>

             <OnCondition test="(1 == 1) .and. (random(1) .lt. (rho_tilda*dt))"><!-
↪- TODO stop when reward is reached -->
                    <EventOut port="spike"/>
            <StateAssignment variable="spike_count" value="spike_count+1" />
                    <StateAssignment variable="lastSpikeTime" value="t" /><!--␣
↪Refer to Canc in original code as to what gets reset-->
                    <StateAssignment variable="v" value="chi" />
                    <StateAssignment variable="I" value="0*I" />
            <StateAssignment variable="rho_decay" value="rho_decay + 1" />
            <StateAssignment variable="rho_rise"  value="rho_rise  + 1" />
            </OnCondition>
```

(continues on next page)

(continued from previous page)

```
    </Dynamics>
</ComponentType>

<!-- NB: V_b may be overridden with EdenCustomSetup -->
<DiscreteRandomishSpiker id="PointCell" rho_peak="0.4 per_ms" dt="1 ms" pc_x="0"␣
↪pc_y="0" pc_sigma="0.4" />
<SrmExpTwoCell id="ActionNeuron" dt="1 ms" v_rho_base="16 mV" rho_base="60 Hz" rho_
↪scale="2 mV" chi="-5mV"
tau_m="20 ms" tau_s="5 ms" tau_gamma="50 ms" v_gamma="20 ms" C="0.2nF" />
'''
```

Note that some variables here like `rho` should be dimensional; figuring out the correct units for each parameter here is left as practice for the reader.

Let's port the synapse models, plastic for input-to-output and static for lateral inhibition. Since the spike pathway is not covered by NeuroML for post-synapses and the firing mechanism is non-standard, we'll have to detect spikes by adding spike counter to the cells and peeking into them (see also above for related additions).

```
## feed-forward synaptic plasticity parameters
ACh_flag=1; # cholinergic depression (1=+ACh; 0=-ACh)
eta_ACh = 1e-3*2; # learning rate acetylcholine

A_pre_post=1;   #amplitude pre-post window
A_post_pre=1;   #amplitude post-pre window
tau_pre_post= 10/2;   #time constant pre-post window. NOTE: altered from the␣
↪original to correct for different numerical method.
tau_post_pre= 10;   #time constant post-pre window
tau_e= 2e3; #time constant eligibility trace

eta_DA=0.01; #learning rate eligibility trace HERE

w_max=3*1; #upper bound feed-forward weights
w_min=1; #lower bound feed-forward weights
syns_defs = f'''
<ComponentType name="ExpCurrLtiSyn" extends="baseSynapse"
    description="An exponential-current synapse with the quirk that current is␣
↪modelled INSIDE the cell. TODO ">
    <Constant name="AMP" dimension="current" value="1 A"/>
    <Constant name="mV" dimension="voltage" value="1 mV"/>
    <Parameter name="wee" dimension="voltage" description="Weight really"/> <!--␣
↪TODO access the weight of the synapse  as defined on the projection, somehow -->
    <!-- <EventPort name="in" direction="in"/> -->
    <WritableRequirement name="I" dimension="voltage"/>
    <Dynamics>
            <DerivedVariable name="i" exposure="i" dimension="current" value="0 *␣
↪AMP" />
            <OnEvent port="in">
                    <StateAssignment variable="I" value="I+wee*{eps0_corrected*1}"␣
↪/> <!-- TODO dimension check! -->
            </OnEvent>
    </Dynamics>
</ComponentType> <!-- TODO updating., init weight, etc. -->
<ComponentType name="ExpCurrLtiSyn_Stdp" extends="baseSynapse"
    description="An exponential-current synapse with the quirk that current is␣
↪modelled INSIDE the cell. TODO ">

    <Constant name="AMP" dimension="current" value="1 A"/>
    <Constant name="w_max" dimension="none" value="{w_max}"/>
    <Constant name="w_min" dimension="none" value="{w_min}"/>
    <Constant name="ACh_flag" dimension="none" value="{ACh_flag}"/>
    <Constant name="mV" dimension="voltage" value="1 mV"/> <!-- TODO access the␣
```

(continues on next page)

(continued from previous page)

```
↪weight of the synapse as defined on the projection, somehow -->

    <Parameter name="w_init" dimension="none"/>

    <Parameter name="A_pre_post" dimension="none"/>
    <Parameter name="A_post_pre" dimension="none"/>
    <Parameter name="tau_pre_post" dimension="time"/>
    <Parameter name="tau_post_pre" dimension="time"/>
    <Parameter name="tau_e" dimension="time"/>

    <Parameter name="eta_ACh" dimension="none"/>
    <Parameter name="eta_DA" dimension="none"/>

    <VariableRequirement name="current_trial" dimension="none"/>
    <VariableRequirement name="reward_found" dimension="none"/>
    <WritableRequirement name="I" dimension="voltage"/>
    <WritableRequirement name="spike_count" dimension="none"/> <!-- TODO this does␣
↪not have to be Writable if Requirements can be resolved for the particular cell-
↪synapse pairs (as done for WritableRequirement) -->
    <Dynamics>

        <StateVariable name="w"  dimension="none"/>
        <StateVariable name="e"  dimension="none"/>
        <StateVariable name="w_old"  dimension="none"/>
        <StateVariable name="conv_pre"  dimension="none"/>
        <StateVariable name="conv_post"  dimension="none"/>
        <StateVariable name="last_trial"  dimension="none"/>
        <StateVariable name="last_post_spike_count"  dimension="none"/>
        <StateVariable name="register_reward"  dimension="none"/>

            <DerivedVariable name="i" exposure="i" dimension="current" value="0 *␣
↪AMP"/>
        <TimeDerivative variable="conv_pre"  value="-conv_pre /tau_pre_post"/>
        <TimeDerivative variable="conv_post" value="-conv_post/tau_post_pre"/>
        <TimeDerivative variable="e" value="-e/tau_e"/>

        <OnCondition test="w .lt. w_min"><StateAssignment variable="w" value="w_min
↪"/></OnCondition>
        <OnCondition test="w .gt. w_max"><StateAssignment variable="w" value="w_max
↪"/></OnCondition>
        <OnCondition test="(current_trial != last_trial)">
            <StateAssignment variable="w" value="w * (1-reward_found) + (w_old*eta_
↪DA*e) * register_reward"/>

            <StateAssignment variable="conv_pre" value="0" />
            <StateAssignment variable="conv_post" value="0" />
            <StateAssignment variable="e" value="0" />

                    <StateAssignment variable="last_trial" value="current_trial" />
                    <StateAssignment variable="w_old" value="w" />

            <StateAssignment variable="register_reward" value="0"/>
        </OnCondition>

        <OnCondition test="reward_found > 0">
            <StateAssignment variable="register_reward" value="reward_found"/>
        </OnCondition>
        <OnCondition test="last_post_spike_count != spike_count">
            <StateAssignment variable="conv_post" value="conv_post+A_post_pre"/>
            <StateAssignment variable="e" value="e+conv_pre"/>
            <StateAssignment variable="w" value="w-eta_ACh*ACh_flag*conv_pre"/>
```

(continues on next page)

(continued from previous page)

```
            <StateAssignment variable="last_post_spike_count" value="spike_count"/>
        </OnCondition>
            <OnEvent port="in">
                    <StateAssignment variable="I" value="I+w*{eps0_corrected*1}*mV
↪"/>
                    <StateAssignment variable="conv_pre" value="conv_pre +A_pre_
↪post"/>
            <StateAssignment variable="e" value="e+conv_post"/>
            <StateAssignment variable="w" value="w-eta_ACh*ACh_flag*conv_post"/>
            </OnEvent>
        <OnStart>
            <StateAssignment variable="w" value="w_init"/>
            <StateAssignment variable="last_trial" value="0"/>
            <StateAssignment variable="last_post_spike_count" value="0"/>
            <StateAssignment variable="register_reward" value="0"/>
        </OnStart>
    </Dynamics>
</ComponentType>
<ExpCurrLtiSyn id="LateralSynapse" wee="0 mV"/>
<ExpCurrLtiSyn_Stdp id="FeedforwardSynapse" w_init="2"
A_pre_post="{A_pre_post}" A_post_pre="{A_post_pre}"
tau_pre_post="{tau_pre_post} ms" tau_post_pre="{tau_post_pre} ms"
tau_e="{tau_e} ms" eta_ACh="{eta_ACh}" eta_DA="{eta_DA}"/>
'''
```

Let's calculate the parameters which change among place cells, action cells, and connections thereof:

```
## Place cells
space_pc = 0.4 #place cells separation distance
bounds_x = [-2, +2] #bounds open field, x axis
bounds_y = [-2, +2] #bounds open field, y axis
x_pc = np.linspace(*bounds_x, int(np.diff(bounds_x)[0]/space_pc +1)) #place cells␣
↪on axis x
n_x = len(x_pc); #nr of place cells on axis x
y_pc= np.linspace(*bounds_y, int(np.diff(bounds_y)[0]/space_pc +1))  #place cells␣
↪on axis y
n_y = len(y_pc); #nr of place cells on axis y

#create grid of place cells
grid_x, grid_y = [z.ravel() for z in np.meshgrid(x_pc, y_pc)]
N_pc=len(grid_x); #number of place cells
```

```
## Action neurons - parameters
N_action=40; #number action neurons
theta_actor = 2*np.pi*np.arange(N_action)/N_action; #angles actions
action_dirs = np.array([np.sin(theta_actor), np.cos(theta_actor)]).T

#action selection
#winner-take-all weights
w_minus = -300;
w_plus = 100;

psi = 20; #the higher, the more narrow the range of excitation
diff_theta = np.tile(theta_actor, (N_action,1)) - np.tile(theta_actor, (N_action,
↪1)).T
f = np.exp(psi*np.cos(diff_theta)); #lateral connectivity function
f -= np.diag(np.diag(f))
normalised = np.sum(f[0]) # NB all the rows/cols shoudl sum up to the same
w_lateral = (w_minus/N_action+w_plus*f/normalised); #lateral connectivity action␣
↪neurons
# print(w_lateral[0])
```

(continues on next page)

(continued from previous page)

```
lateral_conns =  [(i,j,w_lateral[i,j]) for i in range(N_action) for j in range(N_
↪action)]
# TODO check if diag is ignored...
```

```
## delete actions that lead out of the maze
# find index place cells that lie on the walls
sides = [
    np.where(grid_y == bounds_y[0])[0], # bottom wall, y=-2
    np.where(grid_y == bounds_y[1])[0], # top    wall, y=+2
    np.where(grid_x == bounds_x[0])[0], # left   wall, x=-2
    np.where(grid_x == bounds_x[1])[0], # right  wall, x=+2
]
# store index of actions forbidden from each side with trig
ε = 1e-6 # for tolerance...
forbidden_actions = [
    np.where(action_dirs[:,1] < -ε)[0], # actions that point south - theta in (180,
↪ 360) degrees approx
    np.where(action_dirs[:,1] > +ε)[0], # actions that point north - theta in (  0,
↪ 180) degrees approx
    np.where(action_dirs[:,0] < -ε)[0] ,# actions that point eastt - theta in (-90,
↪  90) degrees approx
    np.where(action_dirs[:,0] > +ε)[0], # actions that point westt - theta in ( 90,
↪ 270) degrees approx
]
# kill connections between place cells on the walls and forbidden actions
w_walls = np.ones([N_action, N_pc]);
for actt, sidd in zip(forbidden_actions, sides):
    for a in actt: w_walls[[a]*len(sidd), sidd] = 0

# show for debug
# plt.scatter(*action_dirs.T, s=0)#*range(N_action)) # for autoscale
# for i,(x,y) in enumerate(action_dirs):
#     txtt = ''.join([txt for actt, txt in zip(forbidden_actions, 'SNEW') if i in␣
↪actt])
#     plt.text(x, y, f'{i}{txtt}', ha='center', va='center',c='r')
# print(w_walls[15].reshape([11,-1]), grid_y.reshape([11,-1]), sep='\n')

ff_conns = [(i,j) for i in range(N_pc) for j in range(N_action) if w_walls[j][i] >␣
↪0]
```

And let's put it all together. We'll expose one input stream and output stream to interact with the physical environment:

```
tabline = '\n    ' # annoyingly enough, \ may not exist at all in a f string, not␣
↪even in brackets. Fixed in Python 3.12+

nml_file=f'''
<neuroml>

{cell_defs}
{syns_defs}

<!-- Here comes the network -->
<network id="Net">
    <!-- Add a population with a single cell -->
    <population id="PointCells" component="PointCell" size="{N_pc}"/>
    <population id="ActioCells" component="ActionNeuron" size="{N_action}"/>
    <projection id="Feedforward" presynapticPopulation="PointCells"␣
↪postsynapticPopulation="ActioCells" synapse="FeedforwardSynapse">
        {tabline.join([ f'<connection id="{ii}" preCellId="{pre}" postCellId="
↪{post}"/>' for ii, (pre,post) in enumerate(ff_conns)])}
    </projection>
```

(continues on next page)

(continued from previous page)

```
    <projection id="Lateral" presynapticPopulation="ActioCells"␣
↪postsynapticPopulation="ActioCells" synapse="LateralSynapse">
        {tabline.join([ f'<connection id="{ii}" preCellId="{pre}" postCellId="
↪{post}"/>' for ii, (pre,post,w) in enumerate(lateral_conns)])}
    </projection>

    <!-- Add an input stream ! -->
    <EdenTimeSeriesReader id="Env"  href="file://EnvironmentToSnn.pipe" format=
↪"ascii_v0" instances="1" >
        <InputColumn id="mouse_x" dimension="none"/>
        <InputColumn id="mouse_y" dimension="none"/>
        <InputColumn id="trial_no" dimension="none"/>
        <InputColumn id="reward_found" dimension="none"/>
    </EdenTimeSeriesReader>
</network>
<Simulation id="MySim" length="200 s" step="1 ms" seed="9" target="Net"><!-- NOTE:␣
↪leave the seed value unset to see varying behaviour -->
    <!--
    <EdenOutputFile id="ActioIntern" href="file://ac.gen.txt" format="ascii_v0"␣
↪sampling_interval="1 msec">
        {tabline.join([ f'<OutputColumn id="v_{i}"  quantity="ActioCells[{i}]/v"␣
↪output_units="mV"/>' for i in range(N_action)])}
        {tabline.join([ f'<OutputColumn id="I_{i}"  quantity="ActioCells[{i}]/I"␣
↪output_units="mV"/>' for i in range(N_action)])}
    </EdenOutputFile>
    <EventOutputFile id="SpikesExc" fileName="spikes_pc.gen.txt" format="TIME_ID">
        {tabline.join([ f'<EventSelection id="{i}"  select="PointCells[{i}]"␣
↪eventPort="spike"/>' for i in range(N_pc)])}
    </EventOutputFile>
    <EventOutputFile id="SpikesInh" fileName="spikes_ac.gen.txt" format="TIME_ID">
        {tabline.join([ f'<EventSelection id="{i}"  select="ActioCells[{i}]"␣
↪eventPort="spike"/>' for i in range(N_action)])}
    </EventOutputFile>
    <EdenOutputFile id="Synnnn" href="file://synnnn.txt" format="ascii_v0"␣
↪sampling_interval="1 msec">
        {tabline.join([ f'<OutputColumn id="decay_{i}"  quantity="Feedforward[{i}]/
↪post/e"/>' for i, (pre,post) in enumerate(ff_conns) if pre==60 and post in␣
↪list(range(1)) ])}
        {tabline.join([ f'<OutputColumn id="deca_{i}"  quantity="Feedforward[{i}]/
↪post/conv_pre"/>' for i, (pre,post) in enumerate(ff_conns) if pre==60 and post␣
↪in list(range(1)) ])}
        {tabline.join([ f'<OutputColumn id="decy_{i}"  quantity="Feedforward[{i}]/
↪post/conv_post"/>' for i, (pre,post) in enumerate(ff_conns) if pre==60 and post␣
↪in list(range(1)) ])}
    </EdenOutputFile>
     -->
    <EdenOutputFile id="MyOutSynsRec" href="file://actions.gen.txt" format="ascii_
↪v0" sampling_interval="1 msec">
        {tabline.join([ f'<OutputColumn id="decay_{i}"  quantity="ActioCells[{i}]/
↪rho_decay"/>' for i in range(N_action)])}
        {tabline.join([ f'<OutputColumn id="rise_{i}"  quantity="ActioCells[{i}]/
↪rho_rise"/>' for i in range(N_action)])}
    </EdenOutputFile>
    <!-- Add an output stream ! -->
    <EdenOutputFile id="MyOutSyns" href="file://SnnToEnvironment.pipe" format=
↪"ascii_v0" sampling_interval="1 msec">
        {tabline.join([ f'<OutputColumn id="decay_{i}"  quantity="ActioCells[{i}]/
↪rho_decay"/>' for i in range(N_action)])}
        {tabline.join([ f'<OutputColumn id="rise_{i}"  quantity="ActioCells[{i}]/
↪rho_rise"/>' for i in range(N_action)])}
    </EdenOutputFile>
```

(continues on next page)

(continued from previous page)

```
    <!-- Use EdenCustomSetup ! -->
    <EdenCustomSetup filename="CustomSetup.txt"/>
</Simulation>
<Target component="MySim"/></neuroml>
'''

# {tabline.join([ f'<OutputColumn id="v_{i}"  quantity= "Inh[{i}]/v"/>' for i in␣
↪range(N_inh)])}
with open('Model.nml', 'wt') as f: f.write(nml_file)
eden_setup_lines = []
eden_setup_lines += ['set cell PointCells all all mouse_x Env[0]/mouse_x']
eden_setup_lines += ['set cell PointCells all all mouse_y Env[0]/mouse_y']
eden_setup_lines += ['set cell PointCells all all pc_x multi', 'values '+ ' '.
↪join([str(w) for w in grid_x])]
eden_setup_lines += ['set cell PointCells all all pc_y multi', 'values '+ ' '.
↪join([str(w) for w in grid_y])]
eden_setup_lines += ['set synapse Feedforward all post current_trial Env[0]/trial_
↪no']
eden_setup_lines += ['set synapse Feedforward all post reward_found Env[0]/reward_
↪found']
eden_setup_lines += ['set synapse Lateral all post wee multi mV', 'values '+ ' '.
↪join([str(1*w) for (_,_,w) in lateral_conns])]
setup_file = '\n'.join(eden_setup_lines)
with open('CustomSetup.txt', 'wt') as f: f.write(setup_file)
import eden_simulator as eden
# tra, eve = eden.runEden('Model.nml', reload_events=True, threads=1)
```

Now let's run the simulation with `runEden` … oh wait, we also need to simulate the environment! We'll need a second process running in tandem, and Unix pipes or another conduit for exchanging streaming data. For debugging purposes, though, you can still use a normal file with place-holder trajectories.

```
%%writefile EnvironmentToSnn.pipe
0.0 0 0 1 0
```

```
Writing EnvironmentToSnn.pipe
```

```
# !eden nml Model.nml
```

Onward to writing the environment simulator: it needs to receive a stream of SNN output and provide a stream of input to the SNN. Watch out for deadlocks[297]: each simulation must output trajectories covering at least *one time step ahead* of the data it received. One day the supporting machinery outside `NewTrial` and `Advance` may be provided by Eden for convenience.

```
%%writefile mechsim.py
#!/bin/python3
import sys; argv = sys.argv
import logging
logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)
import numpy as np; import json

# parameters for the simulation
# else: raise ValueError(f'option "{type}" is not valid; only "event" or
↪"trajectory" are')
input_from_snn_filename = argv[3] #'fifa'
output_to_snn_filename = argv[4] #'fifb'
output_log_filename = argv[5] if 6 <= len(argv) else 'mechsim.gen.txt'
output_report_filename = argv[6] if 7 <= len(argv) else 'mechmore.json'
```

(continues on next page)

[297] https://en.wikipedia.org/wiki/Deadlock_(computer_science)

(continued from previous page)

```
# --- constants ---
TIME_STEP = 0.001
# TODO the roundoff can add a jitter of (sim dt) as it stands, make it work␣
↪assuming dt is an integer timestep of usec...
delay = 0.002
## Task parameters
Trials = 40000; #number of trial TODO
T_max=15000; #maximum time trial
starting_position=[0,0]; #initialize position at origin (centre open field)
goals = [(+1.5,+1.5, 0.3), (-1.5,-1.5, 0.3)] # x, y, radius
dx = 0.01; #length of bouncing back from walls
bounds_x = [-2, +2] #bounds open field, x axis
bounds_y = [-2, +2] #bounds open field, y axis
N_action=40; #number action neurons
theta_actor = 2*np.pi*np.arange(N_action)/N_action; #angles actions
action_dirs = np.array([np.sin(theta_actor), np.cos(theta_actor)]).T
a0=.008/3; # actions = a0*action_dirs HERE

# logging.debug(f'ReadyA {type} {input_from_snn_filename} {output_to_snn_filename}
↪')
# buffering is 0 to overcome 'not seekable' and binary is set bc can't have␣
↪unbuffered *text* IO
# and write is not set because then the pipe would remain open (for this process␣
↪that asked) forever
# Open the pipe for reading BEFORE the one for writing, because the latter can't␣
↪be opened until the file has been opened for reading on the other side, likewise␣
↪for the other process...
# or use async IO TODO

class MechSim:
    def __init__(self,**kwargs):
        self.fl = open(output_log_filename,'wt')
        self.InitModel(**kwargs)
    def InitModel(self,**kwargs):
        self.time = 0

        self.mechmore = {
            'time_goal':np.zeros([Trials, len(goals)]),
            'time_found':np.zeros([Trials, len(goals)]),
            'trial_started':[],
        }
        self.NewTrial(0)
        self.Log()
    def Sample(self):
        return [self.pos[0], self.pos[1], self.current_trial+1, self.reward_found]
    def Log(self):
        self.fl.write(f'{self.time} {self.pos[0]} {self.pos[1]} {self.current_
↪trial} {self.reward_found} {self.current_action[0]} {self.current_action[1]}
↪{self.current_motion[0]} {self.current_motion[1]}\n')
    def NewTrial(self, trial_no):
        if trial_no >= Trials: return
        # --- state ---
        self.current_trial = trial_no
        self.trial_started = self.time
        self.mechmore['trial_started'].append(self.trial_started)
        self.pos = np.array(starting_position) #position of the agent at each␣
↪timestep
        self.current_action = [0,0]
        self.current_motion = [0,0]
        self.time_found = 0  # time of reward - initialized to 0 at the beginning␣
↪of the trial
```

(continues on next page)

(continued from previous page)

```
        self.reward_found = 0 # flag that signals when the reward is found
        self.current_goal = 0 if self.current_trial < 10 else 1 # TODO parameterise

    def Advance(self, newtime, numbers):
        advance_until = newtime + delay
        # print('adv', newtime, advance_until)
        action_smooth_decay = numbers[0:N_action]
        action_smooth_rise = numbers[N_action:N_action+N_action]
        action_smooth = np.array(action_smooth_decay) -  action_smooth_rise
        # select action
        self.current_action = a0*(action_dirs.T@action_smooth)/N_action
        # firing_rate_store(:,i) = rho_action_neurons; %store action neurons'␣
↪firing rates
        for i,(x,y,r) in enumerate(goals):
            if np.linalg.norm(self.pos-[x,y]) <= r and self.mechmore['time_goal
↪'][self.current_trial, i] == 0:
                self.mechmore['time_goal'][self.current_trial, i] = newtime
                if self.current_goal == i:
                    logging.debug(f'Success  : {newtime} {self.trial_started}
↪{self.current_goal} {i}')
                    self.reward_found = 1
                    self.time_found = newtime
                    self.mechmore['time_found'][self.current_trial] = self.time_
↪found

        ## position update
        self.current_motion = self.current_action
        max_delta = 0.01/5 # could also add a speed limiter here
        if np.linalg.norm(self.current_motion) > max_delta: self.current_motion *=␣
↪max_delta/np.linalg.norm(self.current_motion)
        self.pos = self.pos+self.current_motion;
        # check if agent is out of boundaries. If it is, bounce back in the␣
↪opposite direction
        if self.pos[0]<=bounds_x[0]: self.pos[0] = bounds_x[0]+dx
        if self.pos[1]<=bounds_y[0]: self.pos[1] = bounds_y[0]+dx
        if self.pos[0]>=bounds_x[1]: self.pos[0] = bounds_x[1]-dx
        if self.pos[1]>=bounds_y[1]: self.pos[1] = bounds_y[1]-dx

        # time when trial end is 300ms after reward is found
        ordd = self.trial_started + T_max/1000; shor=((self.time_found+.3) if self.
↪time_found > 0 else np.inf)
        t_extreme = min(self.trial_started + T_max/1000, ((self.time_found+.3) if␣
↪self.time_found > 0 else np.inf));
        if t_extreme < self.time:
            logging.debug(f'New trial: {t_extreme} {self.time} {self.current_trial}
↪ {i} {ordd} {shor} | {self.trial_started} {self.trial_started+2} {self.trial_
↪started + T_max/1000} {T_max} {T_max/1000}')
            self.NewTrial(self.current_trial+1)

        while self.time < advance_until:
        #     world.Step(TIME_STEP, 10, 10)
            self.time += TIME_STEP
            self.Log()
        # self.Log()
        # print('adt', self.time, advance_until)
        # TODO out rate management? priority queue? a callback also?
        return
    def Fini(self):
        if self.fl: self.fl.close(); self.fl = None; print('Done!') # TODO leave␣
↪some time to ensure this runs... are borken pipes always detected?
        with open(output_report_filename, 'wt') as f: json.dump({
```

(continues on next page)

(continued from previous page)

```
            k:v.tolist() if isinstance(v, np.ndarray) else v for k,v in self.
↪mechmore.items()},f)
sim = MechSim()
try:
    with open(input_from_snn_filename, 'r',buffering=1) as fi:
            logging.debug('ReadyA')
            with open(output_to_snn_filename, 'w',buffering=1) as fo:
                    logging.debug('ReadyA')
                    i = 0
                    spiking = False # TODO
                    # if spiking:
                    #       # logging.debug('waaaa!!')
                    #       fo.write(f'{delay}\n'.encode("utf-8"))
                    while fi:
                            s = fi.readline()
                            if not s:
                                    break
                            # when there is only the timestamp+'\n' it is not␣
↪split... strip it and add \n
                            # or lstrip() + re.findall(r'^\S', st)[0] to preserve␣
↪trailing whitespace
                            timestamp, sep, remainder = s.strip().partition(' ')
                            def SampSendOut():
                                    # Fetch a snapshot
                                    samp = sim.Sample()
                                    newstamp = sim.time
                                    # Make output
                                    outnums = samp

                                    # Send output
                                    newnums = ' '.join(map(str,outnums))
                                    upds = str(newstamp)+sep+newnums+'\n'#+f'A {i}!
↪ \n'
                                    # logging.debug(f'A {i} '+upds)
                                    fo.write(upds)

                            if not spiking and i == 0:
                                    SampSendOut() # we still need that first line␣
↪for eden to begin. TODO a more elegant solution :/

                            # Parse input
                            timestamp = float(timestamp)
                            numbers = list(map(float,remainder.split()))
                            # Consume input
                            sim.Advance(timestamp, numbers)

                            SampSendOut()
                            i += 1
finally:
    sim.Fini()
```

```
Overwriting mechsim.py
```

Now let's write a small script to launch both Eden and the custom environment simulator from the command line, and set up the pipes. This implementation works on Linux and maybe Mac, if you need it for Windows just ask (page 3). GNU parallel may also be convenient[298].

```
%%writefile doit.py
import os, subprocess
```

(continues on next page)

[298] https://stackoverflow.com/questions/77039265/in-bash-how-to-wait-for-one-of-two-subprocesses-to-finish-su 77039490

(continued from previous page)

```
def EnlargeYourPipe(path): #
    # for truly huge sizes cut the pipe in two and install/run buffer(1) in␣
↪between.
    fd = os.open(path, os.O_RDWR) # ideally flags should be 0x3 here for pure␣
↪ioctl but i guess pipes don't like it
    try:
        import fcntl as fc
        # 1031 = F_SETPIPE_SZ but python3.11 doesn't provide it somehow
        ret = fc.fcntl(fd, 1031, 1024*1024) # LATER autodetect from /proc/sys/fs/
↪pipe-max-size or just run windows
        # print(ret)
    finally:
        os.close(fd);fd = None
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
def run_with_pipes(cmdlines, fifo_pair_filenames):
    import subprocess; concurrent_processes = []
    try:
        for type, s_a, a_s in fifo_pair_filenames:
            # s_a, a_s = ( f'{runtime_dir}/'+x for x in (s_a, a_s))
            subprocess.call(['rm', s_a, a_s]);
            subprocess.call(['mkfifo', s_a, a_s]);
            for x in (s_a, a_s): EnlargeYourPipe(x)
            if s_a != a_s:
                p = subprocess.Popen(['python', 'mechsim.py', '0.010', type, f'{s_
↪a}', f'{a_s}'])
                concurrent_processes.append(p)
        invocations = []
        for cmdline in cmdlines:
            p = subprocess.Popen(['bash', '-c', cmdline])
            invocations.append(p)
        for p in invocations:
            p.wait()
            if p.returncode:
                print('Retcode', p.returncode, 'for ',p.args)
        if any(p.returncode != 0 for p in invocations):
            print(435,[p.returncode for p in invocations])
            raise
    finally:
        for p in concurrent_processes:
            try:
                p.wait(timeout=.5) # give some time for graceful shutdown of aux␣
↪processes
            except subprocess.TimeoutExpired: pass
        if any(p for p in concurrent_processes if p.returncode is None): print(
↪'Terminating remaining processes...')
        for p in (p for p in concurrent_processes if p.returncode is None):
            print(p)
            p.kill()
fifo_pair_filenames = [('trajectory', 'SnnToEnvironment.pipe','EnvironmentToSnn.
↪pipe'),]; threads=1
cmdlines = [f'OMP_NUM_THREADS={threads} OMP_SCHEDULE=static eden nml Model.nml >␣
↪log_eden.txt 2>&1']
run_with_pipes(cmdlines, fifo_pair_filenames)
```

```
Overwriting doit.py
```

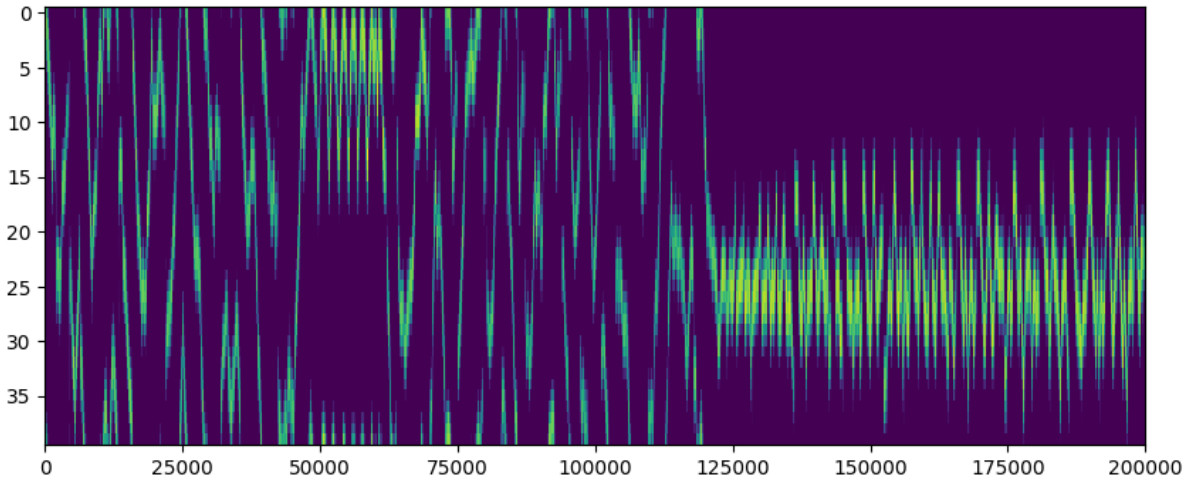
And finally get the whole system running!

```
%%time
!python3 doit.py 2>simulation_log.txt
```

```
Done!
CPU times: user 580 ms, sys: 76.2 ms, total: 657 ms
Wall time: 36.5 s
```

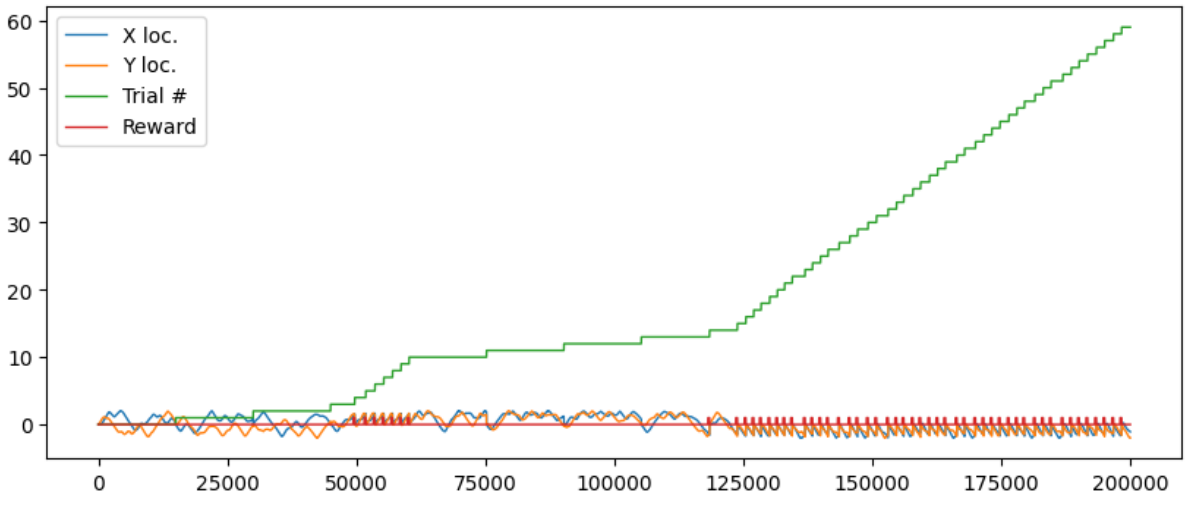
If all goes well, we can see how the output layer steered the agent. Thanks to lateral inhibition, only one direction is chosen at a time, and it changes smoothly. The membrane potential of the action cells can also prove useful to debug the layer's behaviour, especially when comparing with the original.

```
neurdata = np.loadtxt('actions.gen.txt')
actions = neurdata[:,1:1+N_action] - neurdata[:,1+N_action:1+2*N_action] #␣
↪Subtract rising from falling component
plt.figure(figsize=[10,4]);plt.imshow(actions.T,aspect='auto',interpolation='none
↪');#plt.colorbar();
```



The environment simulator wrote down the state of the physical simulation over time (see `Log(...)` above). Let's see what happened:

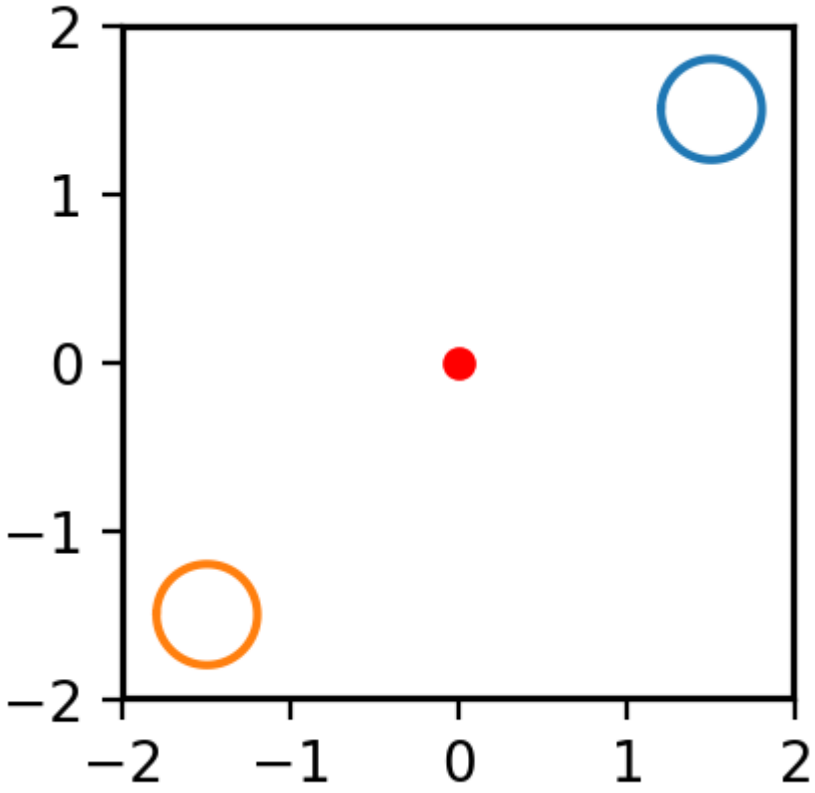
```
neurdata = np.loadtxt('mechsim.gen.txt')
labels = ['time', 'X loc.', 'Y loc.','Trial #','Reward','actx','acty','velx, vely']
plt.figure(figsize=[10,4]);plt.plot(neurdata[:,1:5],lw=1, label=labels[1:5]);plt.
↪legend();
```



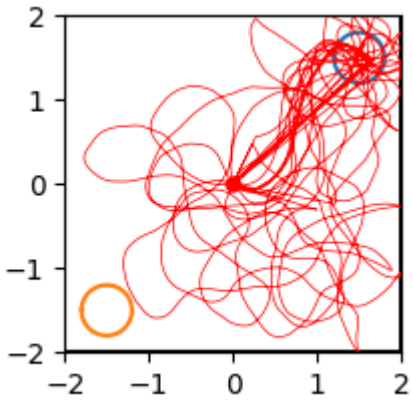
The trials add up over time, and as the SNN learns to navigate to the target they end faster in success. As per the original experiment, the reward is applied (and the new trial started) a short time after the target is reached.

Let's look at the AI agent's trajectory within its confines (a trivial sort of labyrinth, if you think about it):

```
## plotting
starting_position=[0,0]; #initialize position at origin (centre open field)
goals = [(+1.5,+1.5, 0.3), (-1.5,-1.5, 0.3)] # x, y, radius
dx = 0.01; #length of bouncing back from walls
bounds_x = [-2, +2] #bounds open field, x axis
bounds_y = [-2, +2] #bounds open field, y axis
def PlotField(ax, mouse_traj=[], current_trial=None):
    # figure('position',  [0, 0, 1000, 2000])
    thetas = np.arange(-np.pi, +np.pi, 2*np.pi/100)

    for i,(x,y,r) in enumerate(goals): # plot goals
        current_goal = (i == (1 if current_trial > 9 else 0)) if current_trial is␣
↪not None else True
        # print(f'{i}, {current_goal}')
        ax.plot(x+r*np.cos(thetas), y+r*np.sin(thetas), '-' if current_goal else '-
↪-')
    point_plot = ax.plot(*starting_position, '.r', ms=10); #plot initial starting␣
↪point

    #plot walls
    ax.set_aspect('equal', 'box')
    ax.plot([bounds_x[0],bounds_x[0]], [bounds_y[0],bounds_y[1]],'k')
    ax.plot([bounds_x[1],bounds_x[1]], [bounds_y[0],bounds_y[1]],'k')
    ax.plot([bounds_x[0],bounds_x[1]], [bounds_y[0],bounds_y[0]],'k')
    ax.plot([bounds_x[0],bounds_x[1]], [bounds_y[1],bounds_y[1]],'k')
    ax.set_xlim(*bounds_x); ax.set_ylim(*bounds_y)

    #display trajectory of the agent in each trial
    if len(mouse_traj)>0: ax.plot(*mouse_traj, 'r',lw=.5);

    if current_trial is not None: ax.set_title(f'Trial {current_trial+1}')
plt.figure(dpi=200)
plt.subplot(2,2,1)
PlotField(plt.gca())
```

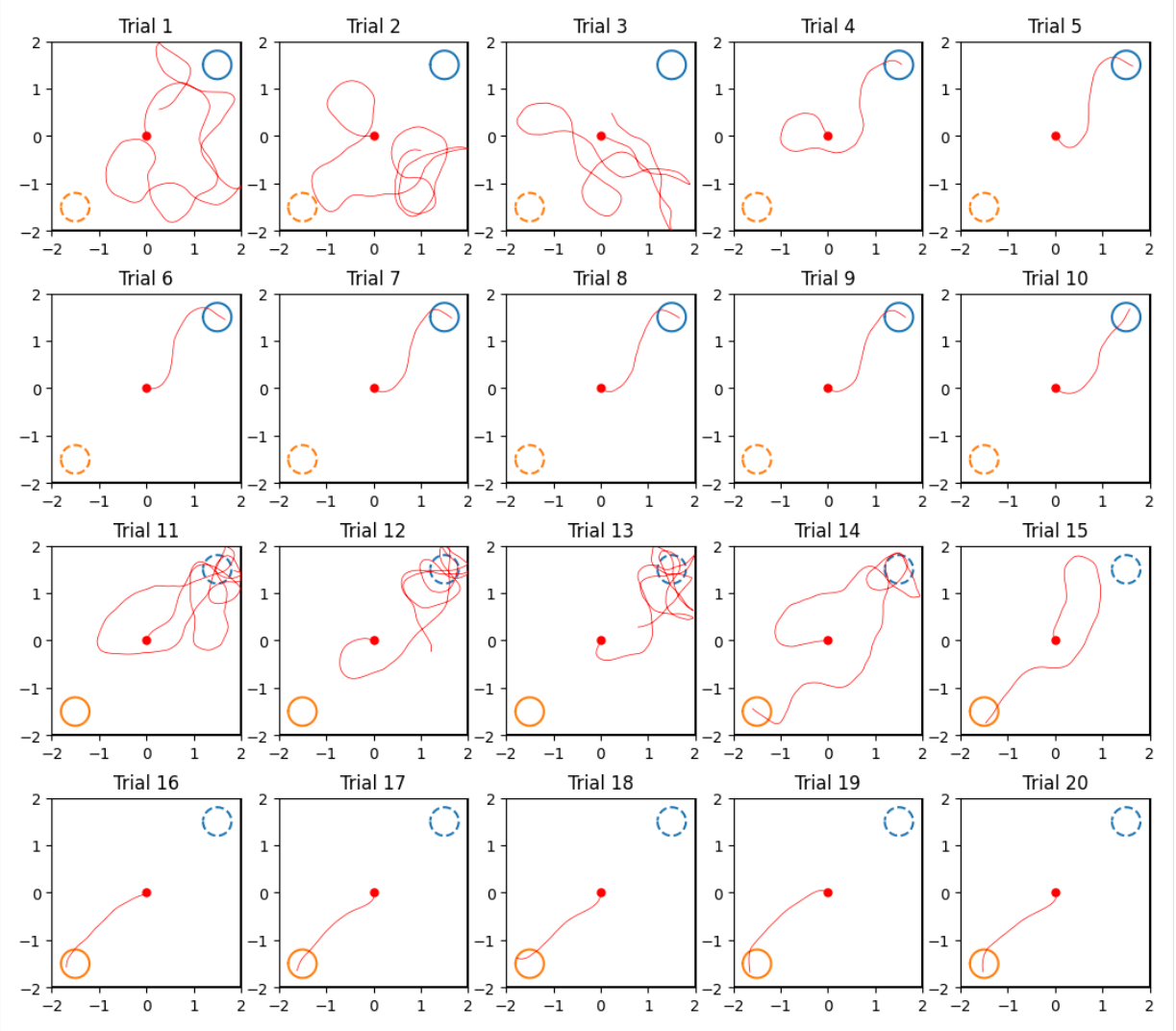



```
plt.figure();ax = plt.subplot(2,2,1)
PlotField(ax, neurdata[0:100000,1:3].T);
```



And trial by trial:

```
tracemax = len(neurdata)
on_reset       = np.all(neurdata[0:tracemax,1:3] == [0, 0], axis=1)
resets = np.r_[0,np.where(on_reset[1:] * ~on_reset[:-1])[0],tracemax]
# print(resets,len(resets))
plt.figure(figsize=[13,11.5])
for trial_no in range(20):#range(len(resets)-1):
    ax = plt.subplot(4,5,trial_no+1)
    PlotField(ax, neurdata[resets[trial_no]+1:resets[trial_no+1],1:3].T, trial_no);
plt.show()
```




```
# Or display them one by one:
# for trial_no in range(len(resets)-1):
#     plt.figure();ax = plt.subplot(2,2,1)
#     PlotField(ax, neurdata[resets[trial_no]+1:resets[trial_no+1],1:3].T, trial_
↪no);
#     plt.show()
```

Feel free to draw the separate trials as progressive shades on the same environment, and animate the evolving *policy* as a vector field of `Σ (action weights * action direction)` per place cell.

Part D

Building on top of the EDEN platform

This part is about the way that the simulation platform EDEN works internally, and how its existing components can be re-used to advance research in neural simulation.

CHAPTER

NINETEEN

SOFTWARE MAINTENANCE

This section concerns the standard EDEN distribution's codebase and build process. It is useful for both maintainers of the codebase, as well as who wish to interface with it.

19.1 Testing process

For Linux: with Docker available, run `make test`. Currently, proper regression testing is limited to Linux for want of time. A special Docker image is built and run, to bundle all necessary software and isolate the test environment from the user's own. Testing per platform can be added, if the other related software (NEURON, NeuroML tools, build tools) and their dependencies can also run. (binaries are often harder to find for non-PC CPUs, and such) A basic smoke test is done at the end of the build process, for all platforms; this is not enough but it can catch many platform specific errors. (Isolation between the build and the smoke-test environment is a work in progress for OSX, Linux, Windows ordered by severity.)

The combinations of interacting mechanisms are way too many to completely cover with tests. To keep bugs in check, make sure to maintain regular, clean-cut interfaces between code modules. Minimise code duplication, re-use elements and use strong typing to increase the likelihood of catching bugs, and the effectiveness of bug fixes. (A bit of duplicated code structure may often be better than a single routine pockmarked with conditionals; exercise judgement.)

Keep modules tight; using strictly one source file per class may be counter-productive. (See also: Java and the usefulness of inner classes.)

19.2 Build process

The primary incremental build tool should be GNU `make`. Although newer build tools like CMake and Meson are more elegant and automagical, this often backfires on custom exotic systems like e.g. supercomputers. In such cases, `make`'s direct and predictable way of issuing commands is much easier to control and adapt. Meanwhile, the older tools like autoconf/automake are not so important given the environment convergence going on and the limited complexity of this program, and besides they may also behave in magical ways.

A good example for how a Makefile should be is the Makefile[299] of the Git program.

This makefile is primarily designed for the GNU Compiler Collection, as it's good for all common platforms. Thanks to user-facing convergence, the compiler options are also largely valid for the clang compiler as well, and likely the new, Clang based Intel compiler.

If it is desired to build with Visual Studio, GNU Make can still be used (look it up if interested). The compiler flags and linkage will need a full overhaul for VC++ though, and edits to the C++ source may also be in order (We try to keep the code standards compliant, but each compiler has multiple conflicting opinions on what is valid C++).

[299] `https://github.com/git/git/blob/master/Makefile`

19.2.1 Build environment

- On Linux, builds are based on `manylinux1` for regular PCs, and `manylinux2014` for most other CPU architectures. A version bump may be needed for newer C++ versions and facilitated by the update treadmill in the userbase. For Windows and OSX, the required set of build/test tools is assembled using the `testing/<os>/download-requirements` scripts.
- On Windows, the software binaries are gathered from various sources. On OSX, they are downloaded and installed with Homebrew; check the install script to separate the home environment from Homebrew's default. (Though all binaries will have to be rebuilt in that case; consider a virtual machine for easy isolation.)
- In principle, automated testing can be set up easily. It will just take some time to set up the tools for testing, the same way it's been done for building.

19.2.2 Release binaries

Binaries are distributed in the Python wheel format, because `pip install eden-simulator` is more convenient and less error prone than manually selecting the correct arch for the target machine and unpacking. Python bindings (page 237) are of course included with the wheels.

Wheels are generated using scripts found at `testing/<OS>/build-test-all-in-one` for each operating system (where they are supposed to be built on). **NB**: only basic "smoke tests" are being run, full testing is done on Linux with `make test` for now (page 223).

Linker issues for release binaries are mitigated by the usual DLL bundling tools, `delocate` and `auditwheel` for Mac and Linux respectively. `delvewheel` can also be used for Windows, but these builds can remain static for now. (Be aware that careless, automatic DLL bundling can violate code licensing; watch out for unintended dependencies.)

References

EXAMPLES GALLERY

FREQUENTLY ASKED QUESTIONS

Here are some answered questions that may come to mind while reading this guide. Many of them have been asked by real users in the past!

- Terminology dictionary (page 230)
- Using the simulator (page 231)
  - What units are simulation results recorded in? (page 231)
  - The simulation output is too large! How can I record values every millisecond or so? (page 231)
  - The neuron potential recording is too large! How can I record only when spikes happened? (page 231)
  - How can I have the same result, every time I run my randomised simulation? (page 231)
  - How should I use EDEN on a (MPI) computer cluster? (page 232)
- NeuroML resources (page 232)
  - Where can I find NeuroML models? (page 232)
  - Which other tools can I use NeuroML with? (page 233)
- Setting up and running models (page 233)
  - What is “discretisation” or “compartments” and how does it affect my model? (page 233)
  - How do I add calcium-modulated ion channels? (page 233)
  - How do I add Ornstein-Uhlenbeck noise? (page 233)
  - How do I add *other ion*-modulated ion channels or other mechanisms (modulated by possibly other quantities)? (page 233)
  - How do I control mechanisms with a single “global” variable? (page 233)
  - How do I explicitly set some variables on *every single timestep*? (page 234)
- Writing LEMS equations (page 234)
  - How do I add *normally distributed* random variables in my LEMS equations? (page 234)
  - How do I use the sgn(x) function in expressions? (page 234)
  - What does `log` mean in equations? (page 234)
- Troubleshooting (page 234)
  - Why is my spatial cell behaving exactly as if it was a single compartment? (page 234)
  - My NeuroML model behaves very differently on EDEN compared to when it runs on e.g. NEURON via jNeuroML ! (page 235)
  - When I use < in LEMS conditions, I get XML errors! (page 235)
- What is Eden capable of? (page 235)
  - Can EDEN run Pong? (page 235)
  - Does EDEN support Blockchain? (page 235)
  - Does EDEN support Artificial Intelligence? (page 235)
  - Can EDEN run Large Language Models? (page 235)

- What does Eden *not* do, and what are the workarounds? (page 235)
  - Recording <DerivedVariable>s (page 235)
  - <Regime>s (page 236)
  - <Display> elements in <Simulation> (page 236)
  - <doubleSynapse>s and <compositeInput>s (page 236)

Terminology dictionary

Table 1: Different names for different simulators

Term	EDEN	NEURON[Page 231, 30]	GENESIS[Page 231, 301]	Arbor[Page 231, 302]	BRIAN[Page 231,	Comments
Point neuron	<ComponentType extends="baseCell">	ARTIFICIAL_CELL		What's not a *cable cell*	What's not a Spatial-Neuron	A neuron model with the morphology largely simplified out.
Multi-compart cell	<cell>	what's not an ARTIFICIAL_CELL	All of them really	cable cell	Spatial-Neuron	A neuron model of multiple finely detailed *finite elements*.
finite element	compartment	compartment, segment	compartment	control volume (CV)	compartment	A piece of neuron whose state is assumed to be uniform.
Tracing point	A 's <proximal> and <distal>	pt3d	Same as compartment	point or segment for a pair	Same as compartment	A (x,y,z,d) data point for a traced neurite.
Neurite span	sao8649213	section		branch	section	A *distinct* unbranched section of neuritic cable.
Arbitrary dependence	<VariableReference> (page 107)	POINTER	addmsg		linked	An continuous-time dependency of a variable on *any* other variable.
Aggregate flux	Multiflux (page 121)	WRITE	addmsg	WRITE	summed	*Distinct* flows of chemical species, supplied by various mechanisms.

continues on next page

Table 1 – continued from previous page

Term	EDEN	NEURON[300]	GENESIS[301]	Arbor[302]	BRIAN[303]	Comments
Spike train	<spikeArray>[304], <EdenEventSetReader> (page 118)	VecStim[305]	TBD	Spiking cell	`SpikeGenerator-Group` and others	A disembodied source of spikes
Time series	<EdenTimeSeriesReader> (page 116)	`Vector.play()`[306]	TBD		`value_name t)`	A disembodied evolving quantity

Using the simulator

What units are simulation results recorded in?

The unspoken NeuroML convention is to use SI-derived units[307] that are products of the seven base SI units. Keep in mind that that the typical real-life unit for concentration is mol/L whereas the SI unit is mol/m^3!

If you prefer quantities to be recorded in specific units, refer to Extended input-output options (page 109).

The simulation output is too large! How can I record values every millisecond or so?

Using `<EdenOutputFile>` (page 109).

The neuron potential recording is too large! How can I record only when spikes happened?

Using `<EventOutputFile>`[308] or `<EdenEventOutputFile>` (page 109).

How can I have the same result, every time I run my randomised simulation?

EDEN does have a provision for that; set the `seed` attribute in your `<EdenOutputFile>` (page 23) tag. The generated random numbers should be the same at least per model contents, and program version.

But this is *not* the end of the story! There are ways for a small perturbation to slip through and be amplified (in principle) indefinitely by chaotic systems, like neural networks tend to be.

The ways that perturbations can slip in the numbers are allowed by the finite accuracy in out computers and the resulting round-off that takes place during calculations, and caused by carrying out the calculations differently at times.

[300] https://neuron.yale.edu
[301] http://genesis-sim.org/
[302] https://arbor-sim.org
[303] https://briansimulator.org/
[304] https://docs.neuroml.org/Userdocs/Schemas/Inputs.html#schema-spikearray
[305] https://www.neuron.yale.edu/phpBB/viewtopic.php?f=28&t=2117
[306] https://www.neuronsimulator.org/en/latest/guide/bio_faq.html#how-do-i-simulate-a-current-clamp-with-non-p
[307] https://en.wikipedia.org/wiki/Fundamental_unit
[308] https://docs.neuroml.org/Userdocs/QuantitiesAndRecording.html#userdocs-quantitiesandrecording-events

These are some factors that can lead to calculations being run differently:

- Different (versions of) the computers' operating systems and code generators[309];
- The need for speed that pushes us to do the processing in whichever way it is convenient on each machine;
- A different order of summation, that happens they are split over multiple cores in a computer;
- The distribution of model parts between computers in a multi-machine simulation.

To study the issue and potential causes further, refer to a web search on "floating point determinism" (for example, "Determinism and Reproducibility in Large-Scale HPC Systems[310]").

In conclusion, measures to attempt deterministic simulation may get the output to be close enough in between simulation runs, but this is regrettably not guaranteed within practical limits.

The good news is that a neural network's function should not rely on one specific pick of random samples, otherwise the random numbers wouldn't be random (being part of the model themselves). It follows that if a model's correct behaviour appears only for a small set of random `seeds`, then that is the model's fault.

If some random number sequences are of critical importance (to replicate findings exactly), consider writing down the exact waveforms and then playing them back with EDEN's `<EdenTimeSeriesReader>` extension (page 116) to NeuroML, in place of the places where `random()` would be used.

How should I use EDEN on a (MPI) computer cluster?

Just use the same command line[311] as usual on all processes launched, targeting the NeuroML file to run. Refer to Building for MPI[312] for how to build from source on a cluster.

NeuroML resources

Where can I find NeuroML models?

Most published NeuroML models can be found in the following places:

- NeuroML-DB[313] for a systematised database of neuron and mechanism models, and a few network models;
- Open Source Brain[314], for more diverse networks and component models in NeuroML, and also other technologies;
- NeuroML models in the ModelDB[315]. Browse also the entire ModelDB for lots of models that could be ported to NeuroML.

See also the NeuroML guide[316] on the subject.

[309] `https://en.wikipedia.org/wiki/Compiler_(computing)`
[310] `https://wodet.cs.washington.edu/wp-content/uploads/2013/03/wodet2013-final12.pdf`
[311] `https://eden-simulator.org/repo#from-the-command-line`
[312] `https://eden-simulator.org/repo#building-for-mpi`
[313] `http://neuroml-db.org`
[314] `https://v1.opensourcebrain.org/projects`
[315] `https://modeldb.science/modellist/154351?all_simu=true`
[316] `https://docs.neuroml.org/Userdocs/FindingNeuroMLModels.html`

Which other tools can I use NeuroML with?

The official NeuroML guide provides an up-to-date list of these tools:

- The official NeuroML tooling[317]
- Third-party tools that can work with NeuroML[318]

Setting up and running models

What is "discretisation" or "compartments" and how does it affect my model?

Refer to the "Discretisation" section (page 40) and try the 'simple cable' exercise (page 52) exercise, on the chapter on spatially-modelled neurons. What usually happens is:

- Having too large compartments shows as *excessive damping* (or *underestimation*) with regard to spatially-sensitive effects: namely spike propagation, and the contribution of point processes like synapses and probes to the neuron's state.
- On the other hand, having too small compartments may stress the numerical methods, over- or undershooting due to *numerical round-off* when the differences become tiny.

Generally, simulators don't like handling multiple timescales; keep things as rough as they still let your model work as designed.

How do I add calcium-modulated ion channels?

With a custom LEMS component (page 80).

How do I add Ornstein-Uhlenbeck noise?

With a custom LEMS component (page 76).

How do I add *other ion*-modulated ion channels or other mechanisms (modulated by possibly other quantities)?

With EDEN's LEMS extension `<VariableReference>` (page 107).

How do I control mechanisms with a single "global" variable?

With `<VariableReference>`s (page 107) that point to the same single quantity (that can be on an abstract cell or wherever).

[317] https://docs.neuroml.org/Userdocs/Software/Software.html
[318] https://docs.neuroml.org/Userdocs/Software/SupportingTools.html

How do I explicitly set some variables on *every single timestep*?

By keeping a state variable `timeSince` and adding a tag like:

```
<OnCondition test="t >= timeSince + (one timestep)">
    ... update things ...
    <StateAssignment variable="timeSince" value="t"/>
</OnCondition>
```

See also the the OU noise example (page 76) which updates the state variable `i` following a random walk.

EDEN and many other NeuroML-capable simulators also run `<OnCondition test="(something unconditional)">` on every timestep which saves one state variable, but it's not part of the specification. Expect discrete updates to become part of the specification some day.

Writing LEMS equations

How do I add *normally distributed* random variables in my LEMS equations?

A simple way is the Box-Muller transform[319]. Refer to the OU noise example (page 76) for details.

How do I use the sgn(x)[320] function in expressions?

Use $2 * H(x) - 1$ instead, and request the feature[321] if it would help a lot to have it.

What does `log` mean in equations?

It stands for the *natural logarithm* (and is an alias for `ln`), so that $log(e) = 1$. If you prefer the decimal logarithm, `log10` can be used instead.

Troubleshooting

Why is my spatial cell behaving exactly as if it was a single compartment?

Either:

- the cytoplasmic `<resistivity> value` is too low;
- the lengths of neurites in the `Morphology` are too small;
- the membrane's `<specificCapacitance>` is too low;
- the channels' conductance is too high;
- or (a cable in) the whole neuron was specified to be a single compartment with the "unbranched section" (page 40) directive, inadvertently.

[319] https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform
[320] https://en.wikipedia.org/wiki/Sign_function
[321] https://github.com/NeuroML/NeuroML2/issues

My NeuroML model behaves very differently on EDEN compared to when it runs on e.g. NEURON via jNeuroML !

Please contact us (page 3) to track down the issue and provide fixes and options.

When I use < in LEMS conditions, I get XML errors!

Unfortunately this character annoys XML. Use the FORTRAN spelling (page 63) `.lt.` (less than) or `.le.` (less/equal) instead, or escape the character with `<` (ugly).

What is Eden capable of?

Can EDEN run Pong?

Yes! Check out the example (page 173).

Does EDEN support Blockchain?

In NeuroML, anatomically detailed cells (page 33) comprise spans of neurite. Each such span can be considered as a chain of interlinked "blocks", also called "compartments" in our terminology. In that sense, EDEN can simulate multiple blockchains within even a single neuron.

Does EDEN support Artificial Intelligence?

Artificial Intelligence (A.I.) may be implemented in various ways; one that's especially bio-inspired is *spiking neural networks*, whose activity can be simulated *in silico* using EDEN. Since EDEN is primarily a scientific tool, it is up to the modeller to design effective A.I. architectures.

Can EDEN run Large Language Models?

As long they can be described as spiking neural networks, they can run. The attention mechanism is not quite SNN-like, but it may be implementable in LEMS. An example is in development.

What does Eden *not* do, and what are the workarounds?

Recording `<DerivedVariable>`s

Instead, add a shadow `<StateVariable>` for each `<DerivedVariable>`; then for each shadow state variable, add a `<OnCondition test="1 == 1">` with a `<StateAssignment>` of the shadow variable to the `DerivedVariable`. (Recording `<DerivedVariable>`s will be supported soon.)

<Regime>s

Instead, a <StateVariable> can be used to indicate the type of the regime, with the affected <DerivedVariable>s becoming <ConditionalDerivedVariable>s.

<Display> elements in <Simulation>

Instead, use <OutputFile> and display with your preferred graphing system, as shown in the guide.

<doubleSynapse>s and <compositeInput>s

Instead, merge the involved <ComponentType>s into a single LEMS component that includes the dynamics of all parts, and exposes their summed influence.

- eden-simulator (page 237)
- Display API (page 238)
  - eden_simulator.display.animation (page 238)
  - eden_simulator.display.spatial (page 239)
  - eden_simulator.display.spatial.k3d (page 239)
  - eden_simulator.display.spatial.pyvista (page 241)
- Experimental API (page 241)

eden-simulator

EDEN bindings and helpers for Python.

runEden(*example_lems_file*, *, *reload_events=False*, *verbose=False*, *threads=None*, *extra_cmdline_args=None*, *full_cmdline=None*, *executable_path=None*)

Run a NeuroML2/LEMS file with EDEN.

Parameters

- **example_lems_file** (*str*) – The filename where the <*Simulation*> to run is located.
- **reload_events** (*bool, optional*) –

  Whether to return a:

  – *dict* with the recorded time series, mapping each recorded <*LemsQuantityPath*> to its sampled values over time. All samples are taken at the same points in time, represented by the additional time vector *t* in seconds.

  – or a pair of *dict* s, the first as before and the second mapping each recorded event to its times of occurrence.
- **threads** (*int, optional*) – The number of threads to run the simulation with, or *None* for automatic selection.
- **verbose** (*bool, optional*) – Add some more console output when running the simulator.
- **extra_cmdline_args** (*list[str], optional*) – Additional command line arguments to pass to EDEN. Refer to the EDEN user's manual, "Command line API" for more details.
- **full_cmdline** (*list[str], optional*) – Specify the exact *argv* to be passed to EDEN. Refer to the EDEN user's manual, "Command line API" for more details.
- **executable_path** (*str, optional*) – Select a specific executable of EDEN to run the simulation with. Useful for custom installations and uses of EDEN.

Returns

- **trajectories** (*dict[str, numpy.ndarray[float[n]]]*) – *when reload_events == False*

- **(trajectories, events)** (*tuple(dict, dict)*) – *when reload_events == True*; see *reload_events* parameter.

Notes

All values are returned in SI units, or derived thereof.

- e.g. concentration is in *mol/m³*, not *mol/L* !

Refer to https://en.wikipedia.org/wiki/SI_derived_unit for a list of such units.

Display API

eden_simulator.display.animation

GetAutoFps (*anim_axis*, *max_auto_fps=60*)

Get an appropriate update frequency for a time series.

subsample_trajectories (*time_axis_sec*, *data=[]*, *animation_speed=0.003*, *animation_frames_per_second=30*)

Subsample a set of time series for animated real-time display.

Parameters

- **time_axis_sec** (*np.array[float]*) – The timebase for the given time series, in seconds.
- **data** (*array*) – A *series* by *time* array of values. Length of *time* must be same as *time_axis_sec*.
- **animation_speed** (*float*) – The ratio of how fast the time series is played back, relative to *time_axis_sec*.
- **animation_frames_per_second** (*float*) – The desired sampling rate for the animation. The time series are subsampled but not interpolated, thus the true frame rate may vary.

Returns

- **samples_picked** (*np.array[int]*) – The samples used from the provided sequence *data*. Useful for resampling other time series similar to *data*. May be less than expected if there are not enough samples to fill the grid.
- **anim_axis** (*np.array[float]*) – Animation-time for each picked sample.
- **sampled_time_axis** (*np.array[float]*) – Original timebase for each picked sample. Equals *time_axis_sec[samples_picked]*.
- **sampled_data** (*list[list]*) – Sub-sampled *data*. Equals *data[samples_picked]*.

eden_simulator.display.spatial

Utilities for displaying solid neurons in euclidean space.

get_mesh_info(*cell_info*)

Convert the 'explain cell' format to mappings from compartments to corresponding vertices and faces of the mesh.

Parameters

cell_info (*dict*) – The information for a cell type, provided by an element of *eden_simulator.experimental.explain_cell*

Returns

(mesh_vertices, mesh_faces, mesh_comp_per_face, n_comps) – The selected mesh-specific subset of *cell_info*.

Return type

tuple

get_verts_faces_per_comp(*mesh_vertices*, *mesh_faces*, *mesh_comp_per_face*, *n_comps*)

Convert the mesh information generated by *get_mesh_info* to per-vertex and per-face mappings to/from associated compartments.

Parameters

eden_simulator.experimental.explain_cell. (*As returned by get_mesh_info and ultimately*)

Returns

- **(mapped_verts, mapped_faces, verts_per_comp, faces_per_comp)** (*tuple*)
- **mapped_verts** (*function*) – A function receiving a 2-D *numpy.array*, and returning the linear combination of its columns as per *verts_per_comp*.
- **verts_per_comp** (*scipy.spmatrix*) – A sparse matrix with the neuron mesh's vertices as rows and associated compartments as columns. TODO
- **mapped_verts** (*function*)
- **verts_per_comp** (*scipy.spmatrix*) – Similar to the previous but mapping compartments to mesh faces. TODO

eden_simulator.display.spatial.k3d

Straightforward display of neurons with [K3D](https://k3d-jupyter.org).

DeduceColormap(*colormap*)

Try to deduce a K3D colormap from the passed argument.

See also

Colormaps[322]

k3d.helpers.map_colors()[323]

IntToRgb(*hex*)

K3D style packed integer array *hex* to … x 3 *rgb* array.

[322] https://k3d-jupyter.org/reference/colormaps.html
[323] https://k3d-jupyter.org/reference/helpers.map_colors.html#k3d.helpers.map_colors

MinimizePlot(*plot*, *more_objects=[]*)

Minimise the state of a K3D plot widget.

Useful for publishing notebooks with K3D plots, when a snapshot is preferable to the fragile ipywidgets machinery and uncompressed widget_state. TODO :param plot: The plot to minimise. :type plot: k3d.Plot :param more_objects: :type more_objects: list[k3d.]

class Plot(**args: t.Any*, ***kwargs: t.Any*)

A subclass of *k3d.Plot* with augmented display capabilities.

To be used with extended elements such as *plot_neuron*, along with the usual *k3d* display elements.

__init__(**args*, ***kwargs*)

Public constructor

display(***kwargs*)

Show plot inside ipywidgets.Output().

get_snapshot(*compression_level=9*, *voxel_chunks=[]*, *additional_js_code=''*, ***kwargs*)

Produce on the Python side a HTML document with the current plot embedded.

show_html(*snapshot_type='inline'*, ***kwargs*)

Show a snapshot of the plot as an *ipynb* displayable.

Parameters

- **snapshot_type** (*str*) – The snapshot_type to use in capturing.
- **kwargs** (*dict*) – Other arguments to pass to `get_snapshot()`.

RgbToInt(*rgb*)

Convert … x 3 *rgb* array to K3D style packed *ints*.

decompress_cells(*k3d_mesh*)

Revert per-cell compression to normal mode.

plot_neuron(*cell_info*, *comp_values=(0.5, 0.5, 0.5)*, *time_axis_sec=None*, *color_map=None*, *compress_cells=True*, ***kwargs*)

Plot a neuron mesh with K3D, optionally coloured and animated. For use with *Plot*. TODO

Parameters

- **vertices** (*Nx3 float array*)
- **faces** (*Mx3 int array*)
- **comp_per_face** (*M float vector*) – The compartment that each face belongs to.
- **comp_values** – The values per compartment:
  - K x 3:
- **nah** (*ndarray of float K x C or T x K or T x K x C or K or C (or T x c?)*) – The values per compartment:
  - K x 3:
- **colormap** – . Cannot be used with RGB values.
- **time_axis_sec** (*T float array*)
- **well...** (*need time per sec as*)
- **kwargs** (*dict*) – Additional parameters to pass through to *k3d.mesh*.

Returns
mesh

Return type
k3d.mesh

eden_simulator.display.spatial.pyvista

Display helpers for [PyVista](https://pyvista.org).

get_neuron_mesh(*cell_info*)

Get a PyVista mesh, colourable per compartment.

Parameters

cell_info (*dict[str]*) – A *dict* containing *'mesh_vertices'*, *'mesh_faces'* and *'mesh_comp_per_face'* describing a neuron's 3-D mesh, as produced by by *eden_simulator.experimental.explain_cell*.

Returns

mesh – A 3D mesh which can be assigned scalars per compartment.

Return type

pyvista.UnstructuredGrid[324]

Experimental API

Additional facilities to use EDEN's experimental features from within Python.

NOTE: As the experimental features are still in design, parts of this API may change between versions!

GetLemsLocatorsForCell(*cell_info*, *compartment_ids=None*)

Generate a list of (extended) LEMS locators to access a property all over a cell.

Parameters

- **cell_info** (*dict*) – The discretisation information about the cell, as returned from *eden_simulator.experimental.explain_cell*.
- **compartment_ids** (*iterable*) – A specific list of compartments to access (by default all over the cell)

Returns

l – The list of locators, to be used in extended LEMS paths capturing the cell property.

Return type

list[str]

See also

eden_simulator.experimental.explain_cell (page 241): For obtaining cell_info. LEMS paths for cell locations (page 87) : For LEMS segment locators including *fractionAlong*. NeuroML and spatially detailed cells (page 33): To lean more about spatially detailed cells.

explain_cell(*nml_file*, *, *verbose=False*, *threads=None*, *extra_cmdline_args=None*, *executable_path=None*, *full_cmdline=None*, ***kwargs*)

Get more information about the structure and properties of physically-modelled cells.

Parameters

- **nml_file** (*str*) – The main NeuroML file (and the ones it <include>s) to consider.
- **verbose** (*bool, optional*) – Add some more console output when running the simulator.

[324] https://docs.pyvista.org/api/core/_autosummary/pyvista.UnstructuredGrid.html#pyvista.UnstructuredGrid

- **threads** (*int, optional*) – The number of threads to run processing with, or *None* for automatic selection. (Does not have an effect in this case yet.)

  The following parameters are less commonly used:

- **mesh_prism_sides_count** (*int, optional*) – A fixed number of sides for the meshes of the tubular sections comprising the neurons' morphologies. Must be 3 or more. Defaults to `5`.
- **extra_cmdline_args** (*list[str], optional*) – Additional command line arguments to pass to EDEN. Refer to the EDEN user's manual, "Command line API" for more details.
- **full_cmdline** (*list[str], optional*) – Specify the exact `argv` to be passed to EDEN. Refer to the EDEN user's manual, "Command line API" for more details.
- **executable_path** (*str, optional*) – Select a specific executable of EDEN to run the simulation with. Useful for custom installations and uses of EDEN.

Returns

-
- **info** (*dict[str, dict]*) – The requested information, keyed by cell type name.

Notes

For physical (i.e. not artificial) cells:

Each keyed entry of *info* may have the following `dict` entries:

`'comp_parent'`: ndarray[(n_comps), int]
The tree parent of each compartment. The first value, that represents the *root* (as modelled) of the neuron, is -1.

`'comp_start_pos'`: ndarray[(n_comps, 3), float]
The position of the most `proximal` end of each compartment.

`'comp_end_pos'`: ndarray[(n_comps, 3), float]
The position of the most `distal` end of each compartment.

`'comp_midpoint'`: ndarray[(n_comps, 3), float]
The 3-D position coordinates of the midpoint of each compartment, relative to the morphology's origin. *NB:* Due to curvature it may not lay exactly between *comp_start_pos* and 'comp_end_pos', or on the line joining them.

`'comp_midpoint_segment'`: ndarray[(n_comps), int]
The NeuroML containing the midpoint of each compartment.

`'comp_midpoint_fractionAlong'`: ndarray[(n_comps), float]
The 0-to-1, proximal-to-distal, NeuroML position where each compartment lies on its *comp_midpoint_segment*.

`'comp_length'`: ndarray[(n_comps), float]
The length of the neurite cable belonging to each compartment. *NB:* Due to curvature and explicit discontinuities this may be more or less than the straight-line distance between *comp_start_pos* and *comp_end_pos*.

`'comp_path_length_from_root'`: ndarray[(n_comps), float]
The distance from the neuron's root tracing back the anatomical path, in microns. Some biophysical attributes track this distance.

`'comp_area'`: ndarray[(n_comps), float]
The total membrane area per compartment, in μm^2.

`'comp_volume'`: ndarray[(n_comps), float]
The total enclosed volume per compartment, in μm^3.

'comp_capacitance': ndarray[(n_comps), float]
The total membrane capacitance per compartment, in *pF*.

'comp_conductance_to_parent': ndarray[(n_comps), float]
The electrical cytosolic conductance between each compartment and its tree parent, in *nS*. The first value for the tree root is zero (not applicable).

'segment_groups': dict[str, dict]
Details about each NeuroML segment group, keyed by name.

Each keyed entry of *segment_groups* may have the following dict entries:

- `'comps'`: The numbered compartments included in the segment group.
- `'cable_compartments'`: Same as above but in guaranteed parent-to-child order, for ["unbranched cable" groups](intro_spatial.ipynb#The-unbranched-section-directive).
- `'cable_comps_fraction_along'`: The position along the *entire cable* for the corresponding `cable_compartments`, where 0 represents the start and 1 the end of the cell. Useful for porting NEURON models with variability over cell "sections".

'mesh_vertices': ndarray[(n_verts, 3), float]
The vertex coordinates of the 3-D mesh representing the neuron, in microns.

'mesh_faces': ndarray[(n_faces, 3), int]
The triangles of *0-indexed vertices* forming the neuron's 3-D mesh.

'mesh_comp_per_face': ndarray[n_faces, int] (optional)
The neuron compartment that each triangle of the mesh maps to.

Useful for selectively colouring relevant regions of the neuron, and displaying the neuron in false colour.

NB: It is present when discretisation is available for the selected mesh generation method.

For physical cells, most entries have as many values as there are compartments. Spatial attributes (position coordinates, distance, and such) are given in microns following the NeuroML convention.

For thinking of the neuron as a "tree" data structure, refer to:

- the *Book of Genesis*, chapter 5, section 5: http://www.genesis-sim.org/GENESIS/iBoG/iBoGpdf/chapt5.pdf
- the TREES toolbox for analysing the structure of neurons: https://doi.org/10.1371/journal.pcbi.1000877
- and the related literature.

PYTHON MODULE INDEX

e

eden_simulator, 237
eden_simulator.display.animation, 238
eden_simulator.display.spatial, 239
eden_simulator.display.spatial.k3d, 239
eden_simulator.display.spatial.pyvista, 241
eden_simulator.experimental, 241

INDEX

Symbols

__init__() (*Plot method*), 240

D

decompress_cells() (*in module eden_simulator.display.spatial.k3d*), 240
DeduceColormap() (*in module eden_simulator.display.spatial.k3d*), 239
display() (*Plot method*), 240

E

eden_simulator
    module, 237
eden_simulator.display.animation
    module, 238
eden_simulator.display.spatial
    module, 239
eden_simulator.display.spatial.k3d
    module, 239
eden_simulator.display.spatial.pyvista
    module, 241
eden_simulator.experimental
    module, 241
explain_cell() (*in module eden_simulator.experimental*), 241

G

get_mesh_info() (*in module eden_simulator.display.spatial*), 239
get_neuron_mesh() (*in module eden_simulator.display.spatial.pyvista*), 241
get_snapshot() (*Plot method*), 240
get_verts_faces_per_comp() (*in module eden_simulator.display.spatial*), 239
GetAutoFps() (*in module eden_simulator.display.animation*), 238
GetLemsLocatorsForCell() (*in module eden_simulator.experimental*), 241

I

IntToRgb() (*in module eden_simulator.display.spatial.k3d*), 239

M

MinimizePlot() (*in module eden_simulator.display.spatial.k3d*), 239
module
    eden_simulator, 237
    eden_simulator.display.animation, 238
    eden_simulator.display.spatial, 239
    eden_simulator.display.spatial.k3d, 239
    eden_simulator.display.spatial.pyvista, 241
    eden_simulator.experimental, 241

P

Plot (*class in eden_simulator.display.spatial.k3d*), 240
plot_neuron() (*in module eden_simulator.display.spatial.k3d*), 240

R

RgbToInt() (*in module eden_simulator.display.spatial.k3d*), 240
runEden() (*in module eden_simulator*), 237

S

show_html() (*Plot method*), 240
subsample_trajectories() (*in module eden_simulator.display.animation*), 238