brian2tools Documentation

Brian authors

Jul 16, 2021

Contents

The *brian2tools* package is a collection of useful tools for the Brian 2 simulator. The project is still in its infancy but it already provides helpful functions for plotting and exporting a neural model to the NeuroML2 format. In the future it will be extended to also provide analysis and additional export/import functions.

Please contact us at brian-development@googlegroups.com (https://groups.google.com/forum/#!forum/brian-development) if you are interested in contributing.

Please report bugs at the github issue tracker or to briansupport@googlegroups.com (https://groups.google.com/forum/#!forum/briansupport).

CHAPTER 1

Contents

1.1 Release notes

1.1.1 brian2tools 0.3

This release adds a number of major new features to the package. In particular:

- the ability to *import neuronal morphologies from NeuroML descriptions*, including information about channel types, channel densities, and general biophysical properties. This feature has been developed by Kapil Kumar (@kapilkd13) as part of the Google Summer of Code 2018.
- improved support for exporting neuronal models to NeuroML. As a demonstration of this framework, it is now possible to export a Brian model description as *human-readable markdown*. This feature has been developed by Vigneswaran C (@Vigneswaran-Chandrasekaran) as part of 2020's Google Summer of Code.
- improved support for *plotting morphologies*. It is now possible to plot a morphology and color code a value that varies across the neuron (e.g. the membrane potential) on top of it. Thanks to @mqcapelle for contributing this feature.

Contributions

- Kapil Kumar (@kapilkd13)
- Vigneswaran C (@Vigneswaran-Chandrasekaran)
- @mqcapelle
- Marcel Stimberg (@mstimberg)

We also acknowledge the support of the INCF in the organization of the Google Summer of Code internships.

1.1.2 brian2tools 0.2.1.1

A maintenance release that adds conda packages for Python 3.6. Also fixes a small bug in morphology plotting.

1.1.3 brian2tools 0.2.1

This release adds initial support to export Brian 2 simulations to the NeuroML2 and LEMS format. This feature has been added by Dominik Krzemiński (@dokato) as part of the Google Summer of Code 2016 under the umbrella of the INCF organization. It currently allows to export neuronal models (with threshold, reset and refractory definition), but not synaptic models or multi-compartmental neurons. See the *NeuroML exporter* documentation for details.

Contributions

- Dominik Krzemiński (@dokato)
- Marcel Stimberg (@mstimberg)

We also thank Padraig Gleeson (@pgleeson) for help and guidance concerning NeuroML2 and LEMS.

1.1.4 brian2tools 0.1.2

This is mostly a bug-fix release but also adds a few new features and improvements around the plotting of synapses (see below).

Improvements and bug fixes

- Synaptic plots of the "image" type with `plot_synapses` (also the default for `brian_plot` for synapses between small numbers of neurons) where plotting a transposed version of the correct connection matrix that was in addition potentially cut off and therefore not showing all connections (#6).
- Fix that `brian_plot` was not always returning the `Axes` object.
- Enable direct calls of `brian_plot` with a synaptic variable or an indexed `StateMonitor` (to only plot a subset of recorded cells).
- Do not plot `0` as a value for non-existing synapses in `image` and `hexbin`-style plots.
- A new function `add_background_pattern` to add a hatching pattern to the figure background (for colormaps that include the background color).

Testing, suggestions and bug reports:

- Ibrahim Ozturk

1.1.5 brian2tools 0.1

This is the first release of the `brian2tools` package (a collection of optional tools for the `Brian 2 simulator`), providing several plotting functions to plot model properties (such as synapses or morphologies) and simulation results (such as raster plots or voltage traces). It also introduces a convenience function `brian_plot` which takes a Brian 2 object as an argument and produces a plot based on it. See *Plotting tools* for details.

Contributions

The code in this first release has been written by Marcel Stimberg (@mstimberg).

1.2 User's guide

1.2.1 Installation instructions

The `brian2tools` package is a pure Python package that should be installable without problems most of the time, either using the Anaconda distribution or using `pip`. However, it depends on the `brian2` package which has more complex requirements for installation. The recommended approach is therefore to first install `brian2` following the instruction in the Brian 2 documentation and then use the same approach (i.e. either installation with Anaconda or installation with `pip`) for `brian2tools`.

Installation with Anaconda

Since `brian2tools` (and `brian2` on which it depends) are not part of the main Anaconda distribution, you have to install it from the brian-team channel. To do so use:

```
conda install -c brian-team brian2tools
```

You can also permanently add the channel to your list of channels:

```
conda config --add channels brian-team
```

This has only to be done once. After that, you can install and update the brian2 packages as any other Anaconda package:

```
conda install brian2tools
```

Installing optional requirements

The 3D plotting of morphologies (see *Morphologies in 2D or 3D*) depends on the mayavi package. You can install it from anaconda as well:

```
conda install mayavi
```

Installation with pip

If you decide not to use Anaconda, you can install *`brian2tools`* from the Python package index: https://pypi.python.org/pypi/brian2tools

To do so, use the `pip` utility:

```
pip install brian2tools
```

You might want to add the `--user` flag, to install Brian 2 for the local user only, which means that you don't need administrator privileges for the installation.

If you have an older version of pip, first update pip itself:

```
# On Linux/MacOsX:
pip install -U pip

# On Windows
python -m pip install -U pip
```

If you don't have `pip` but you have the `easy_install` utility, you can use it to install `pip`:

```
easy_install pip
```

If you have neither `pip` nor `easy_install`, use the approach described here to install `pip`: https://pip.pypa.io/en/latest/installing.htm

Installing optional requirements

The 3D plotting of morphologies (see *Morphologies in 2D or 3D*) depends on the mayavi package. Follow its installation instructions to install it.

1.2.2 Plotting tools

The *brian2tools* package offers plotting tools for some standard plots of various `brian2` objects. It provides two approaches to produce plots:

1. a convenience method *brian_plot* that takes an object such as a `SpikeMonitor` and produces a useful plot out of it (in this case, a raster plot). This method is rather meant for quick investigation than for creating publication-ready plots. The details of these plots might change in future versions, so do not rely in this function if you expect your plots to stay the same.
2. specific methods such as *plot_raster* or *plot_morphology*, that allow for more detailed settings of plot parameters.

In both cases, the plotting functions will return a reference to the matplotlib `Axes` object, allowing to further tweak the code (e.g. setting a title, changing the labels, etc.). The functions will automatically take care of labelling the plot with the names of the plotted variables and their units (for this to work, the "unprocessed" objects have to be used: e.g. plotting `neurons.v` can automatically state the name `v` and the unit of `v`, whereas `neurons.v[:]` can only state its unit and `np.array(neurons.v)` will state neither name nor unit).

Overview

Plotting recorded activity

We'll use the following example (the *CUBA example* from Brian 2) as a demonstration.

```
from brian2 import *

Vt = -50*mV
Vr = -60*mV

eqs = '''dv/dt  = (ge+gi-(v + 49*mV))/(20*ms) : volt (unless refractory)
         dge/dt = -ge/(5*ms) : volt
         dgi/dt = -gi/(10*ms) : volt
      '''
P = NeuronGroup(4000, eqs, threshold='v>Vt', reset='v = Vr', refractory=5*ms,
                method='linear')
P.v = 'Vr + rand() * (Vt - Vr)'
P.ge = 0*mV
P.gi = 0*mV

we = (60*0.27/10)*mV # excitatory synaptic weight (voltage)
wi = (-20*4.5/10)*mV # inhibitory synaptic weight
Ce = Synapses(P[:3200], P, on_pre='ge += we')
Ci = Synapses(P[3200:], P, on_pre='gi += wi')
Ce.connect(p=0.02)
Ci.connect(p=0.02)

spike_mon = SpikeMonitor(P)
rate_mon = PopulationRateMonitor(P)
state_mon = StateMonitor(P, 'v', record=[0, 100, 1000])  # record three cells

run(1 * second)
```

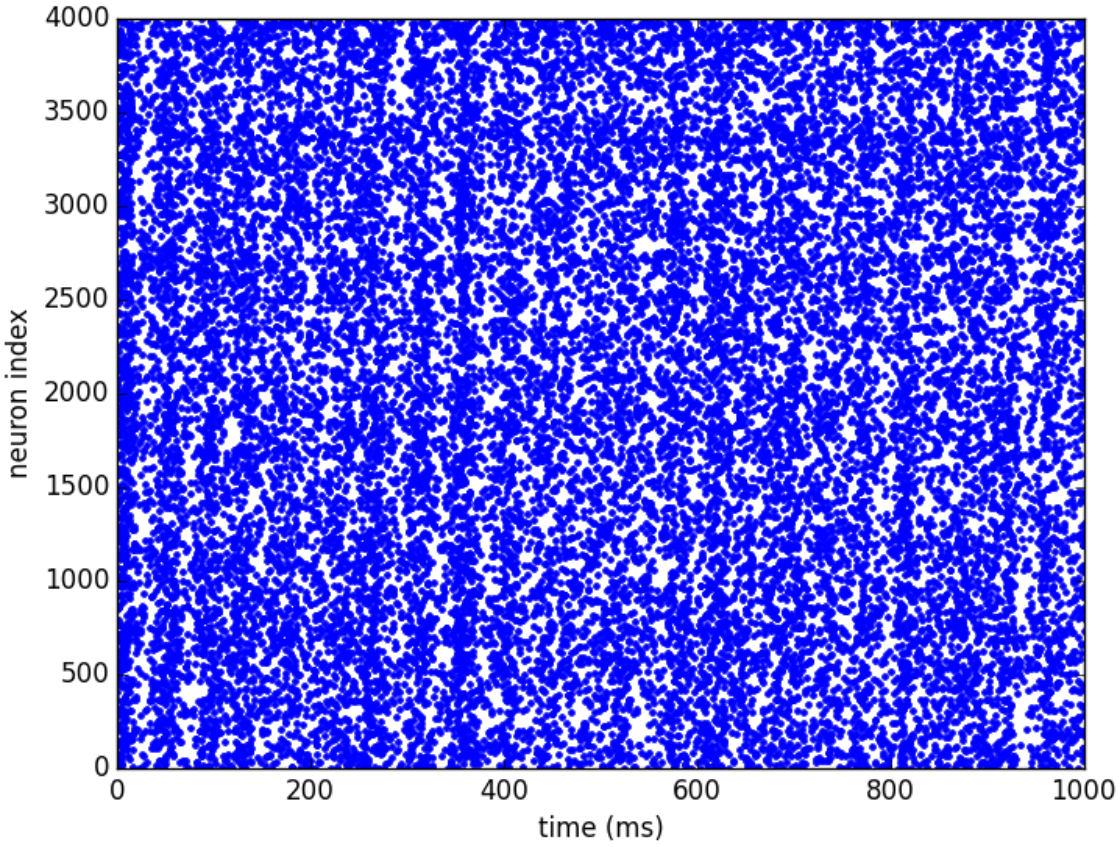
We will also assume that `brian2tools` has been imported like this:

```
from brian2tools import *
```

Spikes

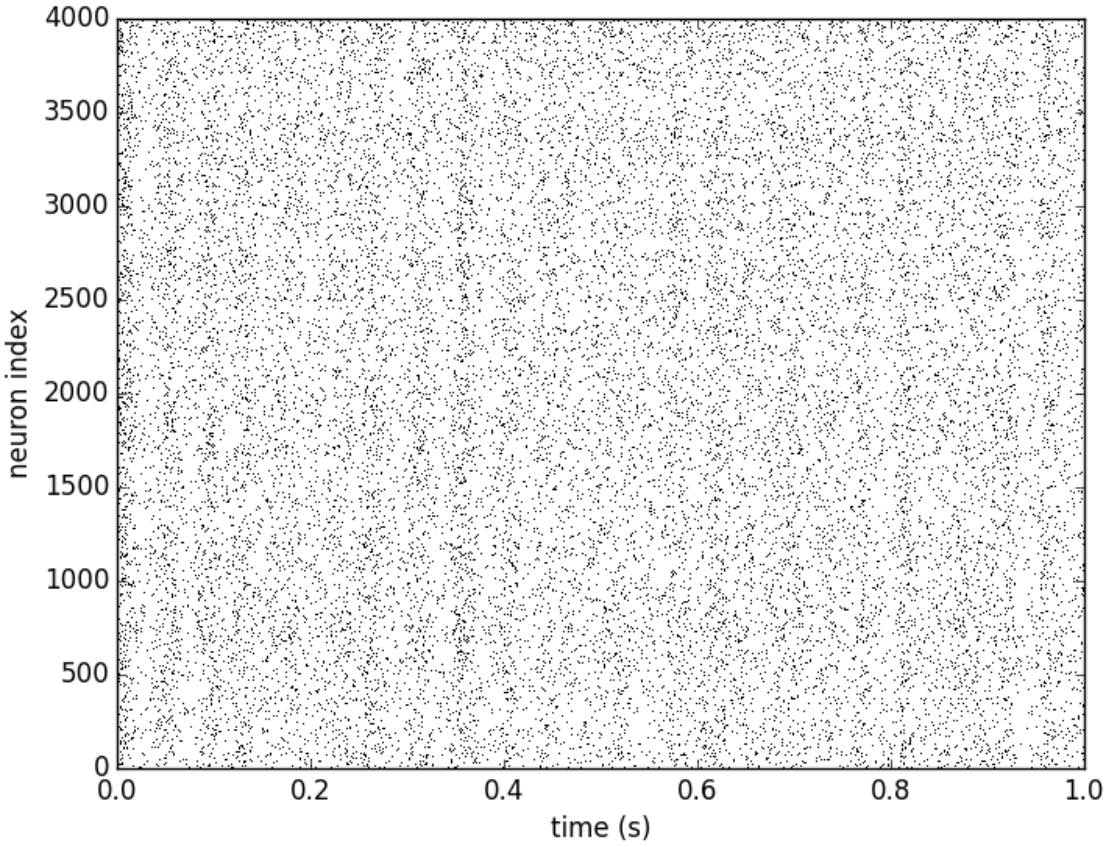
To plot a basic raster plot, you can call *brian_plot* with the `SpikeMonitor` as its argument:

```
brian_plot(spike_mon)
```



To have more control over the plot, or to plot spikes that are not stored in a `SpikeMonitor`, use *`plot_raster`*:

```
plot_raster(spike_mon.i, spike_mon.t, time_unit=second, marker=',', color='k')
```



Rates

Calling *brian_plot* with the PopulationRateMonitor will plot the rate smoothed with a Gaussian window with 1ms standard deviation.:

```
brian_plot(rate_mon)
```

To plot the rate with a different smoothing and/or to set other details of the plot use *plot_raster*:

```
plot_rate(rate_mon.t, rate_mon.smooth_rate(window='flat', width=10.1*ms),
          linewidth=3, color='gray')
```

State variables

Finally, calling *brian_plot* with the StateMonitor will plot the recorded voltage traces:

```
brian_plot(state_mon)
```

By indexing the `StateMonitor`, the plot can be restricted to a subset of the recorded neurons:

```
brian_plot(state_mon[1000])
```

Again, for more detailed control you can directly use the *plot_state* function. Here we also demonstrate the use of the returned `Axes` object to add a legend to the plot:

```
ax = plot_state(state_mon.t, state_mon.v.T, var_name='membrane potential', lw=2)
ax.legend(['neuron 0', 'neuron 100', 'neuron 1000'], frameon=False, loc='best')
```

Plotting synaptic connections and variables

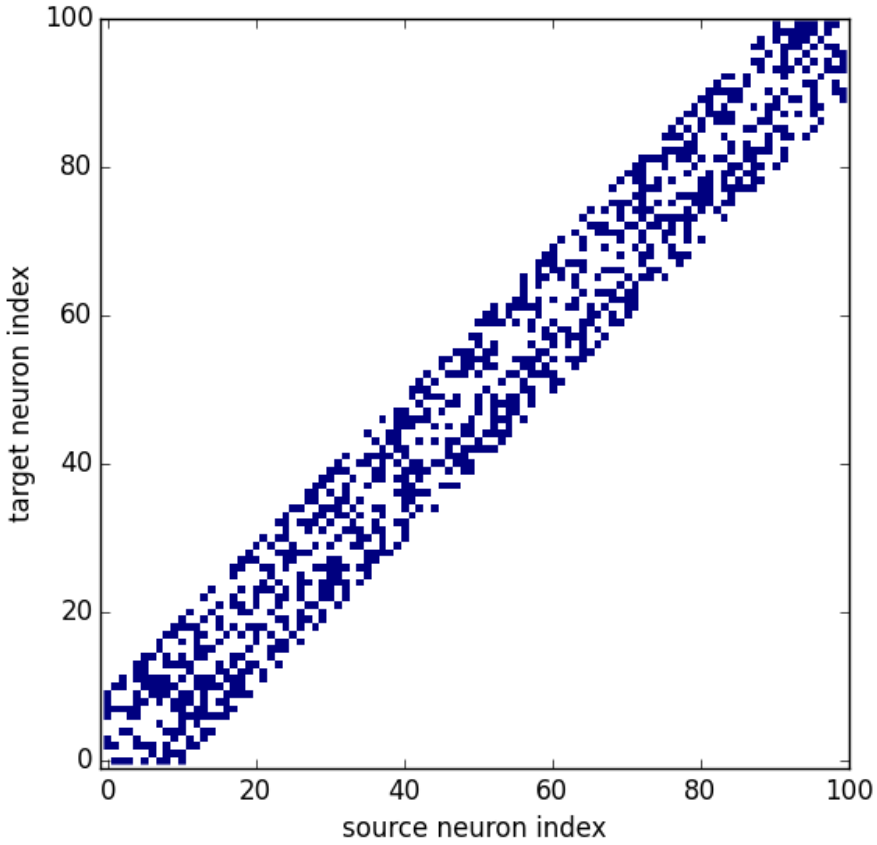
For the following examples, we create synapses and synaptic weights according to "distances" (differences between the source and target indices):

```
from brian2 import *

group = NeuronGroup(100, 'dv/dt = -v / (10*ms) : volt',
                    threshold='v > -50*mV', reset='v = -60*mV')

synapses = Synapses(group, group, 'w : volt', on_pre='v += w')

# Connect to cells with indices no more than +/- 10 from the source index with
# a probability of 50% (but do not create self-connections)
synapses.connect(j='i+k for k in sample(-10, 10, p=0.5) if k != 0',
                 skip_if_invalid=True)  # ignore values outside of the limits
# Set synaptic weights depending on the distance (in terms of indices) between
# the source and target cell and add some randomness
synapses.w = '(exp(-(i - j)**2/10.) + 0.5 * rand())*mV'
# Set synaptic weights randomly
synapses.delay = '1*ms + 2*ms*rand()'
```

Connections

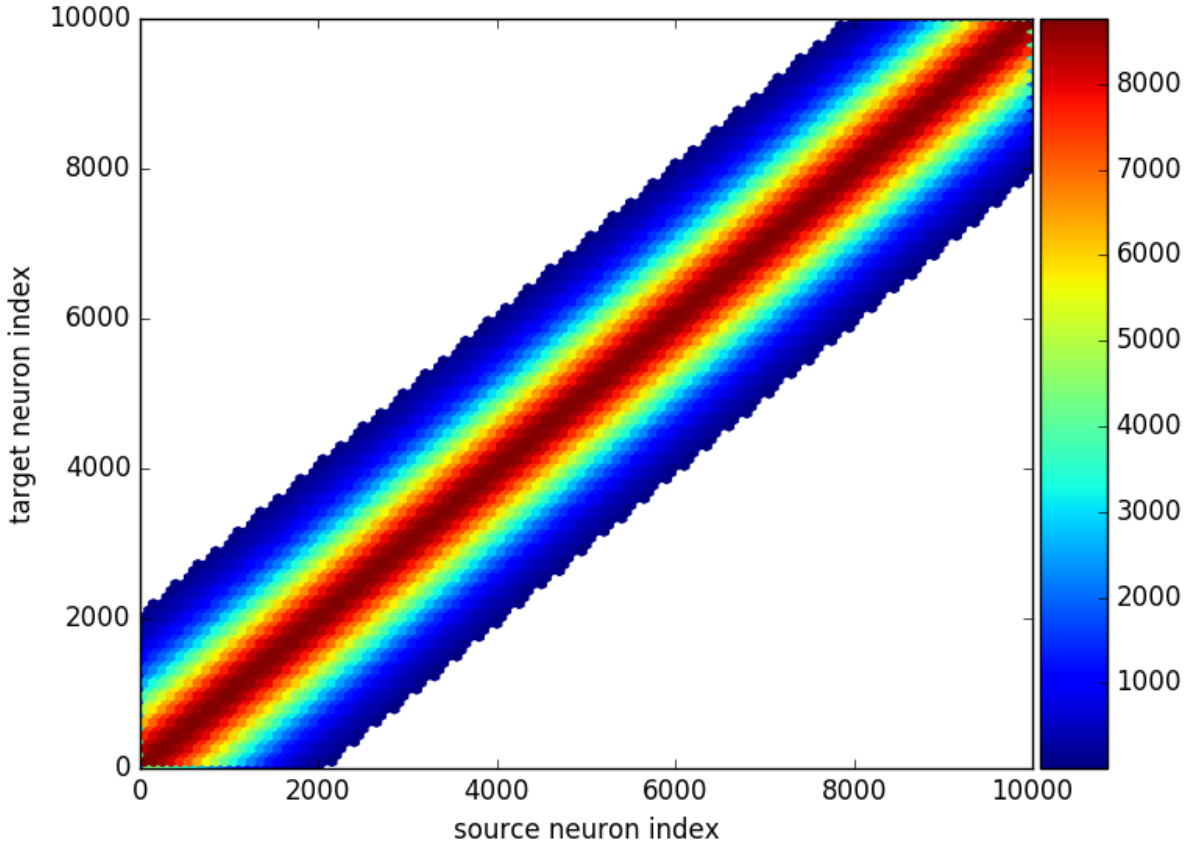
A call of *brian_plot* with a `Synapses` object will plot all connections, plotting either the matrix as an image, the connections as a scatter plot, or a 2-dimensional histogram (using matplotlib's `hexbin` function). The decision which type of plot to use is based on some heuristics applied to the number of synapses and might possibly change in future versions:

```
brian_plot(synapses)
```



As explained above, for a large connection matrix this would instead use an approach based on a hexagonal 2D histogram:

```
big_group = NeuronGroup(10000, '')
many_synapses = Synapses(big_group, big_group)
many_synapses.connect(j='i+k for k in range(-2000, 2000) if rand() < exp(-(k/1000.
↪)**2)',
                      skip_if_invalid=True)
brian_plot(many_synapses)
```



Under the hood *brian_plot* calls *plot_synapses* which can also be used directly for more control:

```
plot_synapses(synapses.i, synapses.j, plot_type='scatter', color='gray', marker='s')
```

Synaptic variables (weights, delays, etc.)

Synaptic variables such as synaptic weights or delays can also be plotted with *brian_plot*:

```
subplot(1, 2, 1)
brian_plot(synapses.w)
subplot(1, 2, 2)
brian_plot(synapses.delay)
tight_layout()
```

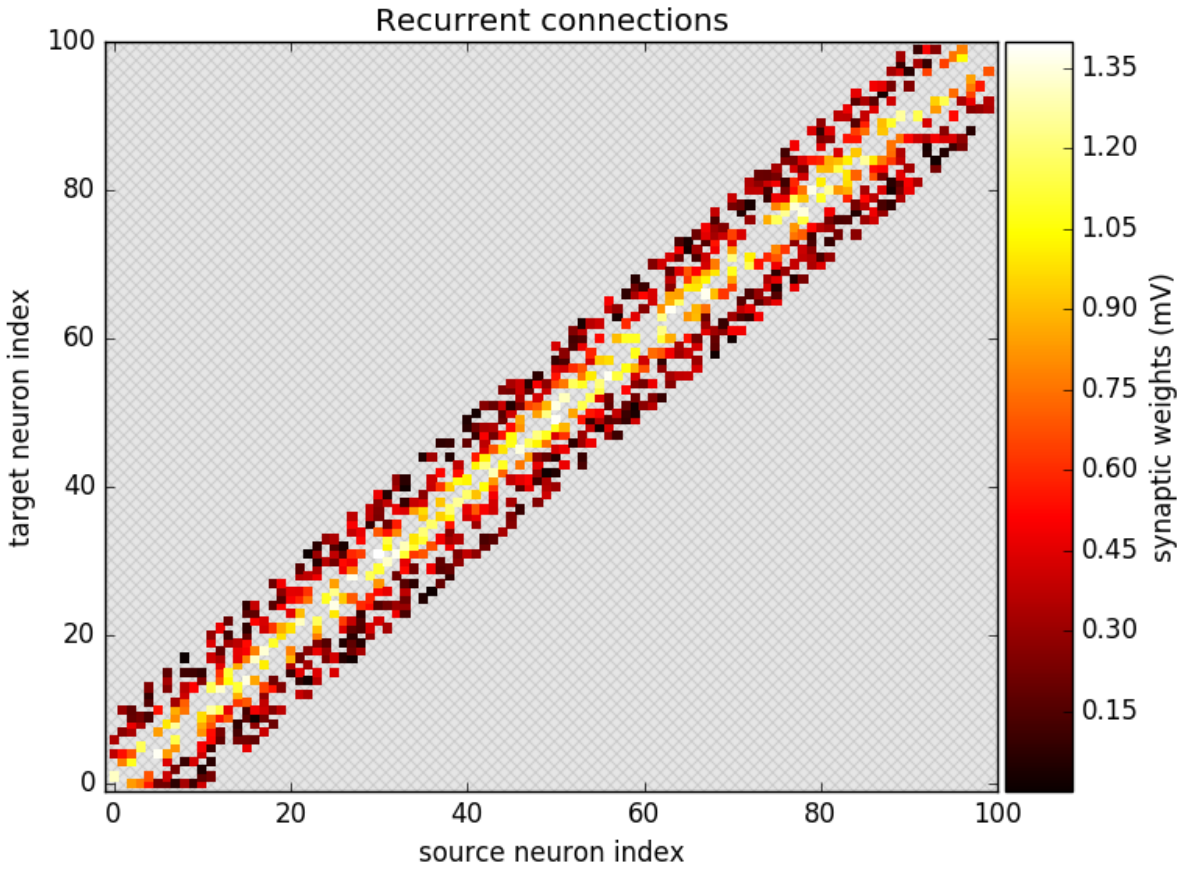
Again, using *plot_synapses* provides more control. The following code snippet also calls the *add_background_pattern* function to make the distinction between white color values and the background clearer:

```
ax = plot_synapses(synapses.i, synapses.j, synapses.w, var_name='synaptic weights',
                   plot_type='scatter', cmap='hot')
```

(continues on next page)

(continued from previous page)

```
add_background_pattern(ax)
ax.set_title('Recurrent connections')
```



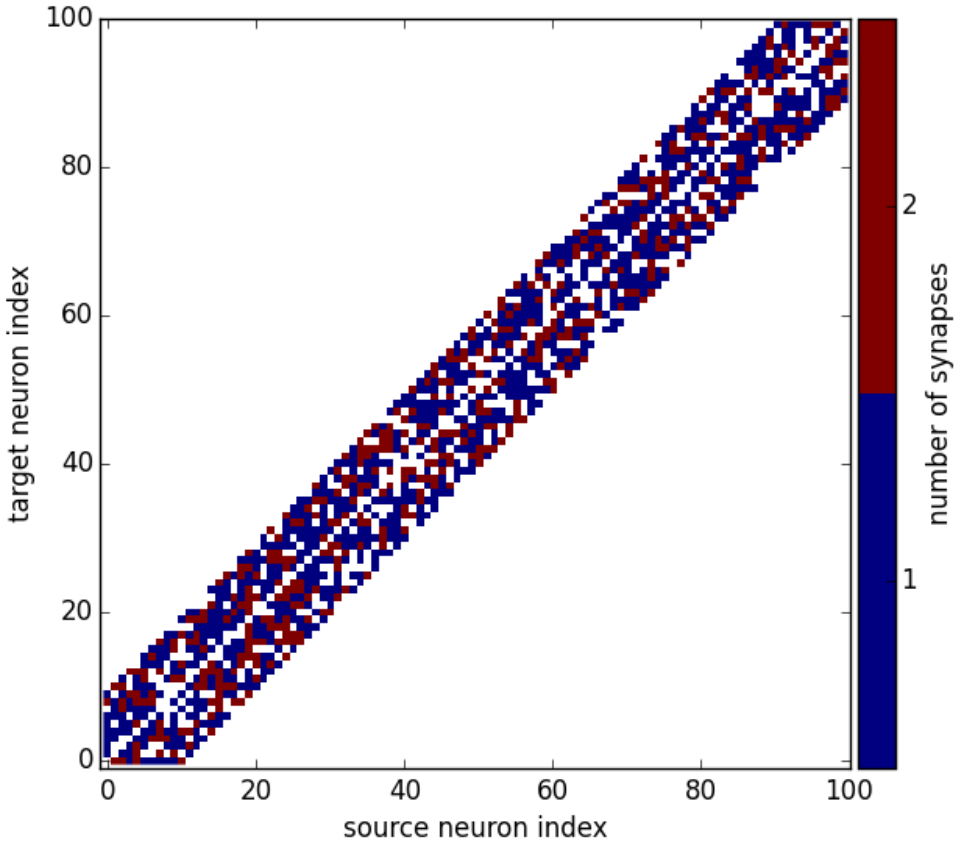
Multiple synapses per source-target pair

In Brian, source-target pairs can be connected by more than a single synapse. In this case you cannot plot synaptic state variables (because it is ill-defined what to plot) but you can still plot connections which will show how many synapses exists. For example, if we make the same `connect` from above a second time, the new synapses will be added to the existing ones so some source-target pairs are now connected by two synapses:

```
synapses.connect(j='i+k for k in sample(-10, 10, p=0.5) if k != 0',
                 skip_if_invalid=True)
```

Calling *brian_plot* or *plot_synapses* will now show the number of synapses between each pair of neurons:

```
brian_plot(synapses)
```



Plotting neuronal morphologies

In the following, we'll use a reconstruction from the Destexhe lab (a neocortical pyramidal neuron from the cat brain[1]) that we load into Brian:

```
from brian2 import *

morpho = Morphology.from_file('51-2a.CNG.swc')
```

Dendograms

Calling *brian_plot* with a Morphology will plot a dendogram:

```
brian_plot(morpho)
```

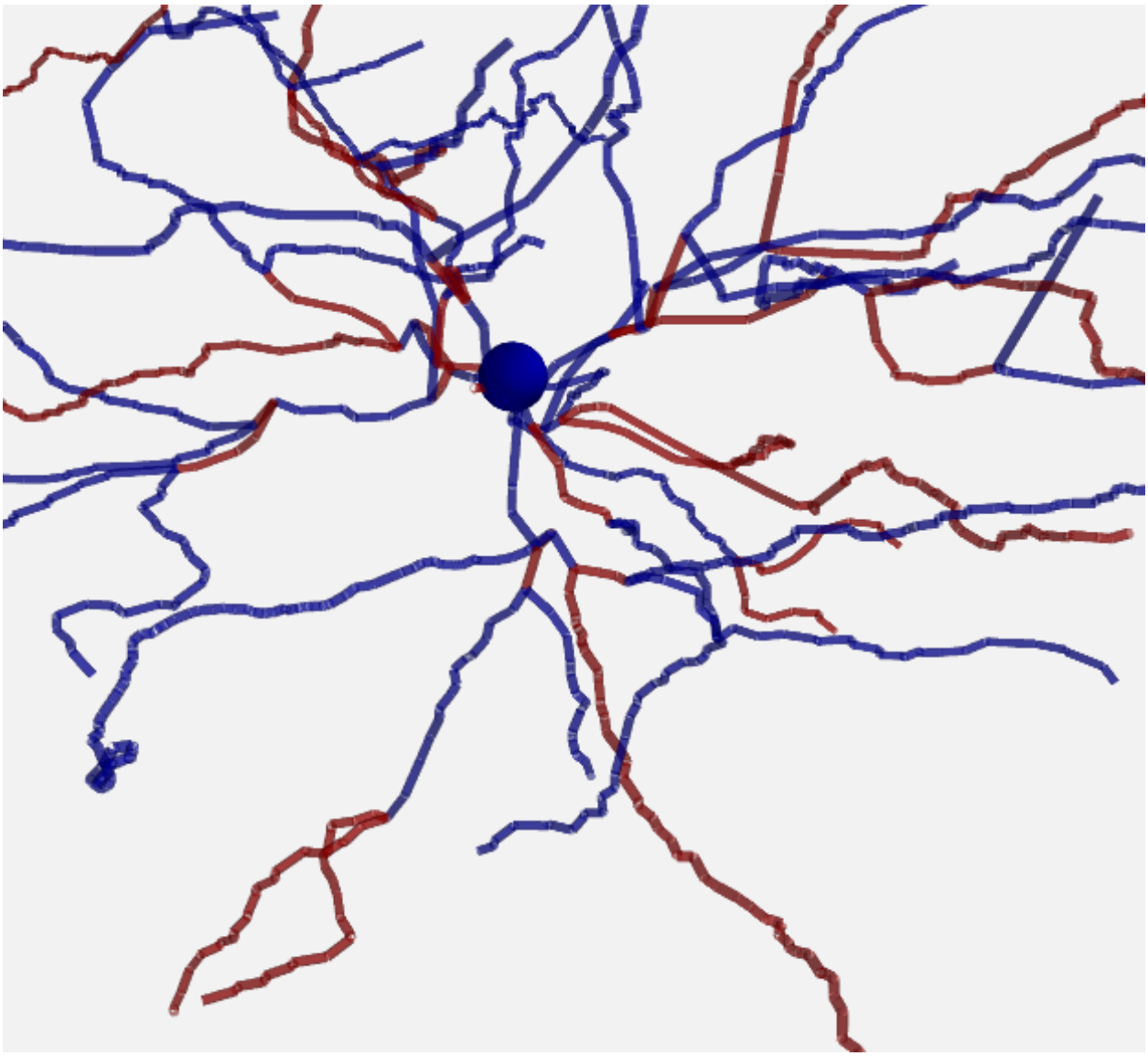
The *plot_dendrogram* function does the same thing, but in contrast to the other plot functions it does not allow any customization at the moment, so there is no benefit over using *brian_plot*.

[1] Available at http://neuromorpho.org/neuron_info.jsp?neuron_name=51-2a

Morphologies in 2D or 3D

In addition to the dendogram which only plots the general structure but not the actual morphology of the neuron in space, you can plot the morphology using *plot_morphology*. For a 3D morphology, this will plot the morphology in 3D using the Mayavi package

```
plot_morphology(morpho)
```

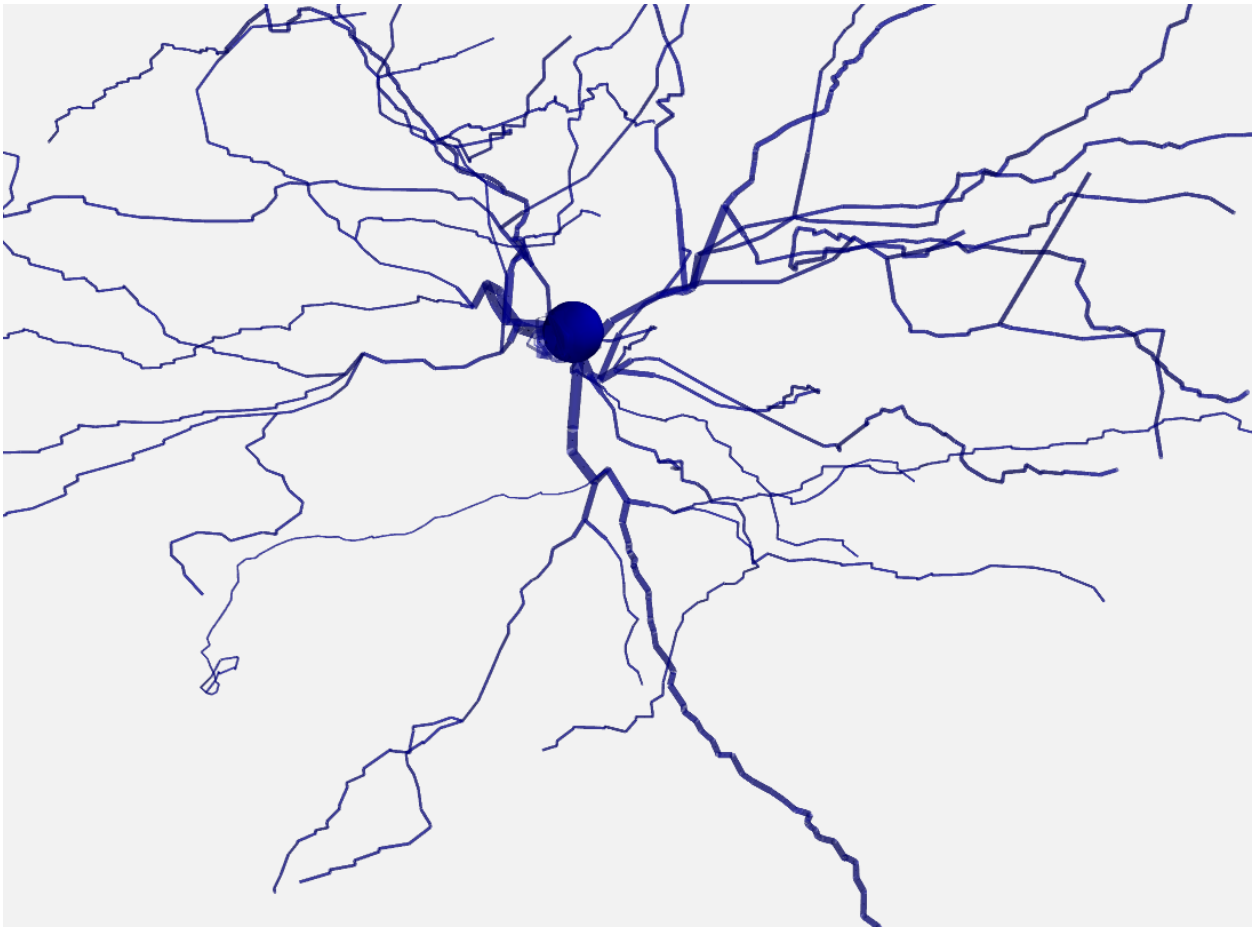


For artificially created morphologies (where one might only use coordinates in 2D) or to get a quick view of a morphology, you can also plot it in 2D (this will be done automatically if the coordinates are 2D only):

```
plot_morphology(morpho, plot_3d=False)
```

Both 2D and 3D morphology plots can be further customized, e.g. they can show the width of the compartments and do not use the default alternation between blue and red for each section:

```
plot_morphology(morpho, plot_3d=True, show_compartments=True,
                show_diameter=True, colors=('darkblue',))
```

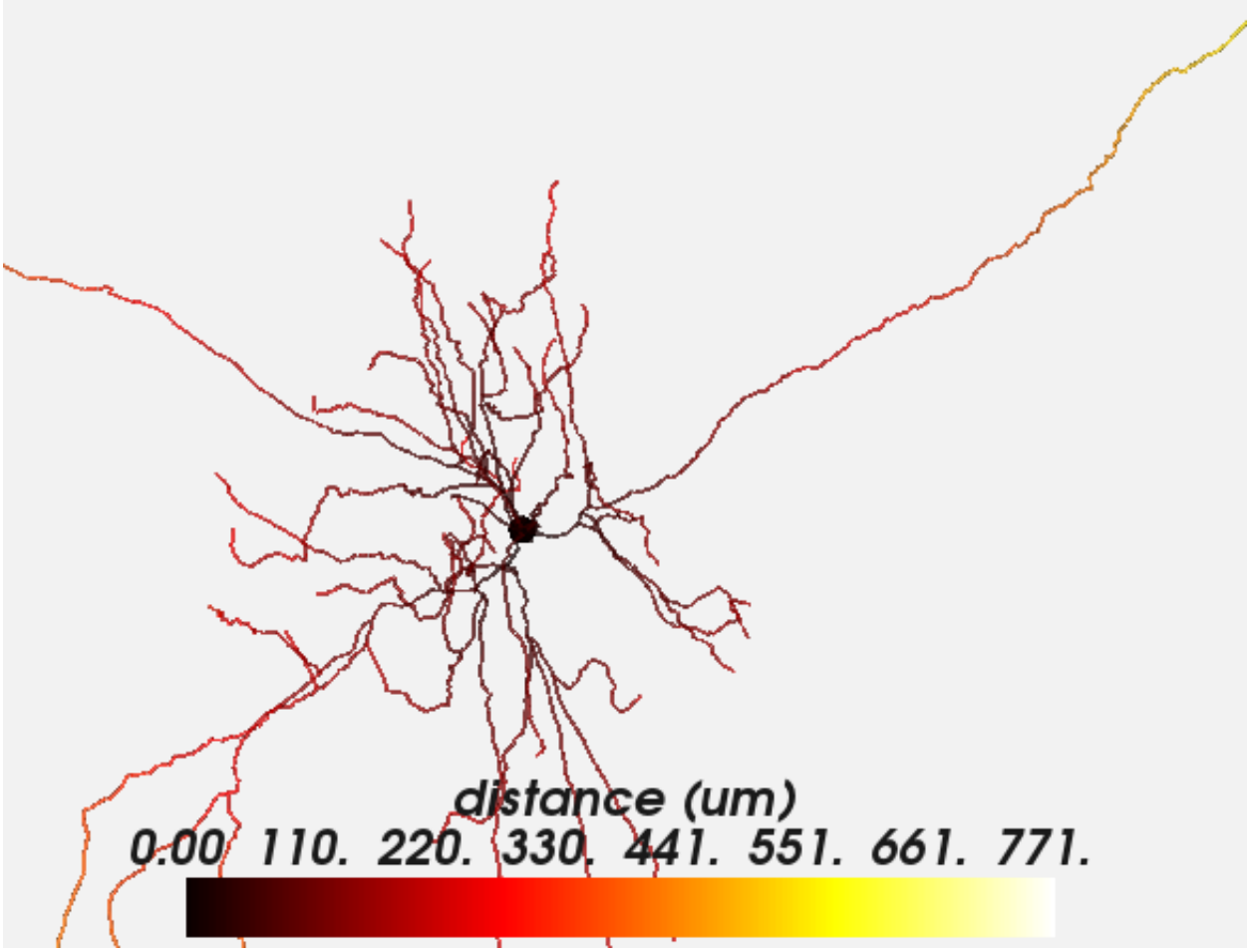


The `plot_morphology` function also makes it possible to use colors according to some property that varies between the compartments of a `SpatialNeuron`. This could be the membrane potential, a current or conductance, a channel density or morphological parameters like the distance to the soma. By default, it will add a colorbar and figure out a useful unit scale (e.g. mV for membrane potential values, or µm for distances):

```
# neuron = SpatialNeuron(...)
plot_morphology(neuron.morphology, values=neuron.distance,
                plot_3d=False)
```

This works for 3D plots as well:

```
plot_morphology(neuron.morphology, values=neuron.distance,
                plot_3d=True)
```



Both types of plots can be further customized, e.g. to change the details of the colorbar, the range, and the colormap:

```
plot_morphology(neuron.morphology, values=neuron.distance,
                value_norm=(50*um, 200*um),
                value_colormap='viridis', value_unit=mm,
                value_colorbar={'label': 'distance from soma in mm',
                                'extend': 'both'},
                plot_3d=False)
```

1.2.3 NeuroML exporter

This is a short overview of the *nmlexport* package, providing functionality to export Brian 2 models to NeuroML2.

NeuroML is a XML-based description that provides a common data format for defining and exchanging descriptions of neuronal cell and network models (NML project website).

Overview

- *Limitations*

Working example

As a demonstration, we use a simple unconnected Integrate & Fire neuron model with refractoriness and given initial values.

```
from brian2 import *
import brian2tools.nmlexport

set_device('neuroml2', filename="nml2model.xml")

n = 100
duration = 1*second
tau = 10*ms

eqs = '''
dv/dt = (v0 - v) / tau : volt (unless refractory)
v0 : volt
'''
group = NeuronGroup(n, eqs, threshold='v > 10*mV', reset='v = 0*mV',
                    refractory=5*ms, method='linear')
group.v = 0*mV
group.v0 = '20*mV * i / (N-1)'

rec_idx = [2, 63]
statemonitor = StateMonitor(group, 'v', record=rec_idx)
spikemonitor = SpikeMonitor(group, record=rec_idx)

run(duration)
```

The use of the exporter requires only a few changes to an existing Brian 2 script. In addition to the standard `brian2` import at the beginning of your script, you need to import the *`brian2tools.nmlexport`* package. You can then set a "device" called `neuroml2` which will generate NeuroML2/LEMS code instead of executing your model. You will also have to specify a keyword argument `filename` with the desired name of the output file.

The above code will result in a file `nml2model.xml` and an additional file `LEMSUnitsConstants.xml` with units definitions in form of constants (necessary due to the way units are handled in LEMS equations).

The file `nml2model.xml` will look like this:

```
<Lems>
  <Include file="NeuroML2CoreTypes.xml"/>
  <Include file="Simulation.xml"/>
  <Include file="LEMSUnitsConstants.xml"/>
  <ComponentType extends="baseCell" name="neuron1">
    <Property dimension="voltage" name="v0"/>
    <Property dimension="time" name="tau"/>
    <EventPort direction="out" name="spike"/>
    <Exposure dimension="voltage" name="v"/>
    <Dynamics>
      <StateVariable dimension="voltage" exposure="v" name="v"/>
      <OnStart>
        <StateAssignment value="0" variable="v"/>
      </OnStart>
      <Regime name="refractory">
```

(continues on next page)

(continued from previous page)

```
        <StateVariable dimension="time" name="lastspike"/>
        <OnEntry>
          <StateAssignment value="t" variable="lastspike"/>
        </OnEntry>
        <OnCondition test="t .gt. ( lastspike + 5.*ms )">
          <Transition regime="integrating"/>
        </OnCondition>
      </Regime>
      <Regime initial="true" name="integrating">
        <TimeDerivative value="(v0 - v) / tau" variable="v"/>
        <OnCondition test="v .gt. (10 * mV)">
          <EventOut port="spike"/>
          <StateAssignment value="0*mV" variable="v"/>
          <Transition regime="refractory"/>
        </OnCondition>
      </Regime>
    </Dynamics>
  </ComponentType>
  <ComponentType extends="basePopulation" name="neuron1Multi">
    <Parameter dimension="time" name="tau_p"/>
    <Parameter dimension="none" name="N"/>
    <Constant dimension="voltage" name="mVconst" symbol="mVconst" value="1mV"/>
    <Structure>
      <MultiInstantiate componentType="neuron1" number="N">
        <Assign property="v0" value="20*mVconst * index /  ( N-1 ) "/>
        <Assign property="tau" value="tau_p"/>
      </MultiInstantiate>
    </Structure>
  </ComponentType>
  <network id="neuron1MultiNet">
    <Component N="100" id="neuron1Multipop" tau_p="10. ms" type="neuron1Multi"/>
  </network>
  <Simulation id="sim1" length="1s" step="0.1 ms" target="neuron1MultiNet">
    <Display id="disp0" timeScale="1ms" title="v" xmax="1000" xmin="0" ymax="11" ymin=
↪"0">
      <Line id="line3" quantity="neuron1Multipop[3]/v" scale="1mV" timeScale="1ms"/>
      <Line id="line64" quantity="neuron1Multipop[64]/v" scale="1mV" timeScale="1ms"/>
    </Display>
    <OutputFile fileName="recording_nml2model.dat" id="of0">
      <OutputColumn id="3" quantity="neuron1Multipop[3]/v"/>
      <OutputColumn id="64" quantity="neuron1Multipop[64]/v"/>
    </OutputFile>
    <EventOutputFile fileName="recording_nml2model.spikes" format="TIME_ID" id="eof">
      <EventSelection eventPort="spike" id="line3" select="neuron1Multipop[3]"/>
      <EventSelection eventPort="spike" id="line64" select="neuron1Multipop[64]"/>
    </EventOutputFile>
  </Simulation>
  <Target component="sim1"/>
</Lems>
```

The exporting device creates a new `ComponentType` for each cell definition implemented as a Brian 2 `NeuronGroup`. Later that particular `ComponentType` is bundled with the initial value assignment into a a new `ComponentType` (here called `neuron1Multi`) by `MultiInstantiate` and eventually a network (`neuron1MultiNet`) is created out of a defined `Component` (`neuron1Multipop`).

Note that the integration method does not matter for the NeuroML export, as NeuroML/LEMS only describes the model not how it is numerically integrated.

To validate the output, you can use the tool jNeuroML. Make sure that `jnml` has access to the `NeuroML2CoreTypes` folder by setting the `JNML_HOME` environment variable.

With `jnml` installed you can run the simulation as follows:

```
jnml nml2model.xml
```

Supported Features

Currently, the NeuroML2 export is restricted to simple neural models and only supports the following classes (and a single run statement per script):

- `NeuronGroup` - The definition of a neuronal model. Mechanisms like threshold, reset and refractoriness are taken into account. Moreover, you may set the initial values of the model parameters (like `v0` above).
- `StateMonitor` - If your script uses a `StateMonitor` to record variables, each recorded variable is transformed into to a `Line` tag of the `Display` in the NeuroML2 simulation and an `OutputFile` tag is added to the model. The name of the output file is `recording_<<filename>>.dat`.
- `SpikeMonitor` - A `SpikeMonitor` is transformed into an `EventOutputFile` tag, storing the spikes to a file named `recording_<<filename>>.spikes`.

Limitations

As stated above, the NeuroML2 export is currently quite limited. In particular, none of the following Brian 2 features are supported:

- Synapses
- Network input (`PoissonGroup`, `SpikeGeneratorGroup`, etc.)
- Multicompartmental neurons (`SpatialNeuronGroup`)
- Non-standard simulation protocols (multiple runs, `store`/`restore` mechanism, etc.).

1.2.4 NeuroML importer

This is a short overview of the *`nmlimport`* package, providing functionality to import a morphology – including some associated information about biophysical properties and ion channels – from an `.nml` file.

NeuroML is a XML-based description that provides a common data format for defining and exchanging descriptions of neuronal cell and network models (NeuroML project website).

Overview

Working example

As a demonstration, we will use the `nmlimport` library to generate a morphology and extract other related information from the `pyr_4_sym.cell.nml` nml file. You can download it from OpenSourceBrain's ACnet2 project.

```
from brian2tools.nmlimport.nml import NMLMorphology
nml_object = NMLMorphology('pyr_4_sym.cell.nml', name_heuristic=True)
```

This call provides the `nml_object` that contains all the information extracted from `.nml` file. When `name_heuristic` is set to `True`, morphology sections will be determined based on the segment names. In this case `Section` names will be created by combining names of the inner segments of the section. When set to `False`, all linearly connected segments combine to form a section with the name `sec{unique_integer}`.

- To obtain a `Morphology` object:

```
>>> morphology = nml_object.morphology

|

With this
```

With this `Morphology` object, you can use all of Brian's functions to get information about the cell:

```
>>> print(morphology.topology())
-|  [root]
   `-|  .apical0
      `---|  .apical0.apical2_3_4
      `-|  .apical0.apical1
   `-|  .basal0
      `-|  .basal0.basal1
      `-|  .basal0.basal2

>>> print(morphology.area)
[ 1228.36272755] um^2

>>> print(morphology.coordinates)
[[  0.   0.   0.]
[  0.  17.   0.]] um

>>> print(morphology.length)
[ 17.] um

>>> print(morphology.distance)
[ 8.5] um
```

- You can plot the morphology using brian2tool's *plot_morphology* function:

```
from brian2tools.plotting.morphology import plot_morphology
plot_morphology(morphology)
```

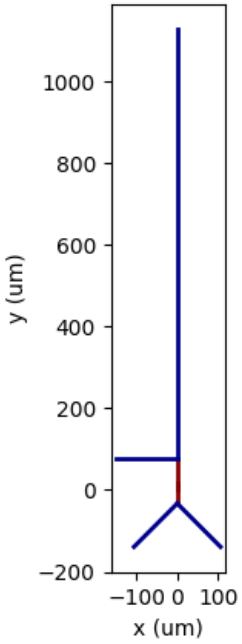


Fig. 1: Pyramidal cell's Morphology plot.

Handling SegmentGroup

In NeuroML, a `SegmentGroup` groups multiple segments into a single object, e.g. to represent a specific dendrite of a cell. This can later be used to apply operations on all segments of a group. The segment groups are stored in a dictionary, mapping the name of the group to the indices within the `Morphology`. Note that these indices are often identical to the `id` values used in the NeuroML file, but they do not have to be.

```
>>> from pprint import pprint
>>> pprint(nml_object.segment_groups)
{'all': [0, 1, 2, 3, 4, 5, 6, 7, 8],
 'apical0': [1],
 'apical1': [5],
 'apical2': [2],
 'apical3': [3],
 'apical4': [4],
 'apical_dends': [1, 2, 3, 4, 5],
 'background_input': [7],
 'basal0': [6],
 'basal1': [7],
 'basal2': [8],
 'basal_dends': [8, 6, 7],
 'basal_gaba_input': [6],
```

(continues on next page)

(continued from previous page)

```
 'dendrite_group': [1, 2, 3, 4, 5, 6, 7, 8],
 'middle_apical_dendrite': [3],
 'soma': [0],
 'soma_group': [0],
 'thalamic_input': [5]}
```

The file `pyr_4_sym.cell.nml` will look something like this:

```
<cell id="pyr_4_sym">
    <morphology id="morphology_pyr_4_sym">
        <segment id="0" name="soma">
            <proximal x="0.0" y="0.0" z="0.0" diameter="23.0"/>
            <distal x="0.0" y="17.0" z="0.0" diameter="23.0"/>
        </segment>
        ..........
        ..........
        ..........

        <segment id="6" name="basal0">
            <parent segment="0" fractionAlong="0.0"/>
            <proximal x="0.0" y="17.0" z="0.0" diameter="4.0"/>
            <distal x="0.0" y="-50.0" z="0.0" diameter="4.0"/>
        </segment>
        ..........
        ..........
        ..........

        <segmentGroup id="apical_dends">
            <include segmentGroup="apical0"/>
            <include segmentGroup="apical2"/>
            <include segmentGroup="apical3"/>
            <include segmentGroup="apical4"/>
            <include segmentGroup="apical1"/>
        </segmentGroup>

        <segmentGroup id="middle_apical_dendrite">
            <include segmentGroup="apical3"/>
        </segmentGroup>
        ........
        ........
        ........
    </morphology>
</cell>
```

Handling sections not connected at the distal end

If you look at the `line 12` in above .nml file, you can see `fractionAlong=0.0`. The `fractionAlong` value defines the point at which the given segment is connected with its parent segment. So a `fractionAlong` value of 1 means the segment is connected to bottom (distal) of its parent segment, 0 means it is connected to the top (proximal) of its parent segment. Similarly a value of 0.5 would mean the segment is connected to the middle point of its parent

segment. Currently, the `nmlimport` library supports `fractionAlong` value to be 0 or 1 only, as there is no predefined way to connect a segment at some intermediate point of its parent segment in `Brian`.

Extracting channel properties and Equations

The generated `nml_object` contains several dictionaries with biophysical information about the cell:

`properties`: A dictionary with general properties such as the threshold condition or the intracellular resistivity. The names are chosen to be consistent with the argument names in `SpatialNeuron`, in many cases you should therefore be able to directly pass this dictionary: ‘‘neuron = SpatialNeuron(..., **nml_object.properties)‘

`channel_properties`: A dictionary of reversal potentials and conductance densities for the different channels in the cell. The dictionary maps the name of segment groups to the name of the respective variable names (e.g. `g_Na`, `E_Na`, ...)

For the example file, these dictionaries look like this:

```
>>> pprint(nml_object.properties) # threshold, refractory etc.
{'Cm': 2.84 * ufarad / cmetre2,
 'Ri': 0.2 * kohm * cmetre,
 'refractory': 'v > 0. * volt',
 'threshold': 'v > 0. * volt'}

>>> pprint(nml_object.channel_properties) # reversal potentials and conductance␣
↪densities
{'all': {'E_LeakConductance_pyr': -66. * mvolt,
         'g_LeakConductance_pyr': 0.1420051 * msiemens / cmetre2},
 'soma_group': {'E_Ca_pyr': 80. * mvolt,
                'E_Kahp_pyr': -75. * mvolt,
                 'E_Kdr_pyr': -75. * mvolt,
                 'E_Na_pyr': 55. * mvolt,
                 'g_Ca_pyr': 10. * msiemens / cmetre2,
                 'g_Kahp_pyr': 2.5 * msiemens / cmetre2,
                 'g_Kdr_pyr': 80. * msiemens / cmetre2,
                 'g_Na_pyr': 120. * msiemens / cmetre2}}
```

If you have a `SpatialNeuron` `neuron` with equations for all the channels in the NML file (e.g. as generated by the `get_channel_equations` method below), then you can assign the reversal potentials and conductances to the individual compartments by combining the information from the `segment_groups` and `channel_properties` dictionaries:

```
for segment_group, values in nml_object.channel_properties.items():
    indices = nml_object.segment_groups[segment_group]
    for property, value in values.items():
        getattr(neuron, property)[indices] = value
```

To get the channel equations for a particular channel, e.g. `Na_pyr`:

```
>>> print(nml_object.get_channel_equations("Na_pyr"))
alpha_h_Na_pyr = (128. * hertz) * exp((v - (-43. * mvolt))/(-18. * mvolt)) : Hz
beta_h_Na_pyr = (4. * khertz) / (1 + exp(0 - (v - (-20. * mvolt))/(5. * mvolt))) : Hz
I_Na_pyr = g_Na_pyr*m_Na_pyr**2*h_Na_pyr*(E_Na_pyr - v) : A/(m^2)
alpha_m_Na_pyr = (1.28 * khertz) * (v - (-46.9 * mvolt)) / (4. * mvolt) / (1 - exp(-␣
↪(v - (-46.9 * mvolt)) / (4. * mvolt))) : Hz
beta_m_Na_pyr = (1.4 * khertz) * (v - (-19.9 * mvolt)) / (-5. * mvolt) / (1 - exp(-␣
↪(v - (-19.9 * mvolt)) / (-5. * mvolt))) : Hz
```

(continues on next page)

(continued from previous page)

```
dh_Na_pyr/dt = alpha_h_Na_pyr*(1-h_Na_pyr) - beta_h_Na_pyr*h_Na_pyr : 1
dm_Na_pyr/dt = alpha_m_Na_pyr*(1-m_Na_pyr) - beta_m_Na_pyr*m_Na_pyr : 1
E_Na_pyr : V (constant)
g_Na_pyr : S/(m^2) (constant)
```

Note that this mechanism currently only supports passive channels and HH-type channels with sigmoidal, exponential, or exponential linear equations for the gating variables.

Note: If your `.nml file` includes other .nml files, make sure they are present in the same folder as your main .nml file. If the files are not present, a warning will be thrown and execution will proceed as normal.

1.2.5 Base exporter

This is the user documentation of the *baseexport* package, that provides functionality to represent Brian2 models in a standard dictionary format. The standard dictionary has a simple and easy to access hierarchy of model information, that can be used for various model description exporters and custom descriptions.

The *baseexport* package is not meant to use directly but to provide general framework to use other model description exporters on top of it. However, the standard dictionary can be easily accessed as mentioned in the *Working Example* section.

Overview

Working example

Once the `device.build()` is called, the standard dictionary can be accessed by `device.runs` variable. As a working example, let us take a simple unconnected Integrate & Fire neuronal model with refractoriness and initializations,

```
from brian2 import *
import brian2tools.baseexport
import pprint    # to pretty print dictionary

set_device('exporter')    # set device mode

n = 100
duration = 1*second
tau = 10*ms
v_th = 1 * volt

eqn = '''
dv/dt = (v_rest - v) / tau : volt (unless refractory)
v_rest : volt
'''
group = NeuronGroup(n, eqn, threshold='v > v_th', reset='v = v_rest',
                    refractory=5*ms, method='euler')
```

(continues on next page)

(continued from previous page)

```
group.v = 0*mV
group.v_rest = 'rand() * 20*mV * i / (N-1)'

statemonitor = StateMonitor(group, 'v', record=True)
spikemonitor = SpikeMonitor(group, record=0)

run(duration)

pprint.pprint(device.runs)    # print standard dictionary
```

The output standard dictionary would look similar to,

```
[{'components': {'neurongroup': [{'N': 100,
                                  'equations': {'v': {'expr': '(v_rest - v) / tau',
                                                      'flags': ['unless refractory'],
                                                      'type': 'differential equation',
                                                      'unit': volt,
                                                      'var_type': 'float'},
                                                 'v_rest': {'type': 'parameter',
                                                            'unit': volt,
                                                            'var_type': 'float'}},
                                  'events': {'spike': {'refractory': 5. * msecond,
                                                       'reset': {'code': 'v = v_rest',
                                                                 'dt': 100. * usecond,
                                                                 'order': 0,
                                                                 'when': 'resets'},
                                                       'threshold': {'code': 'v > v_th',
                                                                     'dt': 100. *␣
↪usecond,
                                                                     'order': 0,
                                                                     'when':
↪'thresholds'}}},
                                  'identifiers': {'tau': 10. * msecond,
                                                  'v_th': 1. * volt},
                                  'name': 'neurongroup',
                                  'order': 0,
                                  'user_method': 'euler',
                                  'when': 'groups'}],
                'spikemonitor': [{'dt': 100. * usecond,
                                  'event': 'spike',
                                  'name': 'spikemonitor',
                                  'order': 1,
                                  'record': 0,
                                  'source': 'neurongroup',
                                  'source_size': 100,
                                  'variables': [],
                                  'when': 'thresholds'}],
                'statemonitor': [{'dt': 100. * usecond,
                                  'n_indices': 100,
                                  'name': 'statemonitor',
                                  'order': 0,
                                  'record': True,
                                  'source': 'neurongroup',
                                  'variables': ['v'],
                                  'when': 'start'}]},
'duration': 1. * second,
'initializers_connectors': [{'index': True,
```

(continues on next page)

(continued from previous page)

```
                                'source': 'neurongroup',
                                'type': 'initializer',
                                'value': 0. * volt,
                                'variable': 'v'},
                                {'identifiers': {'N': 100},
                                'index': 'True',
                                'source': 'neurongroup',
                                'type': 'initializer',
                                'value': 'rand() * 20*mV * i / (N-1)',
                                'variable': 'v_rest'}]}]
```

To the user side, the changes required to use the exporter are very minimal (very similar to accessing other Brian2 device modes). In the standard Brian code, adding *baseexport* import statement and setting device `exporter` with proper `build_options` will be sufficient to use the exporter. To print the dictionary in `stdout`, `debug` option shall also be used, apart from using `device.runs` variable. The changes required to run in `debug` mode for the above example are,

```
from brian2 import *
import brian2tools.baseexport

set_device('exporter', debug=True)    # build in debug mode to print out dictionary

. . . .

run(duration)
```

Most of the standard dictionary items have the same object type as in Brian2. For instance, `identifiers` and `dt` fields have values of type `Quantity` but `N` (population size) is of type `int`.

Limitations

The Base export currently supports almost all Brian2 features except,

- Multicompartmental neurons (`SpatialNeuron`)
- `store`/`restore` mechanism
- Multiple `Network` objects

1.2.6 Markdown exporter

This is the user documentation of *mdexport* package, that provides functionality to describe Brian2 models in Markdown. The markdown description provides human-readable information of Brian components defined. In background, the exporter uses the *Base exporter* to collect information from the run time and expand them to markdown strings.

Overview

Working example

As a quick start to use the package, let us take a simple model with adaptive threshold that increases with each spike and write the markdown output in a file

```
from brian2 import *
import brian2tools.mdexport
# set device 'markdown'
set_device('markdown', filename='model_description')

eqs = '''
dv/dt = -v/(10*ms) : volt
dvt/dt = (10*mV-vt)/(15*ms) : volt
'''

reset = '''
v = 0*mV
vt += 3*mV
'''

IF = NeuronGroup(1, model=eqs, reset=reset, threshold='v>vt',
                 method='exact')
IF.vt = 10*mV
PG = PoissonGroup(1, 500 * Hz)

C = Synapses(PG, IF, on_pre='v += 3*mV')
C.connect()

Mv = StateMonitor(IF, 'v', record=True)
Mvt = StateMonitor(IF, 'vt', record=True)
# Record the value of v when the threshold is crossed
M_crossings = SpikeMonitor(IF, variables='v')
run(2*second)
```

The rendered file `model_description.md` would look like,

Note: By default, Monitors are not included in markdown output, and the order of variable initializations and `connect` statements are not shown but rather included with the respective objects. However, these default options shall be changed according to one's need (see developer documentation of *Markdown exporter* for how to change the default options).

Similar to other Brian2 device modes, to inform Brian to run in the exporter mode, the minimal changes required are importing the package and mentioning device `markdown` in `set_device`. The markdown output can be accessed from `device.md_text`.

The above example can also be run in `debug` mode to print the output in `stdout`. In that case, the changes to the above example are,

```
from brian2 import *
import brian2tools.mdexport
# set device 'markdown'
set_device('markdown', debug=True)  # to print the output in stdout
. . .

run(2*second)
```

Exporter specific build options

Various options (apart from that of `RuntimeDevice`) shall be passed to `set_device` or in `device.build()`. Exporter specific `build_options` are,

`expander` Expander is the object of the call that contains expander functions to get information from *`baseexport`* and use them to write markdown text. By default, *`MdExpander`* is used. The default argument values can be changed and expand functions can be overridden (see developer documentation of *Markdown exporter* for more details and how to write custom expander functions).

A small example to enable `github_md` in expander that specifies, whether rendered output should be non-Mathjax based (as compilers like GitHub)

```
from brian2tools.mdexport.expander import MdExpander
# change default value
custom_options = MdExpander(github_md=True)
set_device('markdown', expander=custom_options)  # pass the custom expander object
. . . .
```

`filename` Filename to write output markdown text. To use the same filename of the user script, `''` (empty string) shall be passed. By default, no file writing is done

Limitations

Since the package uses *`baseexport`* in the background, all the limitations applicable to *`baseexport`* applies here as well

1.3 Developer's guide

1.3.1 Coding guidelines

The coding style should mostly follow the Brian 2 guidelines, with one major exception: for *`brian2tools`* the code should be both Python 2 (for versions >= 2.7) and Python 3 compatible. This means for example to use `range` and not `xrange` for iteration or conversely use `list(range)` instead of just `range` when a list is required. For now, this works without `from __future__` imports or helper modules like `six` but the details of this will be fixed when the need arises.

1.3.2 NeuroML exporter

Overview

The main work of the exporter is done in the *lemsexport* module.

It consists of two main classes:

- `NMLExporter` - responsible for building the NeuroML2/LEMS model.
- `LEMSDevice` - responsible for code generation. It gathers all variables needed to describe the model and calls `NMLExporter` with well-prepared parameters.

NMLExporter

The whole process of building NeuroML model starts with calling the `create_lems_model` method. It selects crucial Brian 2 objects to further parse and pass them to respective methods.

```
if network is None:
    net = Network(collect(level=1))
else:
    net = network

if not constants_file:
    self._model.add(lems.Include(LEMS_CONSTANTS_XML))
else:
    self._model.add(lems.Include(constants_file))
includes = set(includes)
for incl in INCLUDES:
    includes.add(incl)
neuron_groups  = [o for o in net.objects if type(o) is NeuronGroup]
state_monitors = [o for o in net.objects if type(o) is StateMonitor]
spike_monitors = [o for o in net.objects if type(o) is SpikeMonitor]

for o in net.objects:
    if type(o) not in [NeuronGroup, StateMonitor, SpikeMonitor,
                       Thresholder, Resetter, StateUpdater]:
        logger.warn("""{} export functionality
                       is not implemented yet.""".format(type(o).__name__))

# Thresholder, Resetter, StateUpdater are not interesting from our perspective
if len(netinputs)>0:
    includes.add(LEMS_INPUTS)
for incl in includes:
    self.add_include(incl)
# First step is to add individual neuron deifinitions and initialize
# them by MultiInstantiate
for e, obj in enumerate(neuron_groups):
    self.add_neurongroup(obj, e, namespace, initializers)
```

Neuron Group

A method `add_neurongroup` requires more attention. This is the method responsible for building cell model in LEMS (as so-called `ComponentType`) and initializing it when necessary.

In order to build a whole network of cells with different initial values, we need to use the `MultiInstantiate` LEMS tag. A method `make_multiinstantiate` does this job. It iterates over all parameters and analyses

equation to find those with iterator variable `i`. Such variables are initialized in a `MultiInstantiate` loop at the beginning of a simulation.

More details about the methods described above can be found in the code comments.

DOM structure

Until this point the whole model is stored in `NMLExporter._model`, because the method `add_neurongroup` takes advantage of `pylems` module to create a XML structure. After that we export it to `self._dommodel` and rather use NeuroML2 specific content. To extend it one may use `self._extend_dommodel()` method, giving as parameter a proper DOM structure (built for instance using python `xml.dom.minidom`).

```
# DOM structure of the whole model is constructed below
self._dommodel = self._model.export_to_dom()
# input support - currently only Poisson Inputs
for e, obj in enumerate(netinputs):
    self.add_input(obj, counter=e)
# A population should be created in *make_multiinstantiate*
# so we can add it to our DOM structure.
if self._population:
    self._extend_dommodel(self._population)
# if some State or Spike Monitors occur we support them by
# Simulation tag
self._model_namespace['simulname'] = "sim1"
self._simulation = NeuroMLSimulation(self._model_namespace['simulname'],
                                     self._model_namespace['networkname'])
for e, obj in enumerate(state_monitors):
    self.add_statemonitor(obj, filename=recordingsname, outputfile=True)
for e, obj in enumerate(spike_monitors):
    self.add_spikemonitor(obj, filename=recordingsname)
```

Some of the NeuroML structures are already implemented in `supporting.py`. For example:

- `NeuroMLSimulation` - describes Simulation, adds plot and lines, adds outputfiles for spikes and voltage recordings;
- `NeuroMLSimpleNetwork` - creates a network of cells given some ComponentType;
- `NeuroMLTarget` - picks target for simulation runner.

At the end of the model parsing, a simulation tag is built and added with a target pointing to it.

```
simulation = self._simulation.build()
self._extend_dommodel(simulation)
target = NeuroMLTarget(self._model_namespace['simulname'])
target = target.build()
self._extend_dommodel(target)
```

You may access the final DOM structure by accessing the `model`` property or export it to a XML file by calling the `export_to_file()` method of the `NMLExporter` object.

Model namespace

In many places of the code a dictionary `self._model_namespace` is used. As LEMS used identifiers `id` to name almost all of its components, we want to be consistent in naming them. The dictionary stores names of model's components and allows to refer it later in the code.

LEMSDevice

`LEMSDevice` allows you to take advantage of Brian 2's code generation mechanism. It makes usage of the module easier, as it means for user that they just need to import `brian2tools.nmlexport` and set the device `neuroml2` like this:

```
import brian2lems.nmlexport

set_device('neuroml2', filename="ifcgmtest.xml")
```

In the class init a flag `self.build_on_run` was set to `True` which means that exporter starts working immediately after encountering the `run` statement.

```
def __init__(self):
    super(LEMSDevice, self).__init__()
    self.runs = []
    self.assignments = []
    self.build_on_run = True
    self.build_options = None
    self.has_been_run = False
```

First of all method `network_run` is called which gathers of necessary variables from the script or function namespaces and passes it to `build` method. In `build` we select all needed variables to separate dictionaries, create a name of the recording files and eventually build the exporter.

```
initializers = {}
for descriptions, duration, namespace, assignments in self.runs:
    for assignment in assignments:
        if not assignment[2] in initializers:
            initializers[assignment[2]] = assignment[-1]
if len(self.runs) > 1:
    raise NotImplementedError("Currently only single run is supported.")
if len(filename.split("."))!=1:
    filename_ = 'recording_' + filename.split(".")[0]
else:
    filename_ = 'recording_' + filename
exporter = NMLExporter()
exporter.create_lems_model(self.network, namespace=namespace,
                                         initializers=initializers,
                                         recordingsname=filename_)
exporter.export_to_file(filename)
```

LEMS Unit Constants

Last lines of the method are saving `LemsConstantUnit.xml` file alongside with our model file. This is due to the fact that in some places of mathematical expressions LEMS requires unitless variables, e.g. instead of `1 mm` it wants `0.001`. So we store most popular units transformed to constants in a separate file which is included in the model file header.

```
if lems_const_save:
    with open(os.path.join(nmlcdpath, LEMS_CONSTANTS_XML), 'r') as f:
        with open(LEMS_CONSTANTS_XML, 'w') as fout:
            fout.write(f.read())
```

Other modules

If you want to know more about other scripts included in package (*lemsrendering*, *supporting*, cgmhelper), please read their docstrings or comments included in the code.

TODO

- synapses support;

First attempt to make synapses export work was made during GSOC period. The problem with that feature is related to the fact that NeuroML and brian2 internal synapses implementation differs substantially. For instance, in NeuroML there are no predefined rules for connections, but user needs to explicitly define a synapse. Moreover, in Brian 2, for efficiency reasons, postsynaptic potentials are normally modeled in the post-synaptic cell (for linearly summating synapses, this is equivalent but much more efficient), whereas in NeuroML they are modeled as part of the synapse (simulation speed is not an issue here).

- network input support;

Although there are some classes supporting `PoissonInput` in the `supporting.py`, full functionality of input is still not provided, as it is stongly linked with above synapses problems.

1.3.3 Markdown exporter

This is the developer documentation for *mdexport* package, that provides information about understanding *baseexport*'s standard dictionary, standard markdown expander *MdExpander* and writing custom expand functions.

Overview

Standard dictionary

The package *baseexport* collects all the required Brian model information and arranges them in an intuitive way, that can potentially be used for various exporters and use cases. Therefore, understanding the representation will be helpful for further manipulation.

The dictionary contains a list of `run` dictionaries with each containing information about the particular run simulation.

```
# list of run dictionaries
[
    . . . .
    {   # run dictionary
        duration: <Quantity>,
        components: {
                            . . .
                    },
        initializers_connectors: [. . .],
        inactive: [ . . .]
```

(continues on next page)

(continued from previous page)

```
    },
    . . . .
]
```

Typically, a `run` dictionary has four fields,

- `duration`: simulation length
- `components`: dictionary of network components like `NeuronGroup`, `Synapses`, etc.
- `initializers_connectors`: list of initializers and synaptic connections
- `inactive`: list of components that were inactive for the particular run

All the Brian `Network` components that are under `components` field, have components like, `NeuronGroup`, `Synapses`, etc and would look like,

```
{
    'neurongroup': [. . . .],
    'poissongroup': [. . . .],
    'spikegeneratorgroup': [. . . .],
    'statemonitor': [. . . .],
    'synapses': [. . . .],
    . . . .
}
```

Each component field has a list of objects of that component defined in the run time. The dictionary representation of `NeuronGroup` and its similar types would look like,

```
neurongroup: [
    {
        'name': <name of the group>,
        'N': <population size>,
        'user_method': <integration method>,
        'equations': <model equations> {
            '<variable name>':{ 'unit': <unit>,
                                'type': <equation type>
                                'var_type': <variable dtype>
                                'expr': <expression>,
                                'flags': <list of flags>
            }
            . . .
        }
        'events': <events> {
            '<event_name>':{'threshold':{'code': <threshold code>,
                                         'when': <when slot>,
                                         'order': <order slot>,
                                         'dt': <clock dt>
                            },
                            'reset':{'code': <reset code>,
                                     'when': <when slot>,
                                     'order': <order slot>,
                                     'dt': <clock dt>
                            },
                            'refractory': <refractory period>
            }
            . . .
        }
```

(continues on next page)

(continued from previous page)

```
        'run_regularly': <run_regularly statements>
        [
            {
                'name': <name of run_regularly>
                'code': <statement>
                'dt': <run_regularly clock dt>
                'when': <when slot of run_regularly>
                'order': <order slot of run_regularly>
            }
            . . .
        ]
        'when': <when slot of group>,
        'order': <order slot of group>,
        'identifiers': {'<name>': <value>,
        . . .
        }
    }
]
```

Similarly, `StateMonitor` and its similar types are represented like,

```
statemonitor: [
    {
        'name': <name of the group>,
        'source': <name of source>,
        'variables': <list of monitored variables>,
        'record': <list of monitored members>,
        'dt': <time step>
        'when': <when slot of group>,
        'order': <order slot of group>,
    }
. . .
]
```

As `Synapses` has many similarity with `NeuronGroup`, the dictionary of the same also looks similar to it, however some of the `Synapses` specific fields are,

```
synapses: [
    {
        'name': <name of the synapses object>,
        'equations': <model equations> {
            '<variable name>':{ 'unit': <unit>,
                                'type': <equation type>
                                'var_type': <variable dtype>
                                'expr': <expression>,
                                'flags': <list of flags>
            }
            . . .
        }

        'summed_variables': <summed variables>
        [
            {
                'target': <name of target group>,
                'code': <variable name>,
                'name': <name of the summed variable>,
```

(continues on next page)

(continued from previous page)

```
                'dt': <time step>,
                'when': <when slot of run_regularly>,
                'order': <order slot of run_regularly>
            }
            . . .
        ]

        'pathways': <synaptic pathways>
        [
            {
                'prepost': <pre or post event>,
                'event': <event name>,
                'code': <variable name>,
                'source': <source group name>,
                'name': <name of the summed variable>,
                'clock': <time step>,
                'when': <when slot of run_regularly>,
                'order': <order slot of run_regularly>,
            }
            . . .
        ]
    }
]
```

Also, the `identifiers` takes into account of `TimedArray` and custom user functions. The `initializers_connectors` field contains a list of initializers and synaptic connections, and their structure would look like,

```
[
    {   <initializer>
        'source': <source group name>,
        'variable': <variable that is initialized>,
        'index': <indices that are affected>,
        'value': <value>, 'type': 'initializer'
    },
    . . .
    {   <connection>
        {'i': <i>, 'j': <j>,
        'probability': <probability of connection>,
        'n_connections': <number of connections>,
        'synapses': <name of the synapse>,
        'source': <source group name>,
        'target': <target group name>, 'type': 'connect'
    }
    . . .
]
```

As a working example, to get the standard dictionary of model description when using STDP example,

```
[{'components':
{'neurongroup': [{'N': 1,
                'equations': {'ge': {'expr': '-ge / taue',
                                     'type': 'differential equation',
                                     'unit': radian,
                                     'var_type': 'float'},
                              'v': {'expr': '(ge * (Ee-v) + El - v) / taum',
```

(continues on next page)

(continued from previous page)

```
                                    'type': 'differential equation',
                                    'unit': volt,
                                    'var_type': 'float'}},
                'events': {'spike': {'reset': {'code': 'v = vr',
                                               'dt': 100. * usecond,
                                               'order': 0,
                                               'when': 'resets'},
                                     'threshold': {'code': 'v>vt',
                                                   'dt': 100. * usecond,
                                                   'order': 0,
                                                   'when': 'thresholds'}}},
                'identifiers': {'Ee': 0. * volt,
                                'El': -74. * mvolt,
                                'taue': 5. * msecond,
                                'taum': 10. * msecond,
                                'vr': -60. * mvolt,
                                'vt': -54. * mvolt},
                'name': 'neurongroup',
                'order': 0,
                'user_method': 'euler',
                'when': 'groups'}],
 'poissongroup': [{'N': 1000,
                'name': 'poissongroup',
                'rates': 15. * hertz}],
 'spikemonitor': [{'dt': 100. * usecond,
                'event': 'spike',
                'name': 'spikemonitor',
                'order': 1,
                'record': True,
                'source': 'poissongroup',
                'variables': ['i', 't'],
                'when': 'thresholds'}],
 'statemonitor': [{'dt': 100. * usecond,
                'n_indices': 2,
                'name': 'statemonitor',
                'order': 0,
                'record': array([0, 1], dtype=int32),
                'source': 'synapses',
                'variables': ['w'],
                'when': 'start'}],
 'synapses': [{'equations': {'Apost': {'expr': '-Apost / taupost',
                                    'flags': ['event-driven'],
                                    'type': 'differential equation',
                                    'unit': radian,
                                    'var_type': 'float'},
                             'Apre': {'expr': '-Apre / taupre',
                                    'flags': ['event-driven'],
                                    'type': 'differential equation',
                                    'unit': radian,
                                    'var_type': 'float'},
                             'w': {'type': 'parameter',
                                 'unit': radian,
                                 'var_type': 'float'}},
            'identifiers': {'dApost': -0.000105,
                             'dApre': 0.0001,
                             'gmax': 0.01,
                             'taupost': 20. * msecond,
```

(continues on next page)

(continued from previous page)

```
                            'taupre': 20. * msecond},
             'name': 'synapses',
             'pathways': [{'clock': 100. * usecond,
                            'code': 'ge += w\n'
                                    'Apre += dApre\n'
                                    'w = clip(w + Apost, 0, gmax)',
                            'event': 'spike',
                            'name': 'synapses_pre',
                            'order': -1,
                            'prepost': 'pre',
                            'source': 'poissongroup',
                            'target': 'neurongroup',
                            'when': 'synapses'},
                           {'clock': 100. * usecond,
                            'code': 'Apost += dApost\n'
                                    'w = clip(w + Apre, 0, gmax)',
                            'event': 'spike',
                            'name': 'synapses_post',
                            'order': 1,
                            'prepost': 'post',
                            'source': 'neurongroup',
                            'target': 'poissongroup',
                            'when': 'synapses'}],
             'source': 'poissongroup',
             'target': 'neurongroup'}]},
'duration': 100. * second,
'initializers_connectors': [{'index': True,
                             'source': 'poissongroup',
                             'type': 'initializer',
                             'value': 15. * hertz,
                             'variable': 'rates'},
                            {'n_connections': 1,
                             'probability': 1,
                             'source': 'poissongroup',
                             'synapses': 'synapses',
                             'target': 'neurongroup',
                             'type': 'connect'},
                            {'identifiers': {'gmax': 0.01},
                             'index': 'True',
                             'source': 'synapses',
                             'type': 'initializer',
                             'value': 'rand() * gmax',
                             'variable': 'w'}]}]
```

MdExpander

To use the dictionary representation for creating markdown strings, by default *MdExpander* class is used. The class contains expand functions for different Brian components, such that the user can easily override the particular function without affecting others. Also, different options can be given during the instantiation of the object and pass to the `set_device` or `device.build()`.

As a simple example, to use GitHub based markdown rendering for mathematical statements, and use Brian specific jargons,

```
from brian2tools import MdExpander  # import the standard expander
# custom expander
custom = MdExpander(github_md=True, brian_verbose=True)
set_device('markdown', expander=custom)  # pass the custom expander
```

Details about the monitors are not included by default in the output markdown and to include them,

```
# custom expander to include monitors
custom_with_monitors = MdExpander(include_monitors=True)
set_device('markdown', expander=custom_with_monitors)
```

Also, the order of variable initializations and `connect` statements are not shown in the markdown output by default, this may likely result to inaccurate results, when the values of variables during synaptic connections are contingent upon their order. In that case, the order shall be included to markdown output as,

```
# custom expander to include monitors
custom = MdExpander(keep_initializer_order=True)
set_device('markdown', expander=custom)
```

The modified output with details about the order of initialization and Synaptic connection, when running on the *Working example* would look like,

Similarly, `author` and `add_meta` options can also be customized during object instantiation, to add author name and meta data respectively in the header of the markdown output.

Typically, expand function of the component would follow the structure similar to,

```
def expand_object(self, object_dict):
    # use object_dict information to write md_string
    md_string = . . . object_dict['field_A']
    return md_string
```

However, enumerating components like `identifiers`, `pathways` have two functions in which the first one simply loops the list and the second one expands the member. For example, with `identifiers`,

```
def expand_identifiers(self, identifiers_list):
    # calls `expand_identifier` iteratively
    markdown_str = ''
    for identifier in identifiers_list:
        . . .
        markdown_str += self.expand_identifier(identifier)
    return markdown_str

def expand_identifier(self, identifier):
    # individual identifier expander
    markdown_str = ''
    . . . # use identifier dict to write markdown strings
    return markdown_str
```

All the individual expand functions are tied to *`create_md_string`* function that calls and collects all the returned markdown strings to pass it to `device.md_text`

Writing custom expand class

With the understanding of standard dictionary representation and default markdown expand class (*`MdExpander`*), writing custom expand class becomes very straightforward. As a working example, the custom expander class to write equations in a table like format,

```
from brian2tools import MdExpander
from markdown_strings import table  # import table from markdown_strings

# custom expander class to do custom modifications for model equations
class Dynamics_table(MdExpander):

    def expand_equation(self, var, equation):
        # if differential equation pass `differential` flag as `True` to
        # render_expression()
        if equation['type'] == 'differential equation':
            return (self.render_expression(var, differential=True) +
                        '=' + self.render_expression(equation['expr']))
        else:
            return (self.render_expression(var) +
                        '=' + self.render_expression(equation['expr']))

    def expand_equations(self, equations):
        diff_rend_eqn = ['Differential equations']
        sub_rend_eqn = ['Sub-Expressions']
        # loop over
        for (var, eqn) in equations.items():
            if eqn['type'] == 'differential equation':
                diff_rend_eqn.append(self.expand_equation(var, eqn))
            if eqn['type'] == 'subexpression':
                sub_rend_eqn.append(self.expand_equation(var, eqn))

        # now pad space for shorter one
        if len(diff_rend_eqn) > len(sub_rend_eqn):
            shorter = diff_rend_eqn
            longer = sub_rend_eqn
        else:
            shorter = sub_rend_eqn
            longer = diff_rend_eqn
        for _ in range(len(longer) - len(shorter)):
            shorter.append('')

        # return table of rendered equations
        return table([shorter, longer])

custom = Dynamics_table()
set_device('markdown', expander=custom)  # pass the custom expander object
```

when using the above custom class with COBAHH example, the equation part would look like,

1.3.4 Release procedure

In *brian2tools* we use the setuptools_scm package to set the package version information, the basic release procedure therefore consists of setting a git tag and pushing that tag to github.

The `dev/release/prepare_release.py` script automates the tag creation and makes sure that no uncommited changes exist when doing do.

In the future, we will probably also push the pypi packages automatically from the test builds; for now this has to be done manually. The `prepare_release.py` script mentioned above will already create the source distribution and universal wheel files, they can then be uploaded with `twine upload dist/*` or using the `dev/release/upload_to_pypi.py` script.

CHAPTER 2

API reference

2.1 brian2tools package

Tools for use with the Brian 2 simulator.

2.1.1 Subpackages

brian2tools.baseexport package

stdformatexport package to export Brian models to standard representation

Submodules

brian2tools.baseexport.collector module

The file contains simple functions to collect information from BrianObjects and represent them in a standard dictionary format. The parts of the file shall be reused with standard format exporter.

`brian2tools.baseexport.collector.`**`collect_Equations`**(*equations*)
Collect model equations of the NeuronGroup

Parameters **`equations`** (*`brian2.equations.equations.Equations`*) – model equations object

Returns **eqn_dict** – Dictionary with extracted information

Return type dict

`brian2tools.baseexport.collector.`**`collect_EventMonitor`**(*event_mon*)
Collect details of `EventMonitor` class and return them in dictionary format

Parameters **`event_mon`** (*`brian2.EventMonitor`*) – EventMonitor object

Returns **event_mon_dict** – Dictionary representation of the collected details

Return type dict

brian2tools.baseexport.collector.**collect_Events**(*group*)
Collect Events (spiking) of the NeuronGroup

Parameters **group** (*brian2.groups.neurongroup.NeuronGroup*) – NeuronGroup object

Returns

- **event_dict** (*dict*) – Dictionary with extracted information
- **event_identifiers** (*set*) – Set of identifiers related to events

brian2tools.baseexport.collector.**collect_NeuronGroup**(*group*, *run_namespace*)
Collect information from brian2.groups.neurongroup.NeuronGroup and return them in a dictionary format

Parameters

- **group** (*brian2.groups.neurongroup.NeuronGroup*) – NeuronGroup object
- **run_namespace** (*dict*) – Namespace dictionary

Returns **neuron_dict** – Dictionary with extracted information

Return type dict

brian2tools.baseexport.collector.**collect_PoissonGroup**(*poisson_grp*, *run_namespace*)
Extract information from 'brian2.input.poissongroup.PoissonGroup' and represent them in a dictionary format

Parameters

- **poisson_grp** (*brian2.input.poissongroup.PoissonGroup*) – PoissonGroup object
- **run_namespace** (*dict*) – Namespace dictionary

Returns **poisson_grp_dict** – Dictionary with extracted information

Return type dict

brian2tools.baseexport.collector.**collect_PoissonInput**(*poinp*, *run_namespace*)
Collect details of PoissonInput and represent them in dictionary

Parameters

- **poinp** (*brian2.input.poissoninput.PoissonInput*) – PoissonInput object
- **run_namespace** (*dict*) – Namespace dictionary

Returns **poinp_dict** – Dictionary representation of the collected details

Return type dict

brian2tools.baseexport.collector.**collect_PopulationRateMonitor**(*poprate_mon*)
Represent required details of PopulationRateMonitor in dictionary format

Parameters **poprate_mon** (*brian2.monitors.ratemonitor.PopulationRateMonitor*) – PopulationRateMonitor class object

Returns **poprate_mon_dict** – Dictionary format of the details collected

Return type dict

brian2tools.baseexport.collector.**collect_SpikeGenerator**(*spike_gen*, *run_namespace*)
Extract information from 'brian2.input.spikegeneratorgroup.SpikeGeneratorGroup'and represent them in a dictionary format

Parameters

- **spike_gen** (*brian2.input.spikegeneratorgroup.SpikeGeneratorGroup*) – SpikeGenerator object
- **run_namespace** (*dict*) – Namespace dictionary

Returns **spikegen_dict** – Dictionary with extracted information

Return type dict

brian2tools.baseexport.collector.**collect_SpikeMonitor**(*spike_mon*)
Collect details of brian2.monitors.spikemonitor.SpikeMonitor and return them in dictionary format

Parameters **spike_mon** (*brian2.monitors.spikemonitor.SpikeMonitor*) – SpikeMonitor object

Returns **spike_mon_dict** – Dictionary representation of the collected details

Return type dict

brian2tools.baseexport.collector.**collect_SpikeSource**(*source*)
Check SpikeSource and collect details

Parameters **source** (brian2.core.spikesource.SpikeSource) – SpikeSource object

brian2tools.baseexport.collector.**collect_StateMonitor**(*state_mon*)
Collect details of brian2.monitors.statemonitor.StateMonitor and return them in dictionary format

Parameters **state_mon** (*brian2.monitors.statemonitor.StateMonitor*) – StateMonitor object

Returns **state_mon_dict** – Dictionary representation of the collected details

Return type dict

brian2tools.baseexport.collector.**collect_Synapses**(*synapses*, *run_namespace*)
Collect information from brian2.synapses.synapses.Synapses and represent them in dictionary format

Parameters

- **synapses** (*brian2.synapses.synapses.Synapses*) – Synapses object
- **run_namespace** (*dict*) – Namespace dictionary

Returns **synapse_dict** – Standard dictionary format with collected information

Return type dict

brian2tools.baseexport.device module

brian2tools.baseexport.helper module

The file contains helper functions that shall be used for exporting standard representation format

brian2tools.mdexport package

Human-readable export from Brian script

Submodules

brian2tools.mdexport.expander module

Standard markdown expander class to expand Brian objects to markdown text using standard dictionary representation of baseexport

class brian2tools.mdexport.expander.**MdExpander**(*brian_verbose=False*, *include_monitors=False*, *keep_initializer_order=False*, *author=None*, *add_meta=False*, *github_md=False*)

Bases: object

Build Markdown texts from run dictionary. The class contain various expand functions for corresponding Brian objects and get standard dictionary as argument to expand them with sentences in markdown format.

Note: If suppose the user would like to change the format or wordings in the exported model descriptions, one can derive from this standard markdown expander class to override the required changes in expand functions.

---

check_plural(*iterable*, *singular_word=None*, *allow_constants=True*, *is_int=False*)

Function to attach plural form of the word by examining the following iterable

Parameters

- **iterable** (object with __iter__ attribute) – Object that has to be examined
- **singular_word** (*str, optional*) – Word whose plural form has to searched in singular_plural_dict
- **allow_constants** (*bool, optional*) – Whether to assume non iterable as singular, if set as True, the iterable argument must be an iterable
- **is_int** (*int, optional*) – Check whether number 1 is passed, if > 1 return plural, by default set as False. Note: allow_constants should be True

create_md_string(*net_dict*)

Create markdown text by checking the standard dictionary and call required expand functions and arrange the descriptions

expand_EventMonitor(*eventmon*)

Expand EventMonitor from standard representation

Parameters **eventmon** (*dict*) – Standard dictionary of EventMonitor

expand_NeuronGroup(*neurongrp*)

Expand NeuronGroup from standard dictionary

Parameters **neurongrp** (*dict*) – Standard dictionary of NeuronGroup

expand_PoissonGroup(*poisngrp*)

Expand PoissonGroup from standard dictionary

Parameters **poisngrp** (*dict*) – Standard dictionary of PoissonGroup

expand_PoissonInput(*poinp*)
Expand PoissonInput

Parameters **poinp** (*dict*) – Standard dictionary representation for PoissonInput

expand_PopulationRateMonitor(*popratemon*)
Expand PopulationRateMonitor

Parameters **popratemon** (*dict*) – PopulationRateMonitor's baseexport dictionary

expand_SpikeGeneratorGroup(*spkgen*)
Expand SpikeGeneratorGroup from standard dictionary

Parameters **spkgen** (*dict*) – Standard dictionary of SpikeGeneratorGroup

expand_SpikeMonitor(*spikemon*)
Expand SpikeMonitor from standard representation

Parameters **spikemon** (*dict*) – Standard dictionary of SpikeMonitor

expand_SpikeSource(*source*)
Check whether subgroup dict and expand accordingly

Parameters **source** (*str, dict*) – Source group name or subgroup dictionary

expand_StateMonitor(*statemon*)
Expand StateMonitor from standard dictionary

Parameters **statemon** (*dict*) – Standard dictionary of StateMonitor

expand_Synapses(*synapse*)
Expand `Synapses` details from Baseexporter dictionary

Parameters **synapse** (*dict*) – Dictionary representation of `Synapses` object

expand_connector(*connector*)
Expand synaptic connector from connector dictionary

Parameters **connector** (*dict*) – Dictionary representation of connector

expand_equation(*var*, *equation*)
Expand Equation from equation dictionary

Parameters

- **var** (*str*) – Variable name
- **equation** (*dict*) – Details of the equation

expand_equations(*equations*)
Function to loop all equations

expand_event(*event_name*, *event_details*)
Function to expand event dictionary

Parameters

- **event_name** (*str*) – name of the event
- **event_details** (*dict*) – details of the event

expand_events(*events*)
Expand function to loop through all events and call *expand_event*

expand_identifier(*ident_key*, *ident_value*)
Expand identifer (key-value form)

Parameters

- **ident_key** (*str*) – Identifier name
- **ident_value** (*Quantity, str, dict*) – Identifier value. Dictionary if identifer is of type either `TimedArray` or custom function

expand_identifiers(*identifiers*)
Expand function to loop through identifiers and call *expand_identifier*

expand_initializer(*initializer*)
Expand initializer from initializer dictionary

Parameters **initializer** (*dict*) – Dictionary representation of initializer

expand_network_header(*net_dict*)
Expand function to write network header

expand_pathway(*pathway*)
Expand `SynapticPathway`

Parameters **pathway** (*dict*) – SynapticPathway's baseexport dictionary

expand_pathways(*pathways*)
Loop through pathways and call *expand_pathway*

expand_run_header(*run_dict*, *run_indx*, *single_run=False*)
Expand run() header

Parameters

- **run_dict** (*dict*) – run dictionary
- **run_indx** (*int*) – Index of run
- **single_run** (*bool, optional*) – Whether only single run() defined for the network

expand_runregularly(*run_reg*)
Expand run_regularly from standard dictionary

Parameters **run_reg** (*dict*) – Standard dictionary representation for run_regularly()

expand_summed_variable(*sum_variable*)
Expand Summed variable

Parameters **sum_variabe** (*dict*) – SummedVariable's baseexport dictionary

expand_summed_variables(*sum_variables*)
Loop through summed variables and call *expand_summed_variable*

prepare_array(*arr*, *threshold=10*, *precision=2*)
Prepare arrays using `numpy.array2string`

Parameters

- **arr** (`numpy.ndarray`) – Numpy array to prepare
- **threshold** (*int, optional*) – Threshold value to print all members
- **precision** (*int, optional*) – Floating point precision

prepare_math_statements(*statements*, *differential=False*, *separate=False*, *equals='←'*)
Prepare statements to render in markdown format

Parameters

- **statements** (*str*) – String containing mathematical equations and statements

- **differential** (*bool, optional*) – Whether should be treated as variable in differential equation
- **separate** (*bool, optional*) – Whether lhs and rhs of the statement should be separated and rendered
- **equals** (*str, optional*) – Equals operator, by default arrow from right to left

render_expression(*expression*, *differential=False*)
Function to render mathematical expression using `sympy.printing.latex`

Parameters

- **expression** (*str, Quantity*) – Expression that has to rendered
- **differential** (*bool, optional*) – Whether should be treated as variable in differential equation

Returns **rend_exp** – Markdown text for the expression

Return type str

brian2tools.mdexport.mdexporter module

brian2tools.nmlexport package

Submodules

brian2tools.nmlexport.lemsexport module

brian2tools.nmlexport.lemsrendering module

brian2tools.nmlexport.supporting module

class brian2tools.nmlexport.supporting.**NeuroMLPoissonGenerator**(*poissid*, *average_rate*)
Bases: object

Makes XML of spikeGeneratorPoisson for NeuroML2/LEMS simulation.

build()
Builds NeuroML DOM structure of spikeGeneratorPoisson and returns it.

Returns **generator** – DOM representation of generator.

Return type xml.minidom.dom

class brian2tools.nmlexport.supporting.**NeuroMLSimpleNetwork**(*id_*)
Bases: object

NeuroMLSimpleNetwork class representing network tag in NeuroML syntax as a XML DOM representation.

add_component(*id_*, *type_*, ***attributes*)
Adds a component to a network.

Parameters

- **id** (*str*) – component id
- **type** (*str*) – type of component

- **attributes** (*.., optional*) – more attributes

build()
Builds NeuroML DOM structure of network. It returns DOM object.

Returns **network** – DOM representation of network.

Return type xml.minidom.dom

class brian2tools.nmlexport.supporting.**NeuroMLSimulation**(*simid*, *target*, *length='1s'*, *step='0.1ms'*)
Bases: object

NeuroMLSimulation class representing Simulation tag in NeuroML syntax as a XML DOM representation.

add_display(*dispid*, *title=''*, *time_scale='1ms'*, *xmin='0'*, *xmax='1000'*, *ymin='0'*, *ymax='11'*)
Adds a Display element to Simulation.

Parameters

- **dispid** (*str*) – display id
- **title** (*str*) – title printed on display window
- **time_scale** (*str*) – time scale of a plot
- **xmax, ymin, ymax** (*xmin,*) – limits of plot

add_eventoutputfile(*outfileid*, *filename='recordings.spikes'*, *format_='TIME_ID'*)
Adds an EventOutputFile element to a recently added Display.

Parameters

- **outfileid** (*str*) – EventOutputFile id
- **filename** (*str*) – name of an output file
- **format** (*str, optional*) – format of data storage, default TIME_ID

add_eventselection(*esid*, *select*, *event_port='spike'*)
Adds an EventSelection element to a recently added EventOutputFile.

Parameters

- **esid** (*str*) – EventSelection id
- **select** (*str*) – index of selected neuron
- **event_port** (*str*) – event port name, default 'spike'

add_line(*linid*, *quantity*, *scale='1mV'*, *time_scale='1ms'*)
Adds a Line element to a recently added Display.

Parameters

- **linid** (*str*) – line id
- **quantity** (*str*) – measure to plot
- **scale** (*str*) – scale of a function
- **time_scale** (*str*) – time scale of a line

add_outputcolumn(*ocid*, *quantity*)
Adds an OutputColumn element to a recently added OutputFile tag.

Parameters

- **ocid** (*str*) – OutputColumn id
- **quantity** (*str*) – measure to store in a column

add_outputfile(*outfileid*, *filename='recordings.dat'*)
Adds an OutputFile to Simulation.

Parameters

- **outfileid** (*str*) – OutputFile id
- **filename** (*str*) – name of an output file

build()
Builds NeuroML DOM structure of Simulation. It returns DOM object or it can be accessed by *object.simulation*.

Returns **simulation** – DOM representation of simulation.

Return type xml.minidom.dom

create_simulation(*simid*, *target*, *length*, *step*)
Adds a Simulation element to DOM structure at init.

Parameters

- **simid** (*str*) – simulation id.
- **target** (*str*) – target NeuroML object: component or network
- **length** (*str, optional*) – length of simulation, default 1 sec
- **step** (*str, optional*) – step of integration, default 0.1 ms

update_simulation_attribute(*attr_name*, *attr_value*)
Updates simulation attributes.

Parameters

- **attr_name** (*str*) – attribute name
- **attr_value** (*str or int or float*) – attribute value

class brian2tools.nmlexport.supporting.**NeuroMLTarget**(*component*)
Bases: object

Makes XML of target of NeuroML2/LEMS simulation.

build()
Builds NeuroML DOM structure of target and returns it.

Returns **target** – DOM representation of target.

Return type xml.minidom.dom

brian2tools.nmlexport.supporting.**brian_unit_to_lems**(*valunit*)
Returns string representation of LEMS unit where * is between value and unit e.g. “20. mV” -> “20.*mV”.

Parameters **valunit** (Quantity or str) – text or brian2.Quantity representation of a value with unit

Returns **valstr** – string representation of LEMS unit

Return type str

brian2tools.nmlexport.supporting.**from_string**(*rep*)
Returns Quantity object from text representation of a value.

Parameters **rep** (str) – text representation of a value with unit

Returns **q** – Brian Quantity object

Return type Quantity

brian2tools.nmlexport.supporting.**read_nml_dims**(*nmlcdpath=''*)
Read from NeuroMLCoreDimensions.xml all supported by LEMS dimensions and store it as a Python dict with name as a key and Brian2 unit as value.

Parameters **nmlcdpath** (str, optional) – Path to 'NeuroMLCoreDimensions.xml'

Returns **lems_dimenssions** – Dictionary with LEMS dimensions.

Return type dict

brian2tools.nmlexport.supporting.**read_nml_units**(*nmlcdpath=''*)
Read from 'NeuroMLCoreDimensions.xml' all supported by LEMS units.

Parameters **nmlcdpath** (str, optional) – Path to 'NeuroMLCoreDimensions.xml'

Returns **lems_units** – List with LEMS units.

Return type list

brian2tools.nmlimport package

Submodules

brian2tools.nmlimport.helper module

brian2tools.nmlimport.helper.**formatter**(*datum*)

brian2tools.nmlimport.helper.**get_child_segments**(*segments*)

brian2tools.nmlimport.helper.**get_parent_segment**(*segment*, *segments*)

brian2tools.nmlimport.nml module

brian2tools.nmlutils package

Submodules

brian2tools.nmlutils.utils module

brian2tools.nmlutils.utils.**from_string**(*rep*)
Returns Quantity object from text representation of a value.

Parameters **rep** (str) – text representation of a value with unit

Returns **q** – Brian Quantity object

Return type Quantity

brian2tools.nmlutils.utils.**string_to_quantity**(*rep*)
Returns Quantity object from text representation of a value.

Parameters **rep** (str) – text representation of a value with unit

Returns **q** – Brian Quantity object

Return type Quantity

brian2tools.plotting package

Package containing plotting modules.

brian2tools.plotting.**brian_plot**(*brian_obj*, *axes=None*, ***kwds*)

Plot the data of the given object (e.g. a monitor). This function will call an adequate plotting function for the object, e.g. *plot_raster* for a SpikeMonitor. The plotting may apply heuristics to get a generally useful plot (e.g. for a PopulationRateMonitor, it will plot the rates smoothed with a Gaussian window of 1 ms), the exact details are subject to change. This function is therefore mostly meant as a quick and easy way to plot an object, for full control use one of the specific plotting functions.

Parameters

- **brian_obj** (*object*) – The Brian object to plot.
- **axes** (Axes, optional) – The Axes instance used for plotting. Defaults to None which means that a new Axes will be created for the plot.
- **kwds** (*dict, optional*) – Any additional keywords command will be handed over to matplotlib's plot command. This can be used to set plot properties such as the color.

Returns **axes** – The Axes instance that was used for plotting. This object allows to modify the plot further, e.g. by setting the plotted range, the axis labels, the plot title, etc.

Return type Axes

brian2tools.plotting.**add_background_pattern**(*axes*, *hatch='xxx'*, *fill=True*, *fc=(0.9, 0.9, 0.9)*, *ec=(0.8, 0.8, 0.8)*, *zorder=-10*, ***kwds*)

Add a "hatching" pattern to the background of the axes (can be useful to make a difference between "no value" and a value mapped to a color value that is identical to the background color). By default, it uses a cross hatching pattern in gray which can be changed by providing the respective arguments. All additional keyword arguments are passed on to the Rectangle initializer.

Parameters

- **axes** (matplotlib.axes.Axes) – The axes where the background pattern should be added.
- **hatch** (*str, optional*) – See matplotlib.patches.Patch.set_hatch. Defaults to 'xxx'.
- **fill** (*bool, optional*) – See matplotlib.patches.Patch.set_fill. Defaults to True.
- **fc** (*mpl color spec or None or 'none'*) – See matplotlib.patches.Patch.set_facecolor. Defaults to (0.9, 0.9, 0.9).
- **ec** (*mpl color spec or None or 'none'*) – See matplotlib.patches.Patch.set_edgecolor. Defaults to (0.8, 0.8, 0.8).
- **zorder** (*int*) – See matplotlib.artist.Artist.set_zorder. Defaults to -10.

brian2tools.plotting.**plot_raster**(*spike_indices*, *spike_times*, *time_unit=<Mock name='mock.ms' id='140104943437712'>*, *axes=None*, ***kwds*)

Plot a "raster plot", a plot of neuron indices over spike times. The default marker used for plotting is '.', it can be overriden with the marker keyword argument.

Parameters

- **spike_indices** (ndarray) – The indices of spiking neurons, corresponding to the times given in spike_times.
- **spike_times** (Quantity) – A sequence of spike times.
- **time_unit** (Unit, optional) – The unit to use for the time axis. Defaults to ms, but longer simulations could use second, for example.
- **axes** (Axes, optional) – The Axes instance used for plotting. Defaults to None which means that a new Axes will be created for the plot.
- **kwds** (*dict, optional*) – Any additional keywords command will be handed over to matplotlib's plot command. This can be used to set plot properties such as the color.

Returns **axes** – The Axes instance that was used for plotting. This object allows to modify the plot further, e.g. by setting the plotted range, the axis labels, the plot title, etc.

Return type Axes

brian2tools.plotting.**plot_state**(*times*, *values*, *time_unit=<Mock name='mock.ms' id='140104943437712'>*, *var_unit=None*, *var_name=None*, *axes=None*, ***kwds*)

Parameters

- **times** (Quantity) – The array of times for the data points given in values.
- **values** (Quantity, ndarray) – The values to plot, either a 1D array with the same length as times, or a 2D array with len(times) rows.
- **time_unit** (Unit, optional) – The unit to use for the time axis. Defaults to ms, but longer simulations could use second, for example.
- **var_unit** (Unit, optional) – The unit to use to plot the values (e.g. mV for a membrane potential). If none is given (the default), an attempt is made to find a good scale automatically based on the values.
- **var_name** (*str, optional*) – The name of the variable that is plotted. Used for the axis label.
- **axes** (Axes, optional) – The Axes instance used for plotting. Defaults to None which means that a new Axes will be created for the plot.
- **kwds** (*dict, optional*) – Any additional keywords command will be handed over to matplotlib's plot command. This can be used to set plot properties such as the color.

Returns **axes** – The Axes instance that was used for plotting. This object allows to modify the plot further, e.g. by setting the plotted range, the axis labels, the plot title, etc.

Return type Axes

brian2tools.plotting.**plot_rate**(*times*, *rate*, *time_unit=<Mock name='mock.ms' id='140104943437712'>*, *rate_unit=<Mock name='mock.Hz' id='140105029554832'>*, *axes=None*, ***kwds*)

Parameters

- **times** (Quantity) – The time points at which the rate is measured.
- **rate** (Quantity) – The population rate for each time point in times
- **time_unit** (Unit, optional) – The unit to use for the time axis. Defaults to ms, but longer simulations could use second, for example.
- **time_unit** – The unit to use for the rate axis. Defaults to Hz.

- **axes** (Axes, optional) – The Axes instance used for plotting. Defaults to None which means that a new Axes will be created for the plot.
- **kwds** (*dict, optional*) – Any additional keywords command will be handed over to matplotlib's plot command. This can be used to set plot properties such as the color.

Returns **axes** – The Axes instance that was used for plotting. This object allows to modify the plot further, e.g. by setting the plotted range, the axis labels, the plot title, etc.

Return type Axes

brian2tools.plotting.**plot_morphology**(*morphology*, *plot_3d=None*, *show_compartments=False*, *show_diameter=False*, *colors=('darkblue', 'darkred')*, *values=None*, *value_norm=(None, None)*, *value_colormap='hot'*, *value_colorbar=True*, *value_unit=None*, *axes=None*)

Plot a given Morphology in 2D or 3D.

Parameters

- **morphology** (Morphology) – The morphology to plot
- **plot_3d** (*bool, optional*) – Whether to plot the morphology in 3D or in 2D. If not set (the default) a morphology where all z values are 0 is plotted in 2D, otherwise it is plot in 3D.
- **show_compartments** (*bool, optional*) – Whether to plot a dot at the center of each compartment. Defaults to False.
- **show_diameter** (*bool, optional*) – Whether to plot the compartments with the diameter given in the morphology. Defaults to False.
- **colors** (*sequence of color specifications*) – A list of colors that is cycled through for each new section. Can be any color specification that matplotlib understands (e.g. a string such as 'darkblue' or a tuple such as (0, 0.7, 0).
- **values** (*Quantity, optional*) – Values to fill compartment patches with a color that corresponds to their given value.
- **value_norm** (*tuple or callable, optional*) – Normalization function to scale the displayed values. Can be a tuple of a minimum and a maximum value (where either of them can be None to denote taking the minimum/maximum from the data) or a function that takes a value and returns the scaled value (e.g. as returned by matplotlib.colors.PowerNorm). For a tuple of values, will use matplotlib.colors.Normalize```(vmin, vmax, clip=True)``` with the given (vmin, vmax) values.
- **value_colormap** (*str or matplotlib.colors.Colormap, optional*) – Desired colormap for plots. Either the name of a standard colormap or a matplotlib.colors.Colormap instance. Defaults to 'hot'. Note that this uses matplotlib color maps even for 3D plots with Mayavi.
- **value_colorbar** (*bool or dict, optional*) – Whether to add a colorbar for the values. Defaults to True, but will be ignored if no values are provided. Can also be a dictionary with the keyword arguments for matplotlib's colorbar method (2D plot), or for Mayavi's scalarbar method (3D plot).
- **value_unit** (Unit, optional) – A Unit to rescale the values for display in the colorbar. Does not have any visible effect if no colorbar is used. If not specified, will try to determine the "best unit" to itself.

- **axes** (Axes or Scene, optional) – A matplotlib Axes (for 2D plots) or mayavi Scene (for 3D plots) instance, where the plot will be added.

Returns **axes** – The Axes or Scene instance that was used for plotting. This object allows to modify the plot further, e.g. by setting the plotted range, the axis labels, the plot title, etc.

Return type Axes or Scene

brian2tools.plotting.**plot_dendrogram**(*morphology*, *axes=None*)

Plot a "dendrogram" of a morphology, i.e. an abstract representation which visualizes the branching structure and the length of each section.

Parameters

- **morphology** (Morphology) – The morphology to visualize.
- **axes** (Axes, optional) – The Axes instance used for plotting. Defaults to None which means that a new Axes will be created for the plot.

Returns **axes** – The Axes instance that was used for plotting. This object allows to modify the plot further, e.g. by setting the plotted range, the axis labels, the plot title, etc.

Return type Axes

brian2tools.plotting.**plot_synapses**(*sources*, *targets*, *values=None*, *var_unit=None*, *var_name=None*, *plot_type='scatter'*, *axes=None*, ***kwds*)

Parameters

- **sources** (ndarray of int) – The source indices of the connections (as returned by Synapses.i).
- **targets** (ndarray of int) – The target indices of the connections (as returned by Synapses.j).
- **values** (Quantity, ndarray) – The values to plot, a 1D array of the same size as sources and targets.
- **var_unit** (Unit, optional) – The unit to use to plot the values (e.g. mV for a membrane potential). If none is given (the default), an attempt is made to find a good scale automatically based on the values.
- **var_name** (*str, optional*) – The name of the variable that is plotted. Used for the axis label.
- **plot_type** ({'scatter', 'image', 'hexbin'}, optional) – What type of plot to use. Can be 'scatter' (the default) to draw a scatter plot, 'image' to display the connections as a matrix or 'hexbin' to display a 2D histogram using matplotlib's hexbin function. For a large number of synapses, 'scatter' will be very slow. Similarly, an 'image' plot will use a lot of memory for connections between two large groups. For a small number of neurons and synapses, 'hexbin' will be hard to interpret.
- **axes** (Axes, optional) – The Axes instance used for plotting. Defaults to None which means that a new Axes will be created for the plot.
- **kwds** (*dict, optional*) – Any additional keywords command will be handed over to the respective matplotlib command (scatter if the plot_type is 'scatter', imshow for 'image', and hexbin for 'hexbin'). This can be used to set plot properties such as the marker.

Returns **axes** – The Axes instance that was used for plotting. This object allows to modify the plot further, e.g. by setting the plotted range, the axis labels, the plot title, etc.

Return type Axes

Submodules

brian2tools.plotting.base module

Base module for the plotting facilities.

brian2tools.plotting.base.**brian_plot**(*brian_obj*, *axes=None*, ***kwds*)

Plot the data of the given object (e.g. a monitor). This function will call an adequate plotting function for the object, e.g. `plot_raster` for a `SpikeMonitor`. The plotting may apply heuristics to get a generally useful plot (e.g. for a `PopulationRateMonitor`, it will plot the rates smoothed with a Gaussian window of 1 ms), the exact details are subject to change. This function is therefore mostly meant as a quick and easy way to plot an object, for full control use one of the specific plotting functions.

Parameters

- **brian_obj** (*object*) – The Brian object to plot.
- **axes** (Axes, optional) – The `Axes` instance used for plotting. Defaults to `None` which means that a new `Axes` will be created for the plot.
- **kwds** (*dict, optional*) – Any additional keywords command will be handed over to matplotlib's `plot` command. This can be used to set plot properties such as the `color`.

Returns **axes** – The `Axes` instance that was used for plotting. This object allows to modify the plot further, e.g. by setting the plotted range, the axis labels, the plot title, etc.

Return type Axes

brian2tools.plotting.base.**add_background_pattern**(*axes*, *hatch='xxx'*, *fill=True*, *fc=(0.9, 0.9, 0.9)*, *ec=(0.8, 0.8, 0.8)*, *zorder=-10*, ***kwds*)

Add a "hatching" pattern to the background of the axes (can be useful to make a difference between "no value" and a value mapped to a color value that is identical to the background color). By default, it uses a cross hatching pattern in gray which can be changed by providing the respective arguments. All additional keyword arguments are passed on to the `Rectangle` initializer.

Parameters

- **axes** (matplotlib.axes.Axes) – The axes where the background pattern should be added.
- **hatch** (*str, optional*) – See `matplotlib.patches.Patch.set_hatch`. Defaults to `'xxx'`.
- **fill** (*bool, optional*) – See `matplotlib.patches.Patch.set_fill`. Defaults to `True`.
- **fc** (*mpl color spec or None or 'none'*) – See `matplotlib.patches.Patch.set_facecolor`. Defaults to `(0.9, 0.9, 0.9)`.
- **ec** (*mpl color spec or None or 'none'*) – See `matplotlib.patches.Patch.set_edgecolor`. Defaults to `(0.8, 0.8, 0.8)`.
- **zorder** (*int*) – See `matplotlib.artist.Artist.set_zorder`. Defaults to `-10`.

brian2tools.plotting.data module

Module to plot simulation data (raster plots, etc.)

brian2tools.plotting.data.**plot_raster**(*spike_indices*, *spike_times*, *time_unit=<Mock name='mock.ms' id='140104943437712'>*, *axes=None*, ***kwds*)

Plot a "raster plot", a plot of neuron indices over spike times. The default marker used for plotting is `'.'`, it can be overriden with the `marker` keyword argument.

Parameters

- **spike_indices** (ndarray) – The indices of spiking neurons, corresponding to the times given in `spike_times`.
- **spike_times** (Quantity) – A sequence of spike times.
- **time_unit** (Unit, optional) – The unit to use for the time axis. Defaults to `ms`, but longer simulations could use `second`, for example.
- **axes** (Axes, optional) – The `Axes` instance used for plotting. Defaults to `None` which means that a new `Axes` will be created for the plot.
- **kwds** (*dict, optional*) – Any additional keywords command will be handed over to matplotlib's `plot` command. This can be used to set plot properties such as the `color`.

Returns **axes** – The `Axes` instance that was used for plotting. This object allows to modify the plot further, e.g. by setting the plotted range, the axis labels, the plot title, etc.

Return type `Axes`

brian2tools.plotting.data.**plot_state**(*times*, *values*, *time_unit=<Mock name='mock.ms' id='140104943437712'>*, *var_unit=None*, *var_name=None*, *axes=None*, ***kwds*)

Parameters

- **times** (Quantity) – The array of times for the data points given in `values`.
- **values** (Quantity, ndarray) – The values to plot, either a 1D array with the same length as `times`, or a 2D array with `len(times)` rows.
- **time_unit** (Unit, optional) – The unit to use for the time axis. Defaults to `ms`, but longer simulations could use `second`, for example.
- **var_unit** (Unit, optional) – The unit to use to plot the `values` (e.g. `mV` for a membrane potential). If none is given (the default), an attempt is made to find a good scale automatically based on the `values`.
- **var_name** (*str, optional*) – The name of the variable that is plotted. Used for the axis label.
- **axes** (Axes, optional) – The `Axes` instance used for plotting. Defaults to `None` which means that a new `Axes` will be created for the plot.
- **kwds** (*dict, optional*) – Any additional keywords command will be handed over to matplotlib's `plot` command. This can be used to set plot properties such as the `color`.

Returns **axes** – The `Axes` instance that was used for plotting. This object allows to modify the plot further, e.g. by setting the plotted range, the axis labels, the plot title, etc.

Return type `Axes`

brian2tools.plotting.data.**plot_rate**(*times*, *rate*, *time_unit=<Mock name='mock.ms' id='140104943437712'>*, *rate_unit=<Mock name='mock.Hz' id='140105029554832'>*, *axes=None*, ***kwds*)

Parameters

- **times** (Quantity) – The time points at which the rate is measured.
- **rate** (Quantity) – The population rate for each time point in times
- **time_unit** (Unit, optional) – The unit to use for the time axis. Defaults to ms, but longer simulations could use second, for example.
- **time_unit** – The unit to use for the rate axis. Defaults to Hz.
- **axes** (Axes, optional) – The Axes instance used for plotting. Defaults to None which means that a new Axes will be created for the plot.
- **kwds** (*dict, optional*) – Any additional keywords command will be handed over to matplotlib's plot command. This can be used to set plot properties such as the color.

Returns **axes** – The Axes instance that was used for plotting. This object allows to modify the plot further, e.g. by setting the plotted range, the axis labels, the plot title, etc.

Return type Axes

brian2tools.plotting.morphology module

Module to plot Brian Morphology objects.

brian2tools.plotting.morphology.**plot_morphology**(*morphology*, *plot_3d=None*, *show_compartments=False*, *show_diameter=False*, *colors=('darkblue', 'darkred')*, *values=None*, *value_norm=(None, None)*, *value_colormap='hot'*, *value_colorbar=True*, *value_unit=None*, *axes=None*)

Plot a given Morphology in 2D or 3D.

Parameters

- **morphology** (Morphology) – The morphology to plot
- **plot_3d** (*bool, optional*) – Whether to plot the morphology in 3D or in 2D. If not set (the default) a morphology where all z values are 0 is plotted in 2D, otherwise it is plot in 3D.
- **show_compartments** (*bool, optional*) – Whether to plot a dot at the center of each compartment. Defaults to False.
- **show_diameter** (*bool, optional*) – Whether to plot the compartments with the diameter given in the morphology. Defaults to False.
- **colors** (*sequence of color specifications*) – A list of colors that is cycled through for each new section. Can be any color specification that matplotlib understands (e.g. a string such as 'darkblue' or a tuple such as (0, 0.7, 0).
- **values** (*Quantity, optional*) – Values to fill compartment patches with a color that corresponds to their given value.

- **value_norm** (*tuple or callable, optional*) – Normalization function to scale the displayed values. Can be a tuple of a minimum and a maximum value (where either of them can be `None` to denote taking the minimum/maximum from the data) or a function that takes a value and returns the scaled value (e.g. as returned by `matplotlib.colors.PowerNorm`). For a tuple of values, will use `matplotlib.colors.Normalize``(vmin, vmax, clip=True)`` with the given `(vmin, vmax)` values.
- **value_colormap** (*str or matplotlib.colors.Colormap, optional*) – Desired colormap for plots. Either the name of a standard colormap or a `matplotlib.colors.Colormap` instance. Defaults to `'hot'`. Note that this uses `matplotlib` color maps even for 3D plots with Mayavi.
- **value_colorbar** (*bool or dict, optional*) – Whether to add a colorbar for the `values`. Defaults to `True`, but will be ignored if no `values` are provided. Can also be a dictionary with the keyword arguments for matplotlib's `colorbar` method (2D plot), or for Mayavi's `scalarbar` method (3D plot).
- **value_unit** (`Unit`, optional) – A `Unit` to rescale the values for display in the colorbar. Does not have any visible effect if no colorbar is used. If not specified, will try to determine the "best unit" to itself.
- **axes** (`Axes` or `Scene`, optional) – A matplotlib `Axes` (for 2D plots) or mayavi `Scene` (for 3D plots) instance, where the plot will be added.

Returns **axes** – The `Axes` or `Scene` instance that was used for plotting. This object allows to modify the plot further, e.g. by setting the plotted range, the axis labels, the plot title, etc.

Return type `Axes` or `Scene`

brian2tools.plotting.morphology.**plot_dendrogram**(*morphology*, *axes=None*)

Plot a "dendrogram" of a morphology, i.e. an abstract representation which visualizes the branching structure and the length of each section.

Parameters

- **morphology** (`Morphology`) – The morphology to visualize.
- **axes** (`Axes`, optional) – The `Axes` instance used for plotting. Defaults to `None` which means that a new `Axes` will be created for the plot.

Returns **axes** – The `Axes` instance that was used for plotting. This object allows to modify the plot further, e.g. by setting the plotted range, the axis labels, the plot title, etc.

Return type `Axes`

brian2tools.plotting.synapses module

Module to plot synaptic connections.

brian2tools.plotting.synapses.**plot_synapses**(*sources*, *targets*, *values=None*, *var_unit=None*, *var_name=None*, *plot_type='scatter'*, *axes=None*, ***kwds*)

Parameters

- **sources** (`ndarray` of int) – The source indices of the connections (as returned by `Synapses.i`).
- **targets** (`ndarray` of int) – The target indices of the connections (as returned by `Synapses.j`).

- **values** (Quantity, ndarray) – The values to plot, a 1D array of the same size as sources and targets.
- **var_unit** (Unit, optional) – The unit to use to plot the values (e.g. mV for a membrane potential). If none is given (the default), an attempt is made to find a good scale automatically based on the values.
- **var_name** (*str, optional*) – The name of the variable that is plotted. Used for the axis label.
- **plot_type** ({'scatter', 'image', 'hexbin'}, optional) – What type of plot to use. Can be 'scatter' (the default) to draw a scatter plot, 'image' to display the connections as a matrix or 'hexbin' to display a 2D histogram using matplotlib's hexbin function. For a large number of synapses, 'scatter' will be very slow. Similarly, an 'image' plot will use a lot of memory for connections between two large groups. For a small number of neurons and synapses, 'hexbin' will be hard to interpret.
- **axes** (Axes, optional) – The Axes instance used for plotting. Defaults to None which means that a new Axes will be created for the plot.
- **kwds** (*dict, optional*) – Any additional keywords command will be handed over to the respective matplotlib command (scatter if the plot_type is 'scatter', imshow for 'image', and hexbin for 'hexbin'). This can be used to set plot properties such as the marker.

Returns **axes** – The Axes instance that was used for plotting. This object allows to modify the plot further, e.g. by setting the plotted range, the axis labels, the plot title, etc.

Return type Axes

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

m

n

p

Index

E

F

G