brian2tools Documentation

Release

Brian authors

Jun 13, 2017

Contents

1 Contents **3**
- 1.1 Release notes . . . 3
- 1.2 User's guide . . . 4
- 1.3 Developer's guide . . . 18

2 API reference **23**
- 2.1 brian2tools package . . . 23

3 Indices and tables **35**

Python Module Index **37**

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

Release notes

brian2tools 0.2.1.1

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

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.

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

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).

User's guide

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.

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*
  - *Spikes*
  - *Rates*
  - *State variables*
- *Plotting synaptic connections and variables*
  - *Connections*
  - *Synaptic variables (weights, delays, etc.)*
  - *Multiple synapses per source-target pair*
- *Plotting neuronal morphologies*
  - *Dendograms*
  - *Morphologies in 2D or 3D*

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)
```

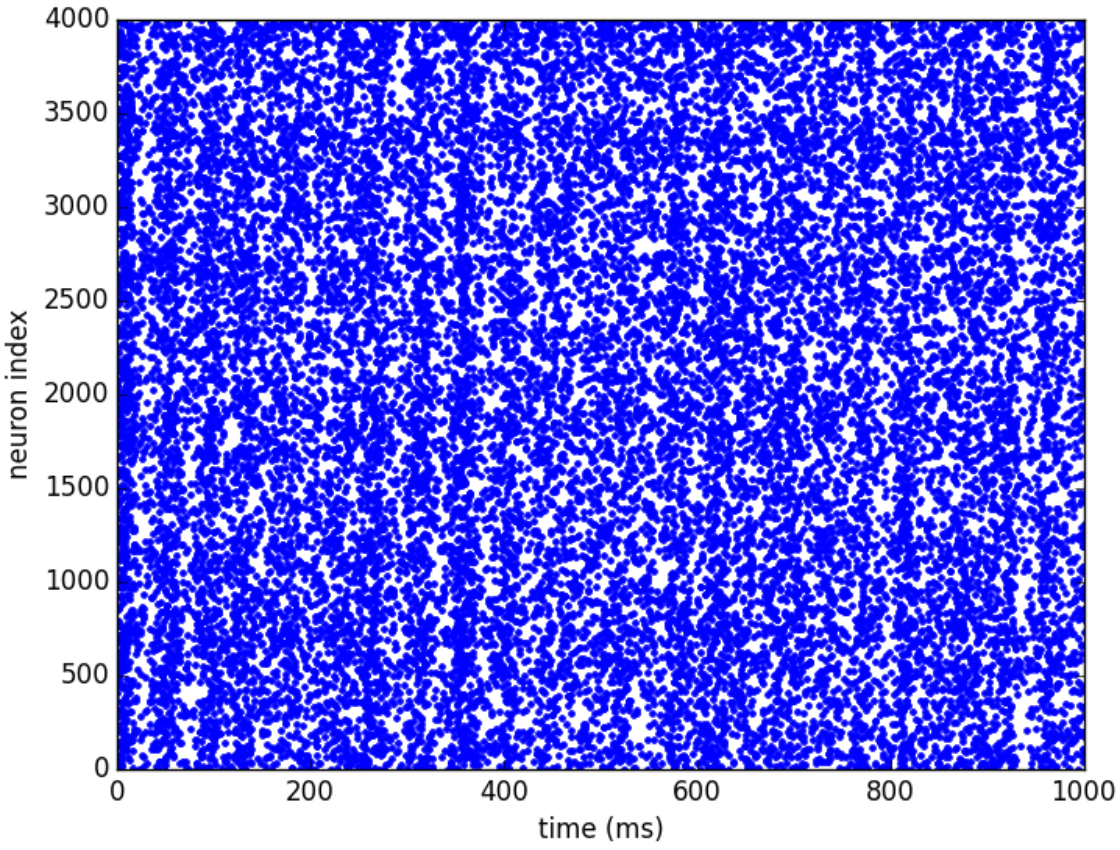
We will also assume that `brian2tools` has been imported like this:

```
from brian2tools import *
```

Spikes

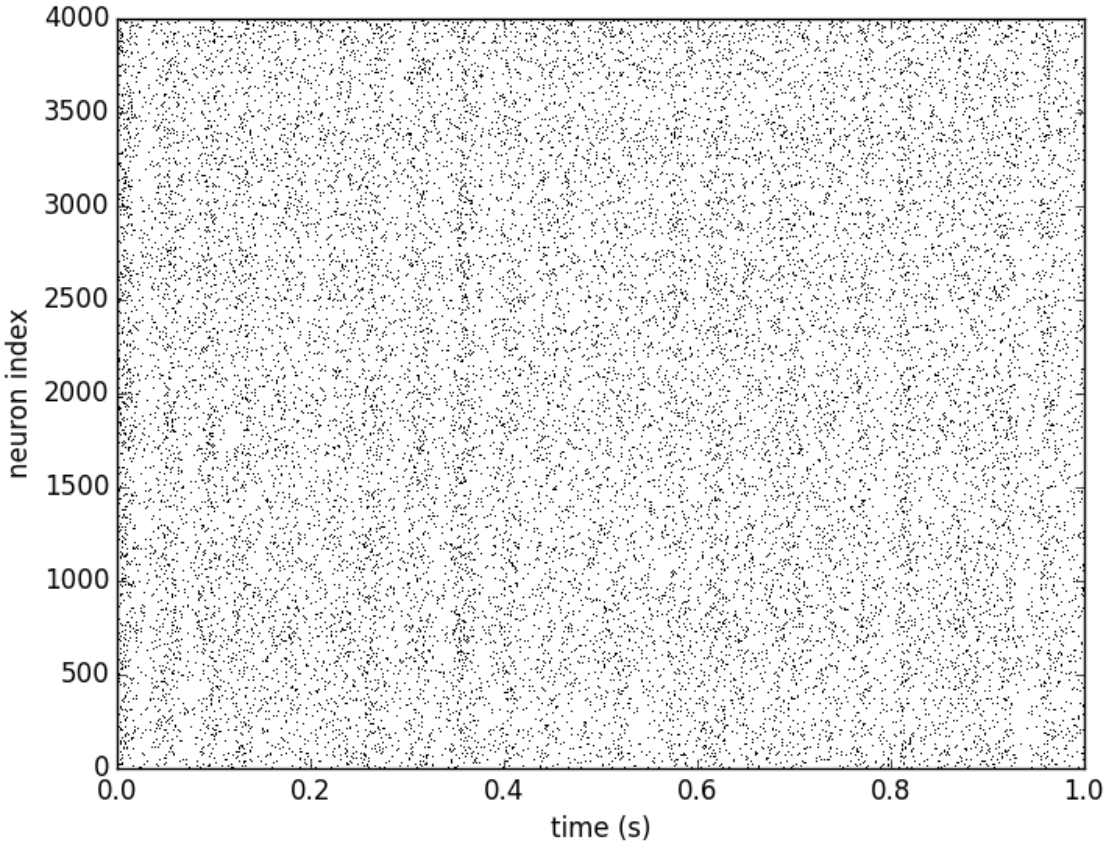
To plot a basic raster plot, you can call *brian_plot* with the `SpikeMonitor` as its argument:

```
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

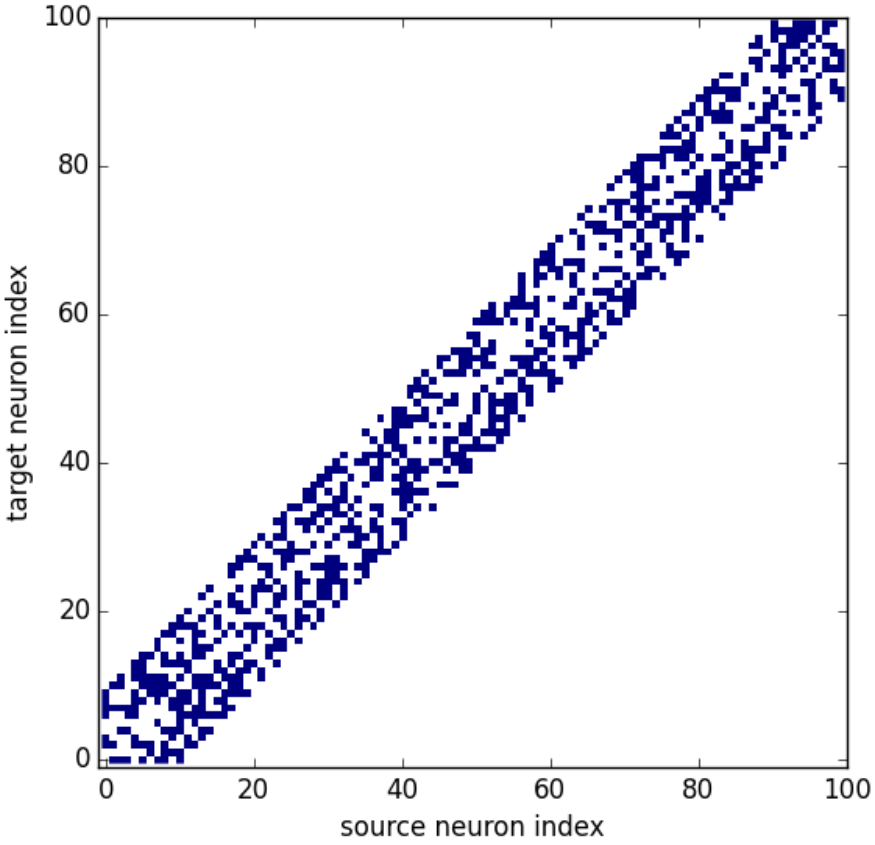
For the following examples, we create synapses and synaptic weights according to "distances" (differences between the source and target indices):

```
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

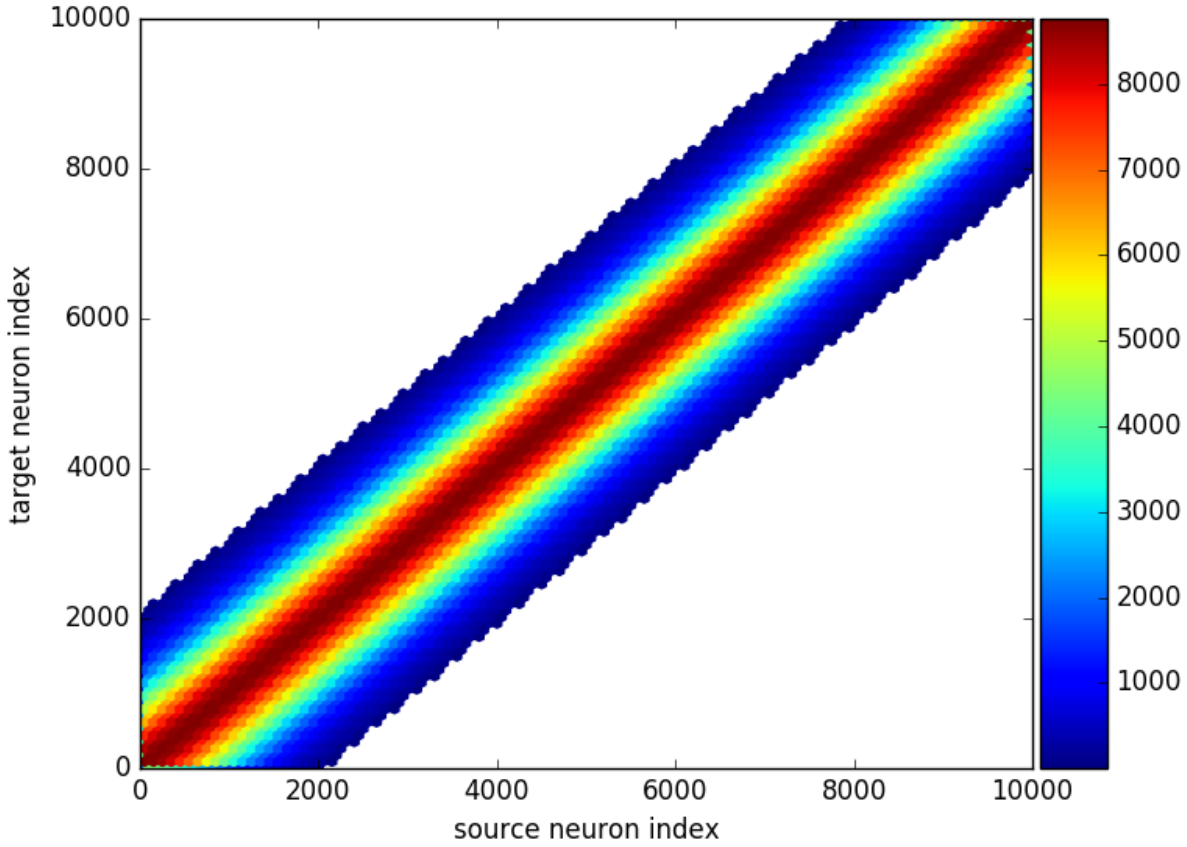
A call of *brian_plot* with a `Synapses` object will plot all connections, plotting either the matrix as an image, the connections as a scatter plot, or a 2-dimensional histogram (using matplotlib's `hexbin` function). The decision which type of plot to use is based on some heuristics applied to the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

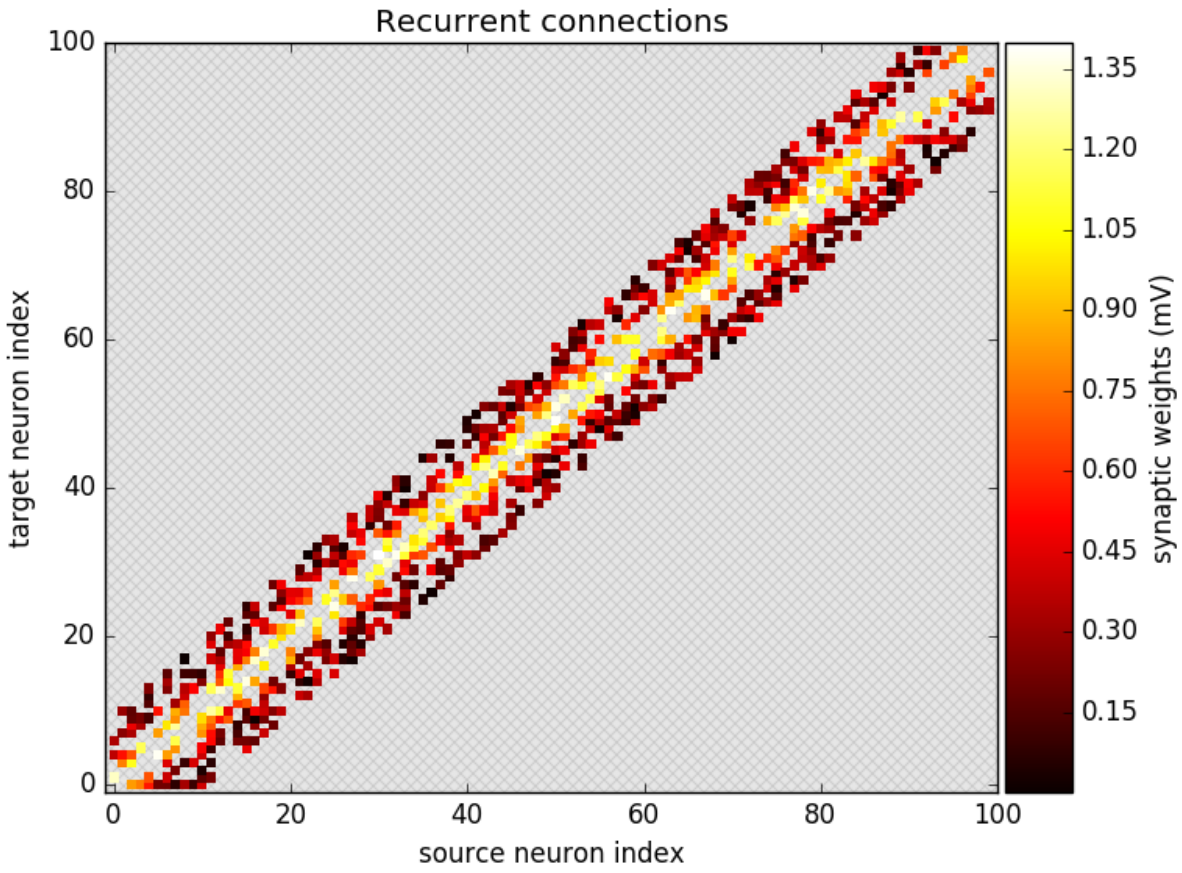
Synaptic variables such as synaptic weights or delays can also be plotted with *brian_plot*:

```
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')
```

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



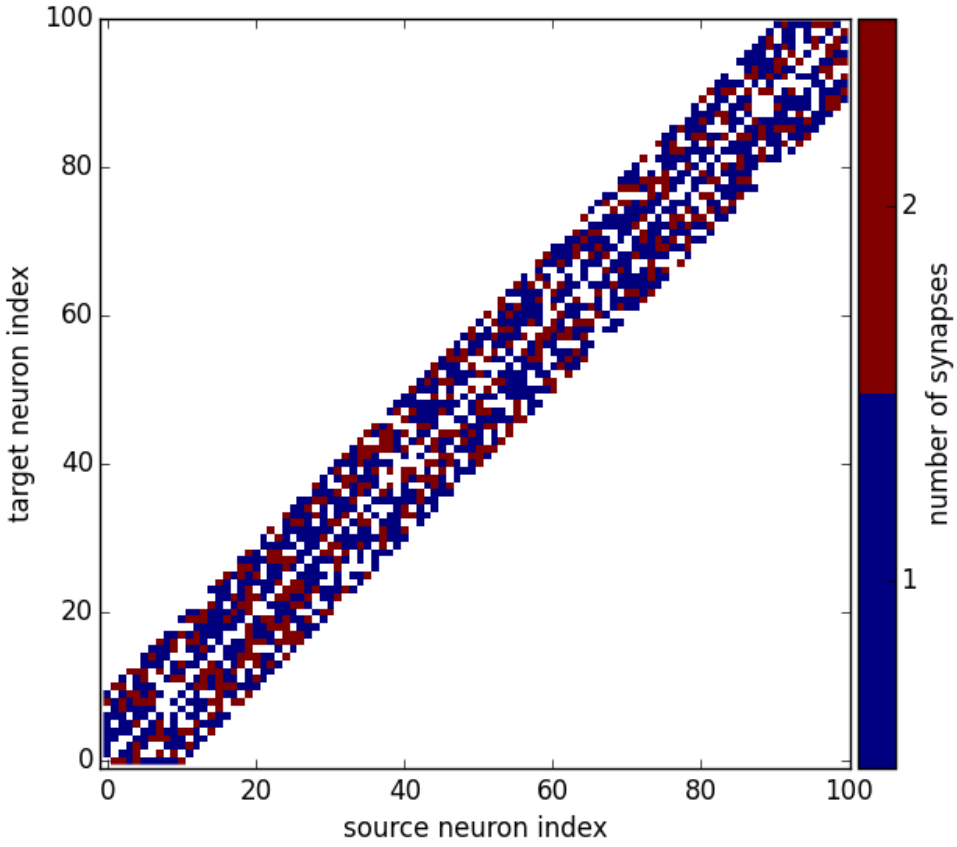
Multiple synapses per source-target pair

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)
```

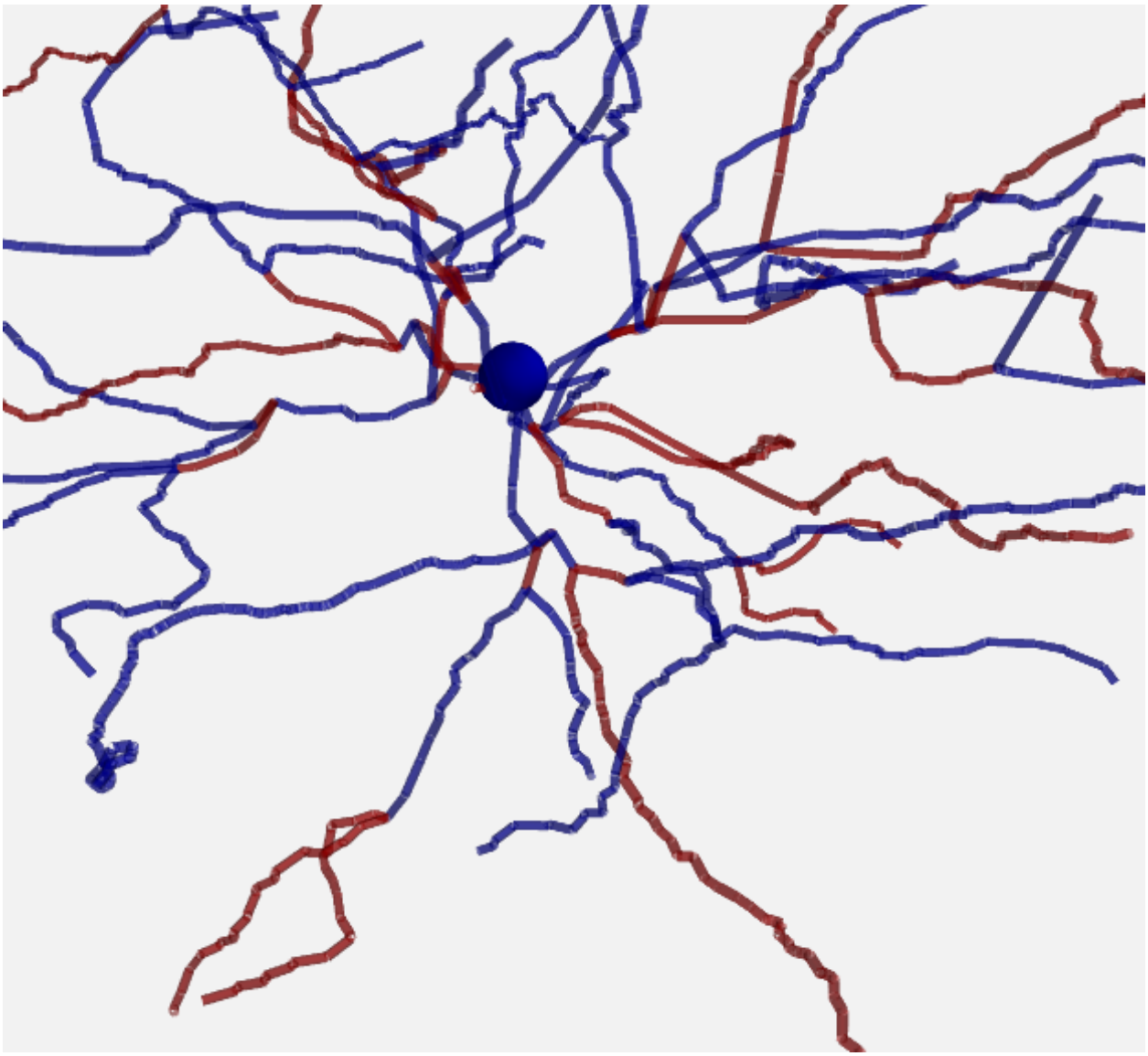
The *plot_dendrogram* function does the same thing, but in contrast to the other plot functions it does not allow any customization at the moment, so there is no benefit over using *brian_plot*.

[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)
```

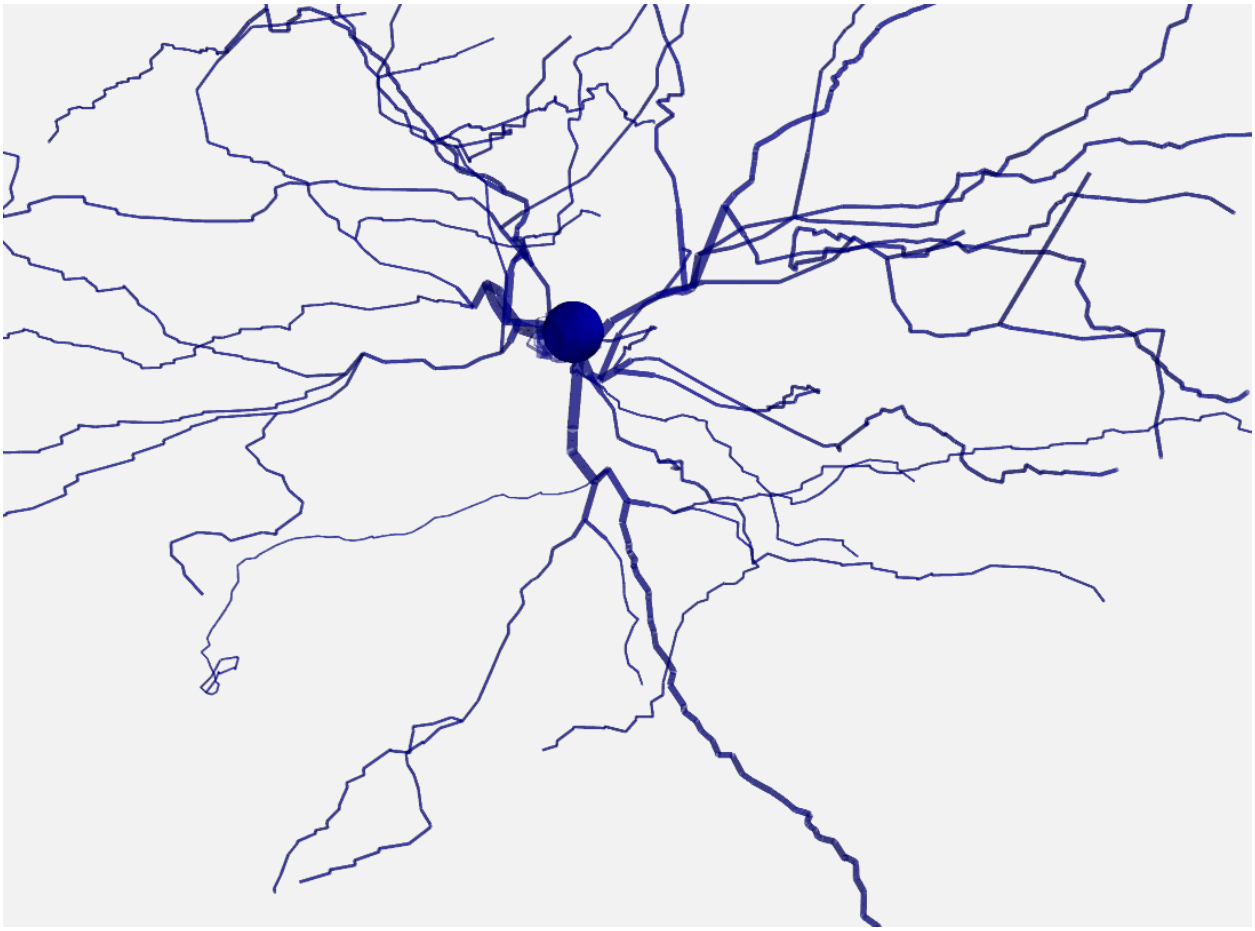


For artificially created morphologies (where one might only use coordinates in 2D) or to get a quick view of a morphology, you can also plot it in 2D (this will be done automatically if the coordinates are 2D only):

```
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',))
```



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

- *Working example*
- *Supported Features*
- *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">
        <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.).

Developer's guide

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.

NeuroML exporter

Overview

- *NMLExporter*
  - *Neuron Group*
  - *DOM structure*
  - *Model namespace*
- *LEMSDevice*
  - *LEMS Unit Constants*
- *Other modules*

- *TODO*

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.

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 test builds on travis will then automatically push the conda packages to anaconda.org.

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

brian2tools package

Tools for use with the Brian 2 simulator.

Subpackages

brian2tools.nmlexport package

Submodules

brian2tools.nmlexport.cgmhelper module

brian2tools.nmlexport.cgmhelper.**description**(*brian_obj*, *run_namespace*)

brian2tools.nmlexport.cgmhelper.**eq_string**(*equations*)

brian2tools.nmlexport.cgmhelper.**get_namespace_dict**(*identifiers*, *neurongroup*, *run_namespace*)

brian2tools.nmlexport.cgmhelper.**neurongroup_description**(*neurongroup*, *run_namespace*)

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.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='139979844788816'>*, *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='139979844788816'>*, *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='139979844788816'>*, *rate_unit=<Mock name='mock.Hz' id='139979844789968'>*, *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')*, *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)`.
- **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='139979844788816'>*, *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='139979844788816'>*, *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='139979844788816'>*, *rate_unit=<Mock name='mock.Hz' id='139979844789968'>*, *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')*, *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)`.
- **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

brian2tools, 23

n

brian2tools.nmlexport, 23
brian2tools.nmlexport.cgmhelper, 23
brian2tools.nmlexport.lemsexport, 23
brian2tools.nmlexport.lemsrendering, 23
brian2tools.nmlexport.supporting, 24

p

brian2tools.plotting, 26
brian2tools.plotting.base, 30
brian2tools.plotting.data, 31
brian2tools.plotting.morphology, 32
brian2tools.plotting.synapses, 33

Index

A

add_background_pattern() (in module brian2tools.plotting), 27
add_background_pattern() (in module brian2tools.plotting.base), 30
add_component() (brian2tools.nmlexport.supporting.NeuroMLSimpleNetwork method), 24
add_display() (brian2tools.nmlexport.supporting.NeuroMLSimulation method), 24
add_eventoutputfile() (brian2tools.nmlexport.supporting.NeuroMLSimulation method), 24
add_eventselection() (brian2tools.nmlexport.supporting.NeuroMLSimulation method), 24
add_line() (brian2tools.nmlexport.supporting.NeuroMLSimulation method), 25
add_outputcolumn() (brian2tools.nmlexport.supporting.NeuroMLSimulation method), 25
add_outputfile() (brian2tools.nmlexport.supporting.NeuroMLSimulation method), 25

B

brian2tools (module), 23
brian2tools.nmlexport (module), 23
brian2tools.nmlexport.cgmhelper (module), 23
brian2tools.nmlexport.lemsexport (module), 23
brian2tools.nmlexport.lemsrendering (module), 23
brian2tools.nmlexport.supporting (module), 24
brian2tools.plotting (module), 26
brian2tools.plotting.base (module), 30
brian2tools.plotting.data (module), 31
brian2tools.plotting.morphology (module), 32
brian2tools.plotting.synapses (module), 33
brian_plot() (in module brian2tools.plotting), 26
brian_plot() (in module brian2tools.plotting.base), 30
brian_unit_to_lems() (in module brian2tools.nmlexport.supporting), 26
build() (brian2tools.nmlexport.supporting.NeuroMLPoissonGenerator method), 24
build() (brian2tools.nmlexport.supporting.NeuroMLSimpleNetwork method), 24
build() (brian2tools.nmlexport.supporting.NeuroMLSimulation method), 25
build() (brian2tools.nmlexport.supporting.NeuroMLTarget method), 26

C

create_simulation() (brian2tools.nmlexport.supporting.NeuroMLSimulation method), 25

D

description() (in module brian2tools.nmlexport.cgmhelper), 23

E

eqn_string() (in module brian2tools.nmlexport.cgmhelper), 23

F

from_string() (in module brian2tools.nmlexport.supporting), 26

G

get_namespace_dict() (in module brian2tools.nmlexport.cgmhelper), 23

N

NeuroMLPoissonGenerator (class in brian2tools.nmlexport.supporting), 24
NeuroMLSimpleNetwork (class in brian2tools.nmlexport.supporting), 24
NeuroMLSimulation (class in brian2tools.nmlexport.supporting), 24
NeuroMLTarget (class in brian2tools.nmlexport.supporting), 26
neurongroup_description() (in module brian2tools.nmlexport.cgmhelper), 23

P

plot_dendrogram() (in module brian2tools.plotting), 29

plot_dendrogram() (in module brian2tools.plotting.morphology), 33
plot_morphology() (in module brian2tools.plotting), 28
plot_morphology() (in module brian2tools.plotting.morphology), 32
plot_raster() (in module brian2tools.plotting), 27
plot_raster() (in module brian2tools.plotting.data), 31
plot_rate() (in module brian2tools.plotting), 28
plot_rate() (in module brian2tools.plotting.data), 32
plot_state() (in module brian2tools.plotting), 28
plot_state() (in module brian2tools.plotting.data), 31
plot_synapses() (in module brian2tools.plotting), 29
plot_synapses() (in module brian2tools.plotting.synapses), 33

R

read_nml_dims() (in module brian2tools.nmlexport.supporting), 26
read_nml_units() (in module brian2tools.nmlexport.supporting), 26

U

update_simulation_attribute() (brian2tools.nmlexport.supporting.NeuroMLSimulation method), 25