Brian 2 Documentation

Release 2.0b4

Brian authors

July 17, 2015

Contents

Contents:

CHAPTER 1

Introduction

1.1 Installation

We recommend users to use the Anaconda distribution by Continuum Analytics. Its use will make the installation of Brian 2 and its dependencies simpler, since packages are provided in binary form, meaning that they don't have to be build from the source code at your machine. Furthermore, our automatic testing on the continuous integration services travis and appveyor are based on Anaconda, we are therefore confident that it works under this configuration.

However, Brian 2 can also be installed independent of Anaconda, either with other Python distributions (Enthought Canopy, Python(x,y) for Windows, ...) or simply based on Python and `pip` (see *Installation from source* below).

1.1.1 Installation with Anaconda

Installing Anaconda

Download the Anaconda distribution for your Operating System. For Windows users that want to use Python 3.x, we strongly recommend installing the 32 Bit version even on 64 Bit systems, since setting the compilation environment (see *Requirements for C++ code generation* below) is less complicated in that case. Note that the choice between Python 2.7 and Python 3.4 is not very important at this stage, Anaconda allows you to create a Python 3 environment from Python 2 Anaconda and vice versa.

After the installation, make sure that your environment is configured to use the Anaconda distribution. You should have access to the `conda` command in a terminal and running `python` (e.g. from your IDE) should show a header like this, indicating that you are using Anaconda's Python interpreter:

```
Python 2.7.10 |Anaconda 2.3.0 (64-bit)| (default, May 28 2015, 17:02:03)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
```

Here's some documentation on how to set up some popular IDEs for Anaconda: http://docs.continuum.io/anaconda/ide_integration.html

Installing Brian 2

You can either install Brian 2 in the Anaconda root environment, or create a new environment for Brian 2 (http://conda.pydata.org/docs/using/envs.html). The latter has the advantage that you can update (or not update) the dependencies of Brian 2 independently from the rest of your system.

Since Brian 2 is not part of the main Anaconda distribution, you have to install it from the brian-team channel. To add the `brian-team` channel, do:

```
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 brian2
```

If you prefer to not add the `brian-team` channel to your list of channels, you can also specify it explicitly when installing/updating/etc. To do an installation this way, use:

```
conda install -c brian-team brian2
```

Installing other useful packages

There are various packages that are useful but not necessary for working with Brian. These include: matplotlib (for plotting), nose (for running the test suite), ipython and ipython-notebook (for an interactive console). To install them, simply do:

```
conda install matplotlib nose ipython ipython-notebook
```

1.1.2 Installation from source

If you decide not to use Anaconda, you can install Brian 2 from the Python package index: https://pypi.python.org/pypi/brian2

To do so, use the `pip` utility. Newer versions of `pip` require you to use the `--pre` option to install Brian 2 since it is not yet a final release:

```
pip install --pre brian2
```

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.

In principle, the above command also install Brian's dependencies. Unfortunately, this does not work for `numpy`, it has to be installed in a separate step before all other dependencies (`pip install numpy`), if it is not already installed.

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

Alternatively, you can download the source package directly and uncompress it. You can then either run `python setup.py install` or `python setup.py develop` to install it, or simply add the source directory to your `PYTHONPATH` (this will only work for Python 2.x).

1.1.3 Requirements for C++ code generation

C++ code generation is highly recommended since it can drastically increase the speed of simulations (see Computational methods and efficiency for details). To use it, you need a C++ compiler and either Cython or weave (only for Python 2.x). Cython/weave will be automatically installed if you perform the installation via Anaconda, as recommended. Otherwise you can install them in the usual way, e.g. using `pip install cython` or `pip install weave`.

On Linux and Mac OS X, you will most likely already have a working C++ compiler installed (try calling `g++ --version` in a terminal). If not, use your distribution's package manager to install a `g++` package, or install `gcc` via Anaconda (`conda install gcc`).

On Windows, the necessary steps depend on the Python version you are using:

Python 2.7

- Download and install the Microsoft Visual C++ Compiler for Python 2.7

This should be all you need.

Python 3.4

- Download and install the Microsoft .NET Framework 4
- Download and install the Microsoft Windows SDK for Windows 7 and .NET Framework 4

For 64 Bit Windows (and Python 3.4), you have to additionally set up your environment correctly every time you run your Brian script (this is why we recommend against using this combination on Windows). To do this, run the following commands (assuming the default installation path) at the CMD prompt, or put them in a batch file:

```
setlocal EnableDelayedExpansion
CALL "C:\Program Files\Microsoft SDKs\Windows\v7.1\Bin\SetEnv.cmd" /x64 /release
set DISTUTILS_USE_SDK=1
```

Try running the test suite (see *Testing Brian* below) after the installation to make sure everything is working as expected.

1.1.4 Development version

To run the latest development code, you can install from brian-team's "dev" channel with Anaconda:

```
conda install -c brian-team/channels/dev brian2
```

You can also directly clone the git repository at github (https://github.com/brian-team/brian2) and then run `python setup.py install` or `python setup.py develop` or simply add the source directory to your `PYTHONPATH` (this will only work for Python 2.x).

Finally, another option is to use `pip` to directly install from github:

```
pip install https://github.com/brian-team/brian2/archive/master.zip
```

1.1.5 Testing Brian

If you have the nose testing utility installed, you can run Brian's test suite:

```
import brian2
brian2.test()
```

It should end with "OK", possibly showing a number of skipped tests but no warnings or errors. For more control about the tests that are run see the developer documentation on testing.

1.2 Release notes

1.2.1 Brian 2.0b4

This is the fourth (and probably last) beta release for Brian 2.0. This release adds a few important new features and fixes a number of bugs so we recommend all users of Brian 2 to upgrade. If you are a user new to Brian, we also recommend to directly start with Brian 2 instead of using the stable release of Brian 1. Note that the new recommended way to install Brian 2 is to use the Anaconda distribution and to install the Brian 2 conda package (see Installation).

This is however still a Beta release, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

Major new features

- In addition to the standard threshold/reset, groups can now define "custom events". These can be recorded with the new *EventMonitor* (a generalization of *SpikeMonitor*) and *Synapses* can connect to these events instead of the standard spike event. See Custom events for more details.
- *SpikeMonitor* and *EventMonitor* can now also record state variable values at the time of spikes (or custom events), thereby offering the functionality of `StateSpikeMonitor` from Brian 1. See *Recording variables at spike time* for more details.
- The code generation modes that interact with C++ code (weave, Cython, and C++ standalone) can now be more easily configured to work with external libraries (compiler and linker options, header files, etc.). See the documentation of the *cpp_prefs* module for more details.

Improvemements and bug fixes

- Cython simulations no longer interfere with each other when run in parallel (thanks to Daniel Bliss for reporting and fixing this).
- The C++ standalone now works with scalar delays and the spike queue implementation deals more efficiently with them in general.
- Dynamic arrays are now resized more efficiently, leading to faster monitors in runtime mode.
- The spikes generated by a *SpikeGeneratorGroup* can now be changed between runs using the *set_spikes* method.
- Multi-step state updaters now work correctly for non-autonomous differential equations
- *PoissonInput* now correctly works with multiple clocks (thanks to Daniel Bliss for reporting and fixing this)
- The *get_states* method now works for *StateMonitor*. This method provides a convenient way to access all the data stored in the monitor, e.g. in order to store it on disk.
- C++ compilation is now easier to get to work under Windows, see Installation for details.

Important backwards-incompatible changes

- The `custom_operation` method has been renamed to *run_regularly* and can now be called without the need for storing its return value.
- *StateMonitor* will now by default record at the beginning of a time step instead of at the end. See *Recording variables continuously* for details.

- Scalar quantities now behave as python scalars with respect to in-place modifications (augmented assignments). This means that `x = 3*mV; y = x; y += 1*mV` will no longer increase the value of the variable `x` as well.

Infrastructure improvements

- We now provide conda packages for Brian 2, making it very easy to install when using the Anaconda distribution (see Installation).

Contributions

Code and documentation contributions (ordered by the number of commits):

- Marcel Stimberg (@mstimberg)
- Dan Goodman (@thesamovar)
- Daniel Bliss (@dabliss)
- Romain Brette (@romainbrette)

Testing, suggestions and bug reports (ordered alphabetically, apologies to everyone we forgot...):

- Daniel Bliss
- Damien Drix
- Rainer Engelken
- Beatriz Herrera Figueredo
- Owen Mackwood
- Augustine Tan
- Ot de Wiljes

1.2.2 Brian 2.0b3

This is the third beta release for Brian 2.0. This release does not add many new features but it fixes a number of important bugs so we recommend all users of Brian 2 to upgrade. If you are a user new to Brian, we also recommend to directly start with Brian 2 instead of using the stable release of Brian 1.

This is however still a Beta release, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

Major new features

- A new *`PoissonInput`* class for efficient simulation of Poisson-distributed input events.

Improvements

- The order of execution for `pre` and `post` statements happending in the same time step was not well defined (it fell back to the default alphabetical ordering, executing `post` before `pre`). It now explicitly specifies the `order` attribute so that `pre` gets executed before `post` (as in Brian 1). See the Synapses documentation for details.

- The default schedule that is used can now be set via a preference (*core.network.default_schedule*). New automatically generated scheduling slots relative to the explicitly defined ones can be used, e.g. `before_resets` or `after_synapses`. See *Scheduling* for details.
- The scipy package is no longer a dependency (note that weave for compiled C code under Python 2 is now available in a separate package). Note that multicompartmental models will still benefit from the scipy package if they are simulated in pure Python (i.e. with the `numpy` code generation target) – otherwise Brian 2 will fall back to a numpy-only solution which is significantly slower.

Important bug fixes

- Fix *`SpikeGeneratorGroup`* which did not emit all the spikes under certain conditions for some code generation targets (#429)
- Fix an incorrect update of pre-synaptic variables in synaptic statements for the `numpy` code generation target (#435).
- Fix the possibility of an incorrect memory access when recording a subgroup with *`SpikeMonitor`* (#454).
- Fix the storing of results on disk for C++ standalone on Windows – variables that had the same name when ignoring case (e.g. `i` and `I`) where overwriting each other (#455).

Infrastructure improvements

- Brian 2 now has a chat room on gitter: https://gitter.im/brian-team/brian2
- The sphinx documentation can now be built from the release archive file
- After a big cleanup, all files in the repository have now simple LF line endings (see https://help.github.com/articles/dealing-with-line-endings/ on how to configure your own machine properly if you want to contribute to Brian).

Contributions

Code and documentation contributions (ordered by the number of commits):

- Marcel Stimberg (@mstimberg)
- Dan Goodman (@thesamovar)
- Konrad Wartke (@kwartke)

Testing, suggestions and bug reports (ordered alphabetically, apologies to everyone we forgot...):

- Daniel Bliss
- Owen Mackwood
- Ankur Sinha
- Richard Tomsett

1.2.3 Brian 2.0b2

This is the second beta release for Brian 2.0, we recommend all users of Brian 2 to upgrade. If you are a user new to Brian, we also recommend to directly start with Brian 2 instead of using the stable release of Brian 1.

This is however still a Beta release, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

Major new features

- Multi-compartmental simulations can now be run using the *C++ standalone* mode (this is not yet well-tested, though).
- The implementation of `TimedArray` now supports two-dimensional arrays, i.e. different input per neuron (or synapse, etc.), see *Timed arrays* for details.
- Previously, not setting a code generation target (using the *codegen.target* preference) would mean that the `numpy` target was used. Now, the default target is `auto`, which means that a compiled language (`weave` or `cython`) will be used if possible. See Computational methods and efficiency for details.
- The implementation of `SpikeGeneratorGroup` has been improved and it now supports a `period` argument to repeatedly generate a spike pattern.

Improvements

- The selection of a numerical algorithm (if none has been specified by the user) has been simplified. See *Numerical integration* for details.
- Expressions that are shared among neurons/synapses are now updated only once instead of for every neuron/synapse which can lead to performance improvements.
- On Windows, The Microsoft Visual C compiler is now supported in the `cpp_standalone` mode, see the respective notes in the Installation and Computational methods and efficiency documents.
- Simulation runs (using the standard "runtime" device) now collect profiling information. See *Profiling* for details.

Infrastructure and documentation improvements

- Tutorials for beginners in the form of ipython notebooks (currently only covering the basics of neurons and synapses) are now available.
- The Examples in the documentation now include the images they generated. Several examples have been adapted from Brian 1.
- The code is now automatically tested on Windows machines, using the appveyor service. This complements the Linux testing on travis.
- Using a version of a dependency (e.g. sympy) that we don't support will now raise an error when you import `brian2` – see *Dependency checks* for more details.
- Test coverage for the `cpp_standalone` mode has been significantly increased.

Important bug fixes

- The preparation time for complicated equations has been significantly reduced.
- The string representation of small physical quantities has been corrected (#361)
- Linking variables from a group of size 1 now works correctly (#383)

Contributions

Code and documentation contributions (ordered by the number of commits):

- Marcel Stimberg (@mstimberg)
- Dan Goodman (@thesamovar)
- Romain Brette (@romainbrette)
- Pierre Yger (@yger)

Testing, suggestions and bug reports (ordered alphabetically, apologies to everyone we forgot...):

- Conor Cox
- Gordon Erlebacher
- Konstantin Mergenthaler

1.2.4 Brian 2.0beta

This is the first beta release for Brian 2.0 and the first version of Brian 2.0 we recommend for general use. From now on, we will try to keep changes that break existing code to a minimum. If you are a user new to Brian, we'd recommend to start with the Brian 2 beta instead of using the stable release of Brian 1.

This is however still a Beta release, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

Major new features

- New classes *`Morphology`* and *`SpatialNeuron`* for the simulation of Multicompartment models
- A temporary "bridge" for `brian.hears` that allows to use its Brian 1 version from Brian 2 (Brian 1 Hears bridge)
- Cython is now a new code generation target, therefore the performance benefits of compiled code are now also available to users running simulations under Python 3.x (where `scipy.weave` is not available)
- Networks can now store their current state and return to it at a later time, e.g. for simulating multiple trials starting from a fixed network state (*Continuing/repeating simulations*)
- C++ standalone mode: multiple processors are now supported via OpenMP (*Multi-threading with OpenMP*), although this code has not yet been well tested so may be inaccurate.
- C++ standalone mode: after a run, state variables and monitored values can be loaded from disk transparently. Most scripts therefore only need two additional lines to use standalone mode instead of Brian's default runtime mode (*C++ standalone*).

Syntax changes

- The syntax and semantics of everything around simulation time steps, clocks, and multiple runs have been cleaned up, making `reinit` obsolete and also making it unnecessary for most users to explicitly generate *`Clock`* objects – instead, a `dt` keyword can be specified for objects such as *`NeuronGroup`* (Running a simulation)
- The `scalar` flag for parameters/subexpressions has been renamed to `shared`
- The "unit" for boolean variables has been renamed from `bool` to `boolean`

- C++ standalone: several keywords of *CPPStandaloneDevice.build* have been renamed
- The preferences are now accessible via `prefs` instead of `brian_prefs`
- The `runner` method has been renamed to *custom_operation*

Improvements

- Variables can now be linked across *NeuronGroup*s (*Linked variables*)
- More flexible progress reporting system, progress reporting also works in the C++ standalone mode (*Progress reporting*)
- State variables can be declared as `integer` (*Equation strings*)

Bug fixes

57 github issues have been closed since the alpha release, of which 26 had been labeled as bugs. We recommend all users of Brian 2 to upgrade.

Contributions

Code and documentation contributions (ordered by the number of commits):

- Marcel Stimberg (@mstimberg)
- Dan Goodman (@thesamovar)
- Romain Brette (@romainbrette)
- Pierre Yger (@yger)
- Werner Beroux (@wernight)

Testing, suggestions and bug reports (ordered alphabetically, apologies to everyone we forgot...):

- Guillaume Bellec
- Victor Benichoux
- Laureline Logiaco
- Konstantin Mergenthaler
- Maurizio De Pitta
- Jan-Hendrick Schleimer
- Douglas Sterling
- Katharina Wilmes

1.3 Changes from Brian 1

In most cases, Brian 2 works in a very similar way to Brian 1 but there are some important differences to be aware of. The major distinction is that in Brian 2 you need to be more explicit about the definition of your simulation in order to avoid inadvertent errors. For example, the equations defining thresholds, resets and refractoriness have to be fully explicitly specified strings. In addition, some cases where you could use the 'magic network' system in Brian 1 won't work in Brian 2 and you'll get an error telling you that you need to create an explicit *Network* object.

The old system of `Connection` and related synaptic objects such as `STDP` and `STP` have been removed and replaced with the new *`Synapses`* class.

A slightly technical change that might have a significant impact on your code is that the way 'namespaces' are handled has changed. You can now change the value of parameters specified outside of equations between simulation runs, as well as changing the `dt` value of the simulation between runs.

The units system has also been modified so that now arrays have a unit instead of just single values. Finally, a number of objects and classes have been removed or simplified.

For more details, see below.

1.3.1 Major interface changes

More explicit model specifications

A design principle of Brian 2 is that model specifications are unambiguous and explicit. Some "guessing" has therefore been removed, for example Brian 2 no longer tries to determine which variable is the membrane potential and should be used for thresholding and reset. This entails:

- Threshold and reset have to use explicit string descriptions, e.g. `threshold='v>-50*mv'` and `reset='v = -70*mV'` instead of `threshold=-50*mV` and `reset=-70*mV`
- When a variable should be clamped during refractoriness (in Brian 1, the membrane potential was clamped by default), it has to be explicitly marked with the flag `(unless refractory)` in the equations

Clocks and networks

Brian's system of handling clocks and networks has been substantially changed. You now usually specify a value of `dt` either globally or explicitly for each object rather than creating clocks (although this is still possible).

More importantly, the behaviour of networks is different:

- Either you create a *`Network`* of objects you want to simulate explicitly, or you use the 'magic' system which now simulates all named objects in the context where you run it.
- The magic network will now raise errors if you try to do something where it cannot accurately guess what you mean. In these situations, we recommend using an explicit *`Network`*.
- Objects can now only belong to a single *`Network`* object, in order to avoid inadvertent errors.
- Similarly, you can no longer change the time explicitly: the only way the time changes is by running a simulation. Instead, you can *`store()`* and *`restore()`* the state of a *`Network`* (including the time).

Removed classes

Several classes have been merged or are replaced by string-based model specifications:

- *Connections*, *STP* and *STDP* are replaced by *`Synapses`*
- All reset and refractoriness classes (*VariableReset*, *CustomRefractoriness*, etc.) are replaced by the new string-based reset and refractoriness mechanisms, see Models and neuron groups and Refractoriness
- *`Clock`* is the only class for representing clocks, *FloatClock* and *EventClock* are obsolete
- The functionality of *MultiStateMonitor* is provided by the standard *`StateMonitor`* class.
- The functionality of *StateSpikeMonitor* is provided by the *`SpikeMonitor`* class.

- The library of models has been removed (*leaky_IF*, *Izhikevich*, *alpha_synapse*, *OrnsteinUhlenbeck*, etc.), specify the models directly in the equations instead

Units

The unit system now extends to arrays, e.g. `np.arange(5) * mV` will retain the units of volts and not discard them as Brian 1 did. Brian 2 is therefore also more strict in checking the units. For example, if the state variable `v` uses the unit of volt, the statement `G.v = np.rand(len(G)) / 1000.` will now raise an error. For consistency, units are returned everywhere, e.g. in monitors. If `mon` records a state variable v, `mon.t` will return a time in seconds and `mon.v` the stored values of `v` in units of volts.

If a pure numpy array without units is needed for further processing, there are several options: if it is a state variable or a recorded variable in a monitor, appending an underscore will refer to the variable values without units, e.g. `mon.t_` returns pure floating point values. Alternatively, the units can be removed by diving through the unit (e.g. `mon.t / second`) or by explicitly converting it (`np.asarray(mon.t)`).

State monitor

The *StateMonitor* has a slightly changed interface and also includes the functionality of the former *MultiStateMonitor*. The stored values are accessed as attributes, e.g.:

```
mon = StateMonitor(G, ['v', 'w'], record=True)
print mon[0].v  # v value for the first neuron, with units
print mon.w_  # v values for all neurons, without units
print mon. t / ms  # stored times
```

If accessed without index (e.g. `mon.v`), the stored values are returned as a two-dimensional array with the size NxM, where N is the number of recorded neurons and M the number of time points. Therefore, plotting all values can be achieved by:

```
plt.plot(mon.t / ms, mon.v.T)
```

The monitor can also be indexed to give the values for a specific neuron, e.g. `mon[0].v`. Note that in case that not all neurons are recorded, writing `mon[i].v` and `mon.v[i]` makes a difference: the former returns the value for neuron i while the latter returns the value for the *ith* recorded neuron.:

```
mon = StateMonitor(G, 'v', record=[0, 2, 4])
print mon[2].v  # v values for neuron number 2
print mon.v[2]  # v values for neuron number 4
```

Another change is that the *StateMonitor* now records in the `'start'` scheduling slot by default. This leads to a more intuitive correspondence between the recorded times and the values: previously (where *StateMonitor* recorded in the `'end'` slot) the recorded value at 0ms was not the initial value of the variable but the value after integrating it for a single time step. The disadvantage of this new default is that the very last value at the end of the last time step of a simulation is not recorded anymore. However, this value can be manually added to the monitor by calling *StateMonitor.record_single_timestep()*.

Miscellaneous changes

- New preferences system (see Preferences system)
- New handling of namespaces (see Equations)
- New "magic" and clock system (see Scheduling and custom progress reporting and Running a simulation)
- New refractoriness system (see Refractoriness)

- More powerful string expressions that can also be used as indices for state variables (see e.g. Synapses)
- "Brian Hears" is being rewritten, but there is a bridge to the version included in Brian 1 until the new version is written (see Brian 1 Hears bridge)
- *`Equations`* objects no longer save their namespace, they now behave just like strings.
- There is no longer any `reinit()` mechanism, this is now handled by *`store()`* and *`restore()`*.

1.3.2 Changes in the internal processing

In Brian 1, the internal state of some objects changed when a network was run for the first time and therefore some fundamental settings (e.g. the clock's dt, or some code generation settings) were only taken into account before that point. In Brian 2, objects do not change their internal state, instead they recreate all necessary data structures from scratch at every run. This allows to change external variables, a clock's dt, etc. between runs. Note that currently this is not optimized for performance, i.e. some work is unnecessarily done several times, the setup phase of a network and of each individual run may therefore appear slow compared to Brian 1 (see #124).

1.4 Known issues

List of known issues

1.4.1 Cannot find msvcr90d.dll

If you see this message coming up, find the file `PythonDir\Lib\site-packages\numpy\distutils\mingw32ccompiler` and modify the line `msvcr_dbg_success = build_msvcr_library(debug=True)` to read `msvcr_dbg_success = False` (you can comment out the existing line and add the new line immediately after).

1.4.2 Problems with numerical integration

If the beginning of a run takes a long time, the reason might be the automatic determination of a suitable numerical integration algorithm. This can in particular happen for complicated equations where sympy's solvers take a long time trying to solve the equations symbolically (typically failing in the end). We try to improve this situation (see #351) but until then, chose a numerical integration algorithm explicitly (*Numerical integration*).

CHAPTER 2

Tutorials

The tutorial consists of a series of IPython notebooks [1]. If you run such a notebook on your own computer, you can interactively change the code in the tutorial and experiment with it – this is the recommended way to get started with Brian. The first link for each tutorial below leads to a non-interactive version of the notebook; use the links under "Notebook files" to get a file that you can run on your computer. You can also copy such a link and paste it at http://nbviewer.ipython.org – this will get you a nicer (but still non-interactive) rendering than the one you see in our documentation.

For more information about how to use IPython notebooks, see the IPython notebook documentation.

2.1 Introduction to Brian part 1: Neurons

> **Note:** This tutorial is written as an interactive notebook that should be run on your own computer. See the tutorial overview page for more details.
>
> Download link for this tutorial: `1-intro-to-brian-neurons.ipynb`.

All Brian scripts start with the following. If you're trying this notebook out in IPython, you should start by running this cell.

```
from brian2 import *
```

Later we'll do some plotting in the notebook, so we activate inline plotting in the IPython notebook by doing this:

```
%matplotlib inline
```

2.1.1 Units system

Brian has a system for using quantities with physical dimensions:

```
print 20*volt
```

```
20. V
```

All of the basic SI units can be used (volt, amp, etc.) along with all the standard prefixes (m=milli, p=pico, etc.), as well as a few special abbreviations like `mV` for millivolt, `pF` for picofarad, etc.

[1] The project has been partly renamed to Jupyter recently

```
print 1000*amp
```

```
1. kA
```

```
print 1e6*volt
```

```
1. MV
```

```
print 1000*namp
```

```
1. uA
```

Also note that combinations of units with work as expected:

```
print 10*nA*5*Mohm
```

```
50. mV
```

And if you try to do something wrong like adding amps and volts, what happens?

```
print 5*amp+10*volt
```

```
---------------------------------------------------------------------------

DimensionMismatchError                    Traceback (most recent call last)

<ipython-input-8-a44fa670700d> in <module>()
----> 1 print 5*amp+10*volt


/home/marcel/programming/brian2/brian2/units/fundamentalunits.pyc in __add__(self, other)
   1301         return self._binary_operation(other, operator.add,
   1302                                       fail_for_mismatch=True,
-> 1303                                       message='Addition')
   1304
   1305     def __radd__(self, other):


/home/marcel/programming/brian2/brian2/units/fundamentalunits.pyc in _binary_operation(self, other, c
   1249
   1250         if fail_for_mismatch:
-> 1251             fail_for_dimension_mismatch(self, other, message)
   1252
   1253         if inplace:


/home/marcel/programming/brian2/brian2/units/fundamentalunits.pyc in fail_for_dimension_mismatch(obj1
    147         if error_message is None:
    148             error_message = 'Dimension mismatch'
--> 149         raise DimensionMismatchError(error_message, dim1, dim2)
    150
    151


DimensionMismatchError: Addition, dimensions were (A) (m^2 kg s^-3 A^-1)
```

If you haven't see an error message in Python before that can look a bit overwhelming, but it's actually quite simple and it's important to know how to read these because you'll probably see them quite often.

You should start at the bottom and work up. The last line gives the error type `DimensionMismatchError` along with a more specific message (in this case, you were trying to add together two quantities with different SI units, which is impossible).

Working upwards, each of the sections starts with a filename (e.g. `C:\Users\Dan\...`) with possibly the name of a function, and then a few lines surrounding the line where the error occurred (which is identified with an arrow).

The last of these sections shows the place in the function where the error actually happened. The section above it shows the function that called that function, and so on until the first section will be the script that you actually run. This sequence of sections is called a traceback, and is helpful in debugging.

If you see a traceback, what you want to do is start at the bottom and scan up the sections until you find your own file because that's most likely where the problem is. (Of course, your code might be correct and Brian may have a bug in which case, please let us know on the email support list.)

2.1.2 A simple model

Let's start by defining a simple neuron model. In Brian, all models are defined by systems of differential equations. Here's a simple example of what that looks like:

```
tau = 10*ms
eqs = '''
dv/dt = (1-v)/tau : 1
'''
```

In Python, the notation `'''` is used to begin and end a multi-line string. So the equations are just a string with one line per equation. The equations are formatted with standard mathematical notation, with one addition. At the end of a line you write `: unit` where `unit` is the SI unit of that variable.

Now let's use this definition to create a neuron.

```
G = NeuronGroup(1, eqs)
```

In Brian, you only create groups of neurons, using the class `NeuronGroup`. The first two arguments when you create one of these objects are the number of neurons (in this case, 1) and the defining differential equations.

Let's see what happens if we didn't put the variable `tau` in the equation:

```
eqs = '''
dv/dt = 1-v : 1
'''
G = NeuronGroup(1, eqs)
```

```
---------------------------------------------------------------------------

DimensionMismatchError                    Traceback (most recent call last)

<ipython-input-11-70d526e22e27> in <module>()
      2 dv/dt = 1-v : 1
      3 '''
----> 4 G = NeuronGroup(1, eqs)


/home/marcel/programming/brian2/brian2/groups/neurongroup.pyc in __init__(self, N, model, method, thr
    403         # can spot unit errors in the equation already here.
    404         try:
--> 405             self.before_run(None)
    406         except KeyError:
    407             pass
```

```
/home/marcel/programming/brian2/brian2/groups/neurongroup.pyc in before_run(self, run_namespace, leve
    643         # Check units
    644         self.equations.check_units(self, run_namespace=run_namespace,
--> 645                                    level=level+1)
    646
    647     def _repr_html_(self):


/home/marcel/programming/brian2/brian2/equations/equations.pyc in check_units(self, group, run_namesp
    861                                                  '\n%s') % (eq.varname,
    862                                                             ex.desc),
--> 863                                                 *ex.dims)
    864             elif eq.type == SUBEXPRESSION:
    865                 try:


DimensionMismatchError: Inconsistent units in differential equation defining variable v:
Expression 1-v  does not have the expected units, dimensions were (1) (s^-1)
```

An error is raised, but why? The reason is that the differential equation is now dimensionally inconsistent. The left hand side `dv/dt` has units of `1/second` but the right hand side `1-v` is dimensionless. People often find this behaviour of Brian confusing because this sort of equation is very common in mathematics. However, for quantities with physical dimensions it is incorrect because the results would change depending on the unit you measured it in. For time, if you measured it in seconds the same equation would behave differently to how it would if you measured time in milliseconds. To avoid this, we insist that you always specify dimensionally consistent equations.

Now let's go back to the good equations and actually run the simulation.

```
start_scope()

tau = 10*ms
eqs = '''
dv/dt = (1-v)/tau : 1
'''

G = NeuronGroup(1, eqs)
run(100*ms)
```

First off, ignore that `start_scope()` at the top of the cell. You'll see that in each cell in this tutorial where we run a simulation. All it does is make sure that any Brian objects created before the function is called aren't included in the next run of the simulation.

So, what has happened here? Well, the command `run(100*ms)` runs the simulation for 100 ms. We can see that this has worked by printing the value of the variable `v` before and after the simulation.

```
start_scope()

G = NeuronGroup(1, eqs)
print 'Before v =', G.v[0]
run(100*ms)
print 'After v =', G.v[0]
```

```
Before v = 0.0
After v = 0.99995460007
```

By default, all variables start with the value 0. Since the differential equation is `dv/dt=(1-v)/tau` we would expect after a while that `v` would tend towards the value 1, which is just what we see. Specifically, we'd expect `v` to have the value `1-exp(-t/tau)`. Let's see if that's right.

```
print 'Expected value of v =', 1-exp(-100*ms/tau)
```

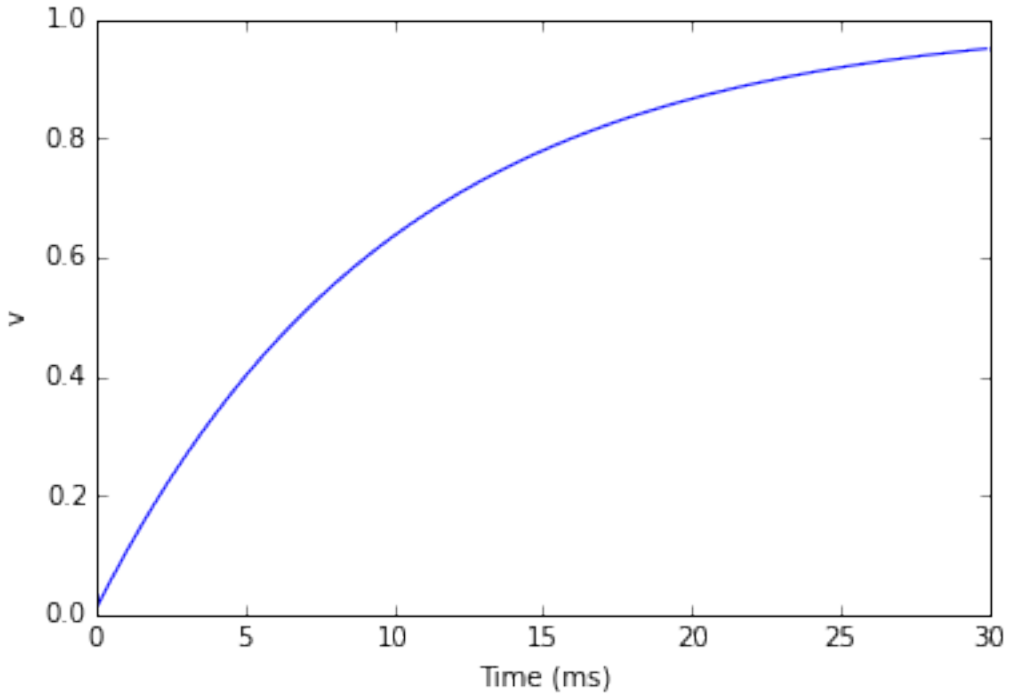
```
Expected value of v = 0.99995460007
```

Good news, the simulation gives the value we'd expect!

Now let's take a look at a graph of how the variable `v` evolves over time.

```
start_scope()

G = NeuronGroup(1, eqs)
M = StateMonitor(G, 'v', record=True)

run(30*ms)

plot(M.t/ms, M.v[0])
xlabel('Time (ms)')
ylabel('v')
```

```
<matplotlib.text.Text at 0x7ff097050790>
```

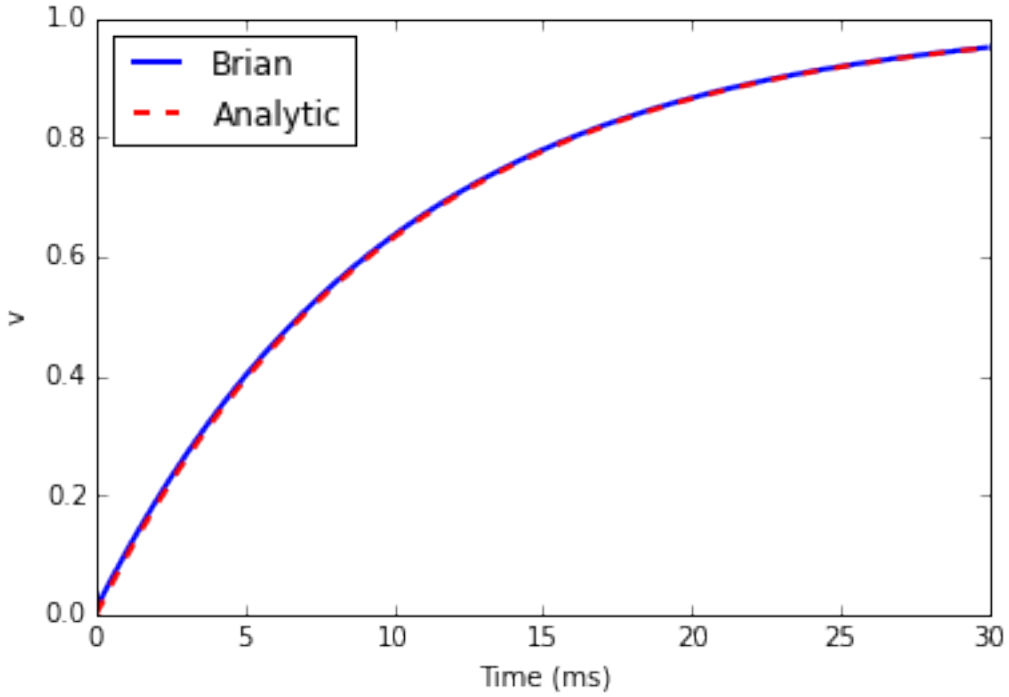


This time we only ran the simulation for 30 ms so that we can see the behaviour better. It looks like it's behaving as expected, but let's just check that analytically by plotting the expected behaviour on top.

```
start_scope()

G = NeuronGroup(1, eqs)
M = StateMonitor(G, 'v', record=0)

run(30*ms)

plot(M.t/ms, M.v[0], '-b', lw=2, label='Brian')
plot(M.t/ms, 1-exp(-M.t/tau), '--r', lw=2, label='Analytic')
xlabel('Time (ms)')
```

```
ylabel('v')
legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x7ff095e9c510>
```



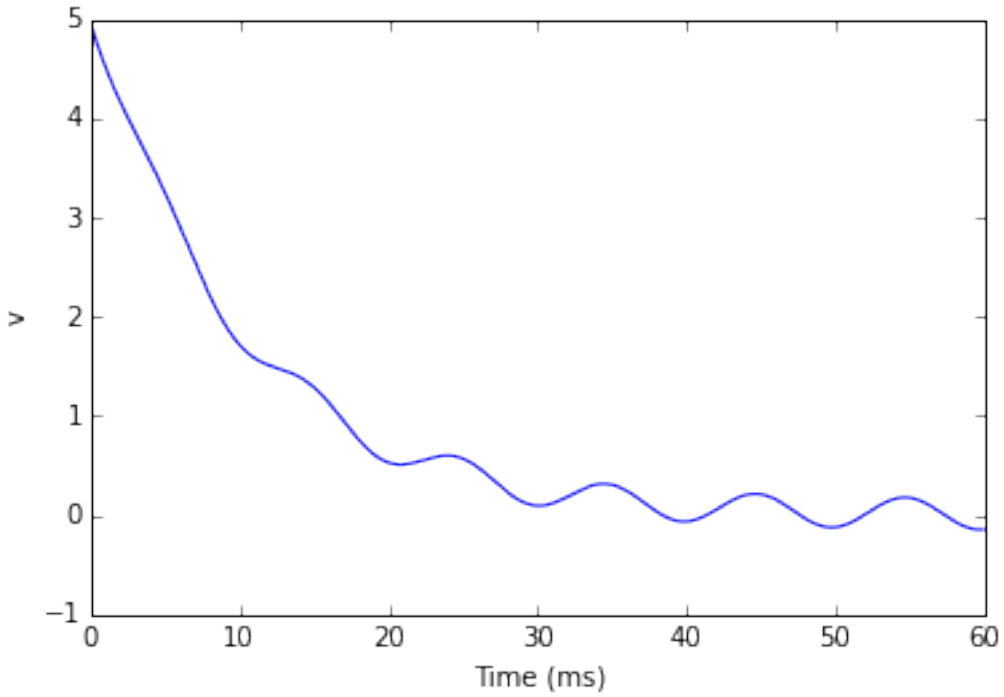
As you can see, the blue (Brian) and dashed red (analytic solution) lines coincide.

In this example, we used the object `StateMonitor` object. This is used to record the values of a neuron variable while the simulation runs. The first two arguments are the group to record from, and the variable you want to record from. We also specify `record=0`. This means that we record all values for neuron 0. We have to specify which neurons we want to record because in large simulations with many neurons it usually uses up too much RAM to record the values of all neurons.

Now try modifying the equations and parameters and see what happens in the cell below.

```
start_scope()

tau = 10*ms
eqs = '''
dv/dt = (sin(2*pi*100*Hz*t)-v)/tau : 1
'''

G = NeuronGroup(1, eqs, method='euler') # TODO: we shouldn't have to specify euler here
M = StateMonitor(G, 'v', record=0)

G.v = 5 # initial value

run(60*ms)

plot(M.t/ms, M.v[0])
xlabel('Time (ms)')
ylabel('v')
```

```
<matplotlib.text.Text at 0x7ff096190cd0>
```

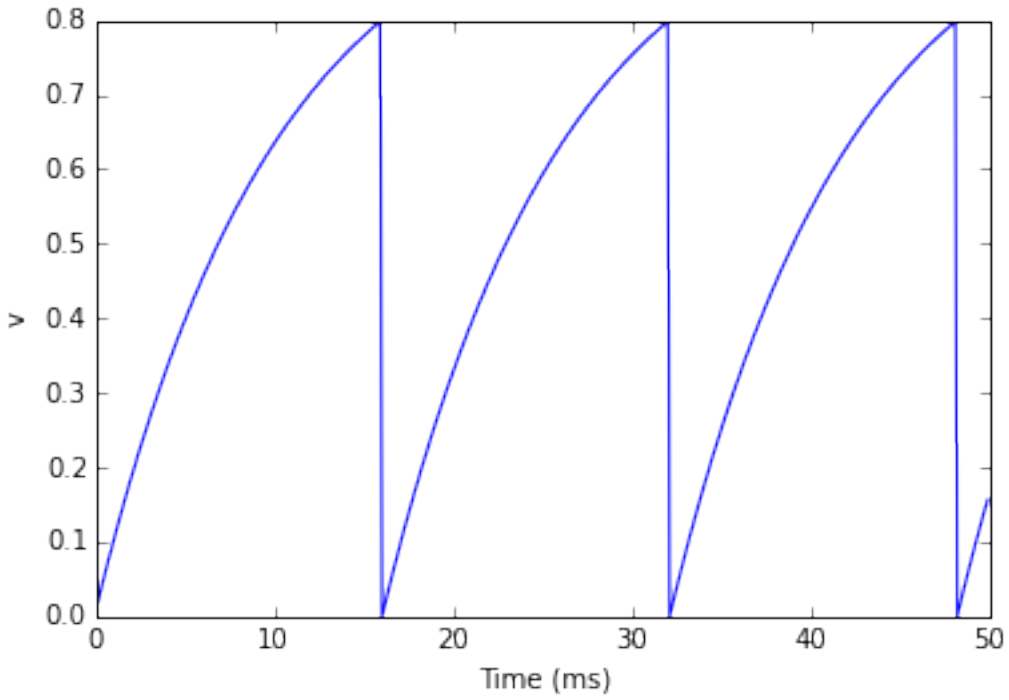


2.1.3 Adding spikes

So far we haven't done anything neuronal, just played around with differential equations. Now let's start adding spiking behaviour.

```
start_scope()

tau = 10*ms
eqs = '''
dv/dt = (1-v)/tau : 1
'''

G = NeuronGroup(1, eqs, threshold='v>0.8', reset='v = 0')

M = StateMonitor(G, 'v', record=0)
run(50*ms)
plot(M.t/ms, M.v[0])
xlabel('Time (ms)')
ylabel('v')
```

```
<matplotlib.text.Text at 0x7ff09612bd90>
```



We've added two new keywords to the `NeuronGroup` declaration: `threshold='v>0.8'` and `reset='v = 0'`. What this means is that when `v>1` we fire a spike, and immediately reset `v = 0` after the spike. We can put any expression and series of statements as these strings.

As you can see, at the beginning the behaviour is the same as before until `v` crosses the threshold `v>0.8` at which point you see it reset to 0. You can't see it in this figure, but internally Brian has registered this event as a spike. Let's have a look at that.

```
start_scope()

G = NeuronGroup(1, eqs, threshold='v>0.8', reset='v = 0')

spikemon = SpikeMonitor(G)

run(50*ms)

print 'Spike times:', spikemon.t[:]
```

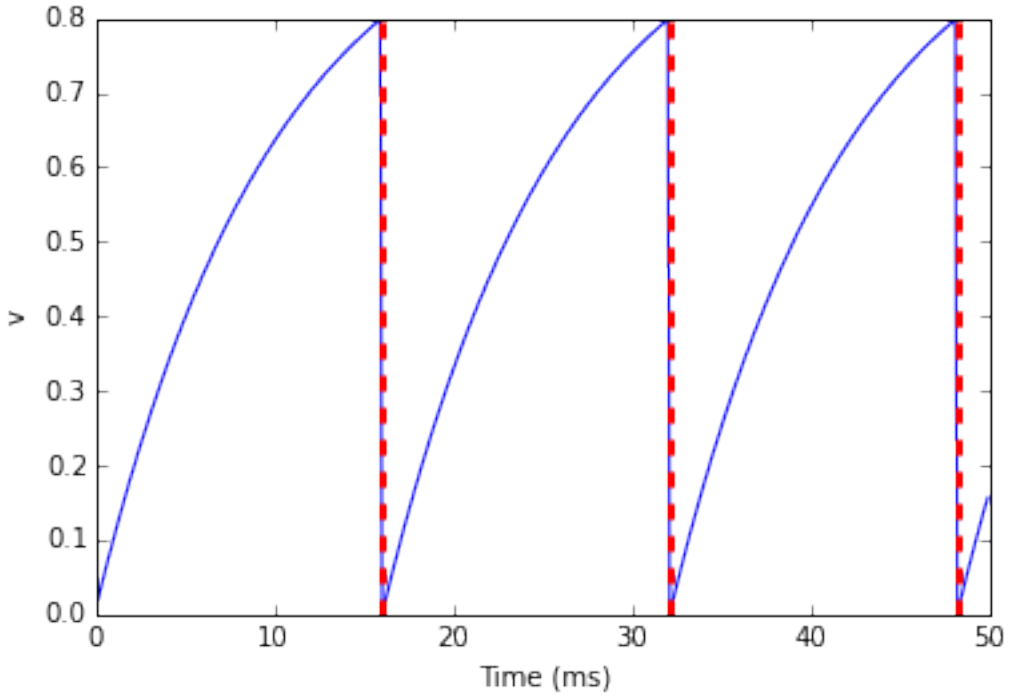
```
Spike times: [ 16.   32.1  48.2] ms
```

The `SpikeMonitor` object takes the group whose spikes you want to record as its argument and stores the spike times in the variable `t`. Let's plot those spikes on top of the other figure to see that it's getting it right.

```
start_scope()

G = NeuronGroup(1, eqs, threshold='v>0.8', reset='v = 0')

statemon = StateMonitor(G, 'v', record=0)
spikemon = SpikeMonitor(G)

run(50*ms)

plot(statemon.t/ms, statemon.v[0])
for t in spikemon.t:
```

```
    axvline(t/ms, ls='--', c='r', lw=3)
xlabel('Time (ms)')
ylabel('v')
```

```
<matplotlib.text.Text at 0x7ff095649a50>
```



Here we've used the `axvline` command from `matplotlib` to draw a red, dashed vertical line at the time of each spike recorded by the `SpikeMonitor`.

Now try changing the strings for `threshold` and `reset` in the cell above to see what happens.

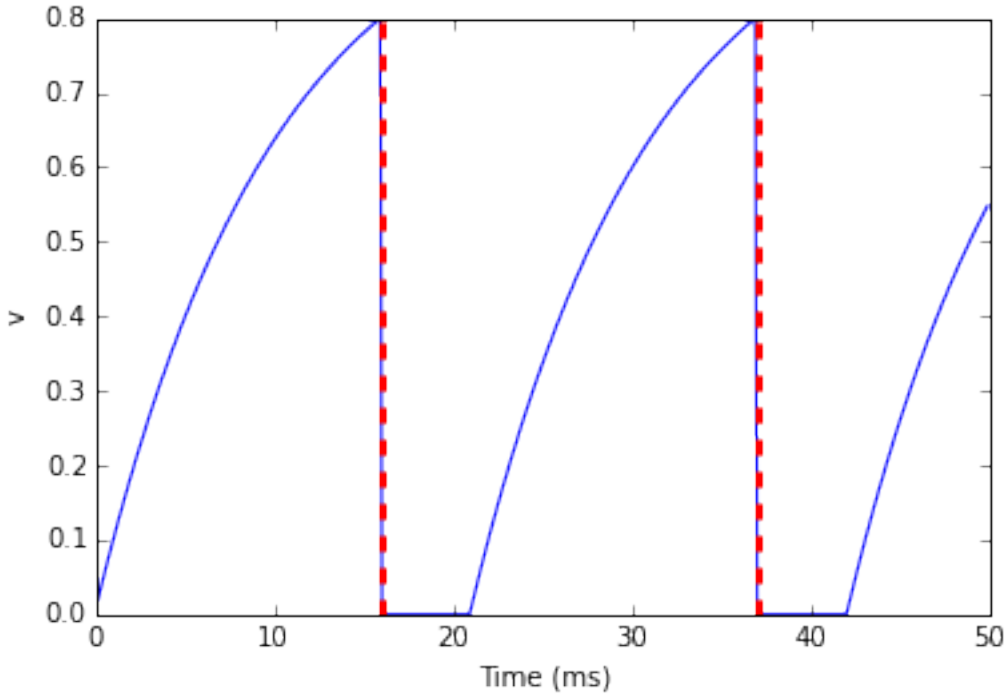
2.1.4 Refractoriness

A common feature of neuron models is refractoriness. This means that after the neuron fires a spike it becomes refractory for a certain duration and cannot fire another spike until this period is over. Here's how we do that in Brian.

```
start_scope()

tau = 10*ms
eqs = '''
dv/dt = (1-v)/tau : 1 (unless refractory)
'''

G = NeuronGroup(1, eqs, threshold='v>0.8', reset='v = 0', refractory=5*ms)

statemon = StateMonitor(G, 'v', record=0)
spikemon = SpikeMonitor(G)

run(50*ms)

plot(statemon.t/ms, statemon.v[0])
for t in spikemon.t:
```

```
    axvline(t/ms, ls='--', c='r', lw=3)
xlabel('Time (ms)')
ylabel('v')
```

```
<matplotlib.text.Text at 0x7ff0956f6290>
```



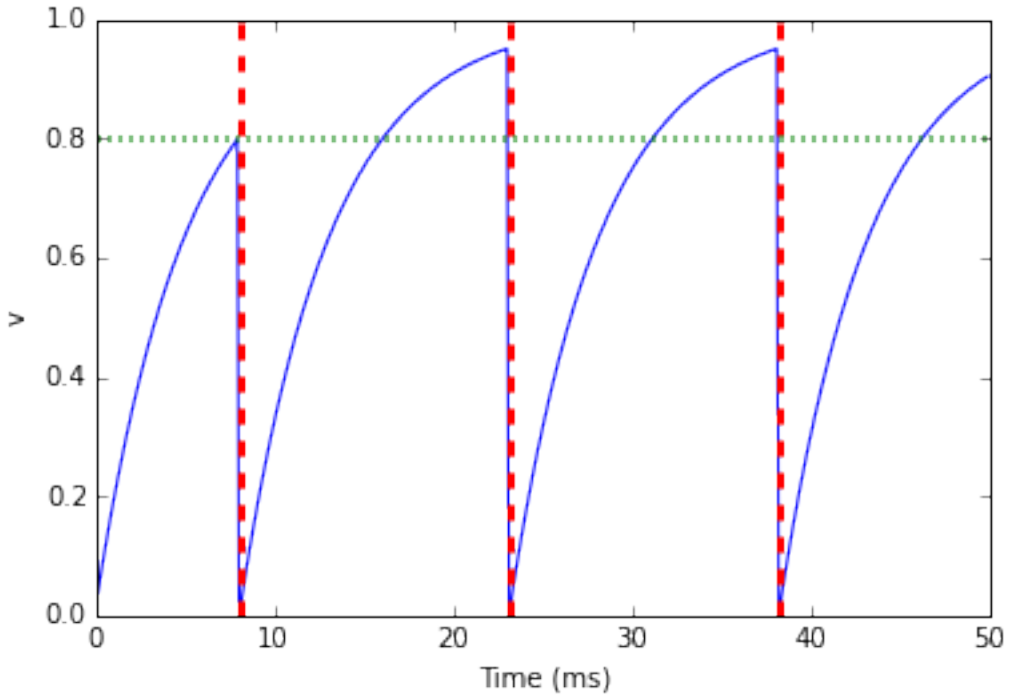
As you can see in this figure, after the first spike, `v` stays at 0 for around 5 ms before it resumes its normal behaviour. To do this, we've done two things. Firstly, we've added the keyword `refractory=5*ms` to the `NeuronGroup` declaration. On its own, this only means that the neuron cannot spike in this period (see below), but doesn't change how `v` behaves. In order to make `v` stay constant during the refractory period, we have to add `(unless refractory)` to the end of the definition of `v` in the differential equations. What this means is that the differential equation determines the behaviour of `v` unless it's refractory in which case it is switched off.

Here's what would happen if we didn't include `(unless refractory)`. Note that we've also decreased the value of `tau` and increased the length of the refractory period to make the behaviour clearer.

```
start_scope()

tau = 5*ms
eqs = '''
dv/dt = (1-v)/tau : 1
'''

G = NeuronGroup(1, eqs, threshold='v>0.8', reset='v = 0', refractory=15*ms)

statemon = StateMonitor(G, 'v', record=0)
spikemon = SpikeMonitor(G)

run(50*ms)

plot(statemon.t/ms, statemon.v[0])
for t in spikemon.t:
    axvline(t/ms, ls='--', c='r', lw=3)
```

```
axhline(0.8, ls=':', c='g', lw=3)
xlabel('Time (ms)')
ylabel('v')
print "Spike times:", spikemon.t[:]
```

```
Spike times: [  8.   23.1  38.2] ms
```



So what's going on here? The behaviour for the first spike is the same: `v` rises to 0.8 and then the neuron fires a spike at time 8 ms before immediately resetting to 0. Since the refractory period is now 15 ms this means that the neuron won't be able to spike again until time 8 + 15 = 23 ms. Immediately after the first spike, the value of `v` now instantly starts to rise because we didn't specify `(unless refractory)` in the definition of `dv/dt`. However, once it reaches the value 0.8 (the dashed green line) at time roughly 8 ms it doesn't fire a spike even though the threshold is `v>0.8`. This is because the neuron is still refractory until time 23 ms, at which point it fires a spike.

Note that you can do more complicated and interesting things with refractoriness. See the full documentation for more details about how it works.

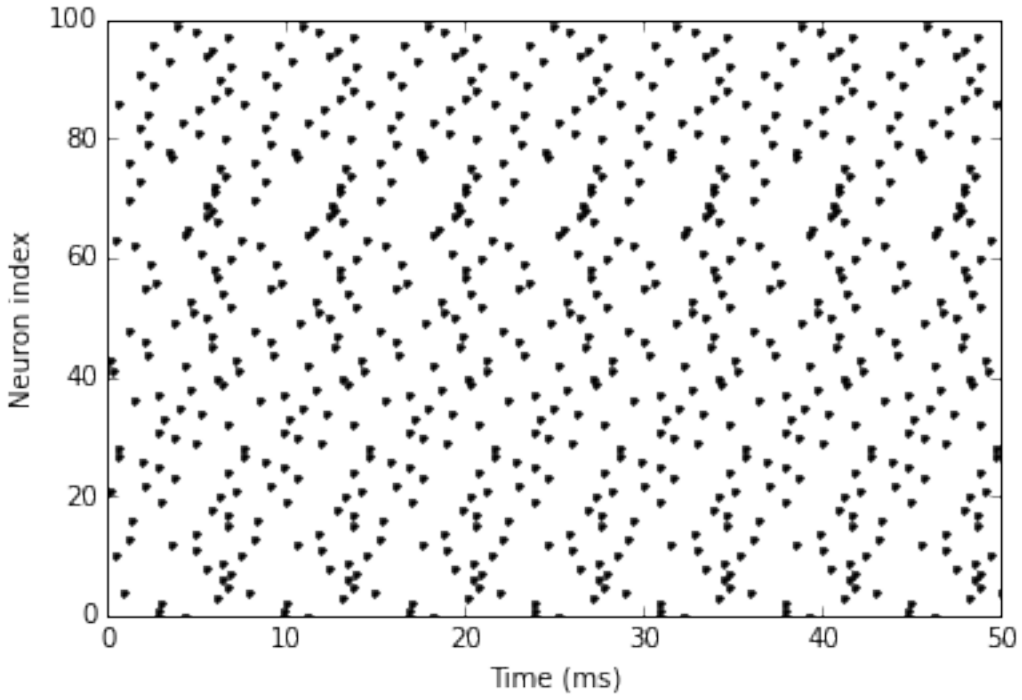
2.1.5 Multiple neurons

So far we've only been working with a single neuron. Let's do something interesting with multiple neurons.

```
start_scope()

N = 100
tau = 10*ms
eqs = '''
dv/dt = (2-v)/tau : 1
'''

G = NeuronGroup(N, eqs, threshold='v>1', reset='v=0')
G.v = 'rand()'
```

```
spikemon = SpikeMonitor(G)

run(50*ms)

plot(spikemon.t/ms, spikemon.i, '.k')
xlabel('Time (ms)')
ylabel('Neuron index')
```

```
<matplotlib.text.Text at 0x7ff0937e8b50>
```



This shows a few changes. Firstly, we've got a new variable `N` determining the number of neurons. Secondly, we added the statement `G.v = 'rand()'` before the run. What this does is initialise each neuron with a different uniform random value between 0 and 1. We've done this just so each neuron will do something a bit different. The other big change is how we plot the data in the end.

As well as the variable `spikemon.t` with the times of all the spikes, we've also used the variable `spikemon.i` which gives the corresponding neuron index for each spike, and plotted a single black dot with time on the x-axis and neuron index on the y-value. This is the standard "raster plot" used in neuroscience.

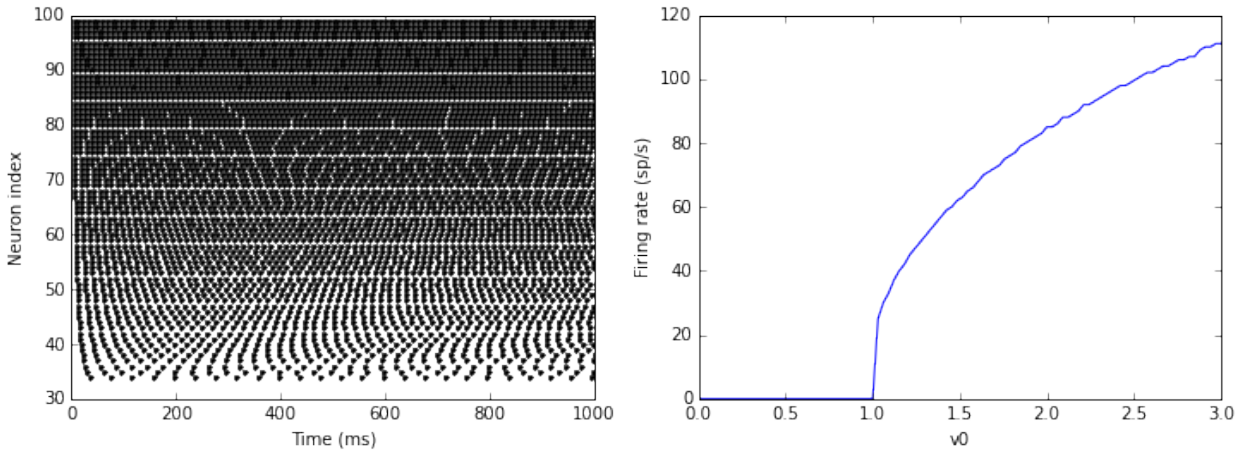
2.1.6 Parameters

To make these multiple neurons do something more interesting, let's introduce per-neuron parameters that don't have a differential equation attached to them.

```
start_scope()

N = 100
tau = 10*ms
v0_max = 3.
duration = 1000*ms

eqs = '''
```

```
dv/dt = (v0-v)/tau : 1 (unless refractory)
v0 : 1
'''

G = NeuronGroup(N, eqs, threshold='v>1', reset='v=0', refractory=5*ms)
M = SpikeMonitor(G)

G.v0 = 'i*v0_max/(N-1)'

run(duration)

figure(figsize=(12,4))
subplot(121)
plot(M.t/ms, M.i, '.k')
xlabel('Time (ms)')
ylabel('Neuron index')
subplot(122)
plot(G.v0, M.count/duration)
xlabel('v0')
ylabel('Firing rate (sp/s)')
```

```
<matplotlib.text.Text at 0x7ff092b21290>
```



The line `v0 : 1` declares a new per-neuron parameter `v0` with units `1` (i.e. dimensionless).

The line `G.v0 = 'i*v0_max/(N-1)'` initialises the value of v0 for each neuron varying from 0 up to `v0_max`. The symbol `i` when it appears in strings like this refers to the neuron index.

So in this example, we're driving the neuron towards the value `v0` exponentially, but we fire spikes when `v` crosses `v>1` it fires a spike and resets. The effect is that the rate at which it fires spikes will be related to the value of `v0`. For `v0<1` it will never fire a spike, and as `v0` gets larger it will fire spikes at a higher rate. The right hand plot shows the firing rate as a function of the value of `v0`. This is the I-f curve of this neuron model.

Note that in the plot we've used the `count` variable of the `SpikeMonitor`: this is an array of the number of spikes each neuron in the group fired. Dividing this by the duration of the run gives the firing rate.

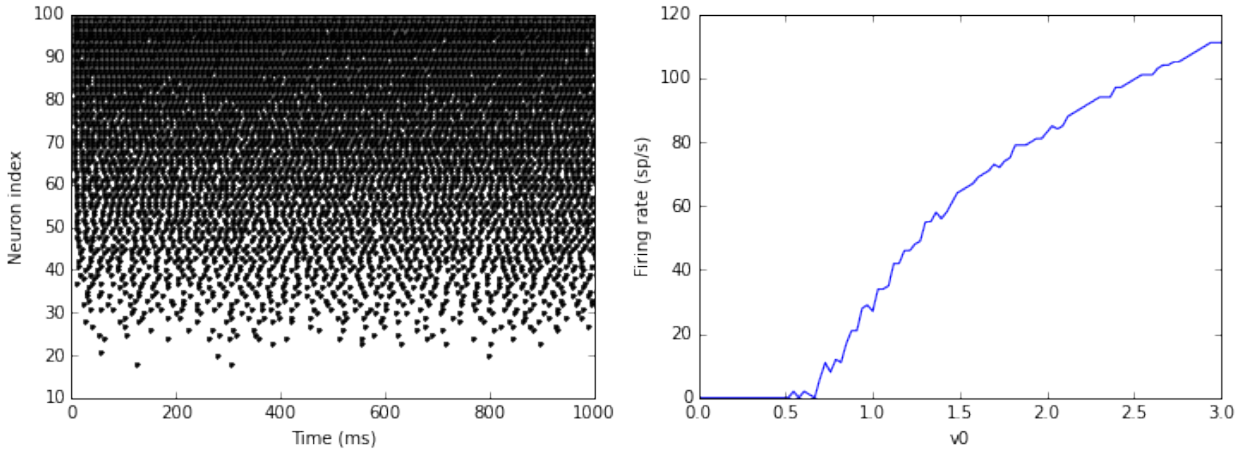
2.1.7 Stochastic neurons

Often when making models of neurons, we include a random element to model the effect of various forms of neural noise. In Brian, we can do this by using the symbol `xi` in differential equations. Strictly speaking, this symbol is a "stochastic differential" but you can sort of thinking of it as just a Gaussian random variable with mean 0 and standard

deviation 1. We do have to take into account the way stochastic differentials scale with time, which is why we multiply it by `tau**-0.5` in the equations below (see a textbook on stochastic differential equations for more details).

```
start_scope()

N = 100
tau = 10*ms
v0_max = 3.
duration = 1000*ms
sigma = 0.2

eqs = '''
dv/dt = (v0-v)/tau+sigma*xi*tau**-0.5 : 1 (unless refractory)
v0 : 1
'''

G = NeuronGroup(N, eqs, threshold='v>1', reset='v=0', refractory=5*ms)
M = SpikeMonitor(G)

G.v0 = 'i*v0_max/(N-1)'

run(duration)

figure(figsize=(12,4))
subplot(121)
plot(M.t/ms, M.i, '.k')
xlabel('Time (ms)')
ylabel('Neuron index')
subplot(122)
plot(G.v0, M.count/duration)
xlabel('v0')
ylabel('Firing rate (sp/s)')
```

```
<matplotlib.text.Text at 0x7ff0929db710>
```



That's the same figure as in the previous section but with some noise added. Note how the curve has changed shape: instead of a sharp jump from firing at rate 0 to firing at a positive rate, it now increases in a sigmoidal fashion. This is because no matter how small the driving force the randomness may cause it to fire a spike.

2.1.8 End of tutorial

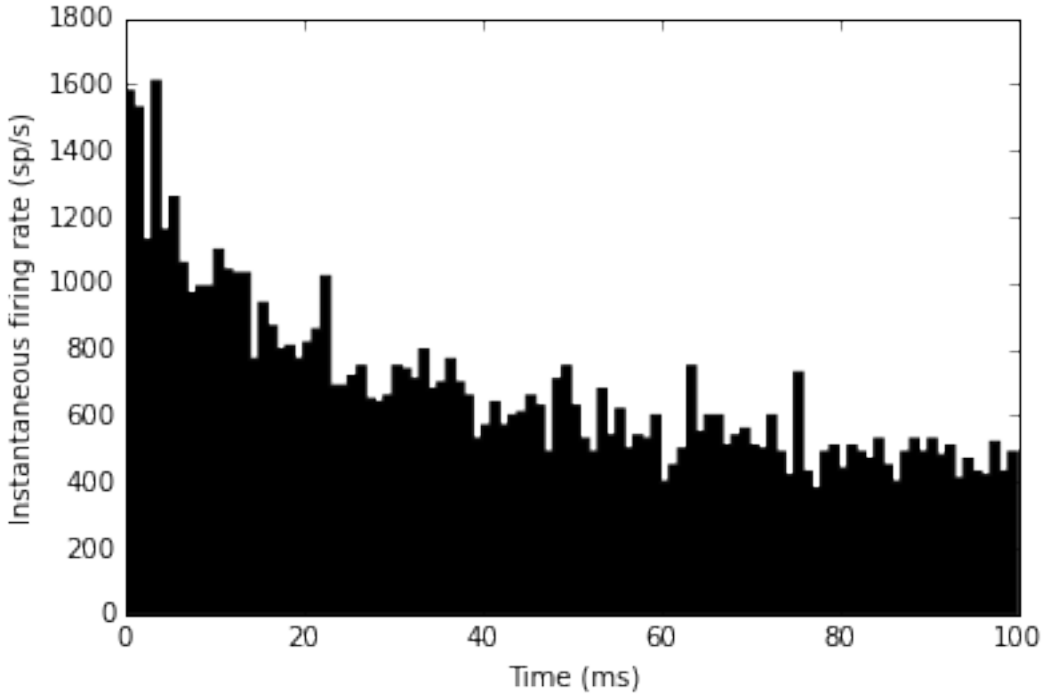
That's the end of this part of the tutorial. The cell below has another example. See if you can work out what it is doing and why. Try adding a `StateMonitor` to record the values of the variables for one of the neurons to help you understand it.

You could also try out the things you've learned in this cell.

Once you're done with that you can move on to the next tutorial on Synapses.

```
start_scope()

N = 1000
tau = 10*ms
vr = -70*mV
vt0 = -50*mV
delta_vt0 = 5*mV
tau_t = 100*ms
sigma = 0.5*(vt0-vr)
v_drive = 2*(vt0-vr)
duration = 100*ms

eqs = '''
dv/dt = (v_drive+vr-v)/tau + sigma*xi*tau**-0.5 : volt
dvt/dt = (vt0-vt)/tau_t : volt
'''

reset = '''
v = vr
vt += delta_vt0
'''

G = NeuronGroup(N, eqs, threshold='v>vt', reset=reset, refractory=5*ms)
spikemon = SpikeMonitor(G)

G.v = 'rand()*(vt0-vr)+vr'
G.vt = vt0

run(duration)

_ = hist(spikemon.t/ms, 100, histtype='stepfilled', facecolor='k', weights=ones(len(spikemon))/(N*de
xlabel('Time (ms)')
ylabel('Instantaneous firing rate (sp/s)')
```

```
<matplotlib.text.Text at 0x7ff0937dd650>
```



2.2 Introduction to Brian part 2: Synapses

Note: This tutorial is written as an interactive notebook that should be run on your own computer. See the tutorial overview page for more details.

Download link for this tutorial: `2-intro-to-brian-synapses.ipynb`.

If you haven't yet read part 1: Neurons, go read that now.

As before we start by importing the Brian package and setting up matplotlib for IPython:

```
from brian2 import *
%matplotlib inline
```

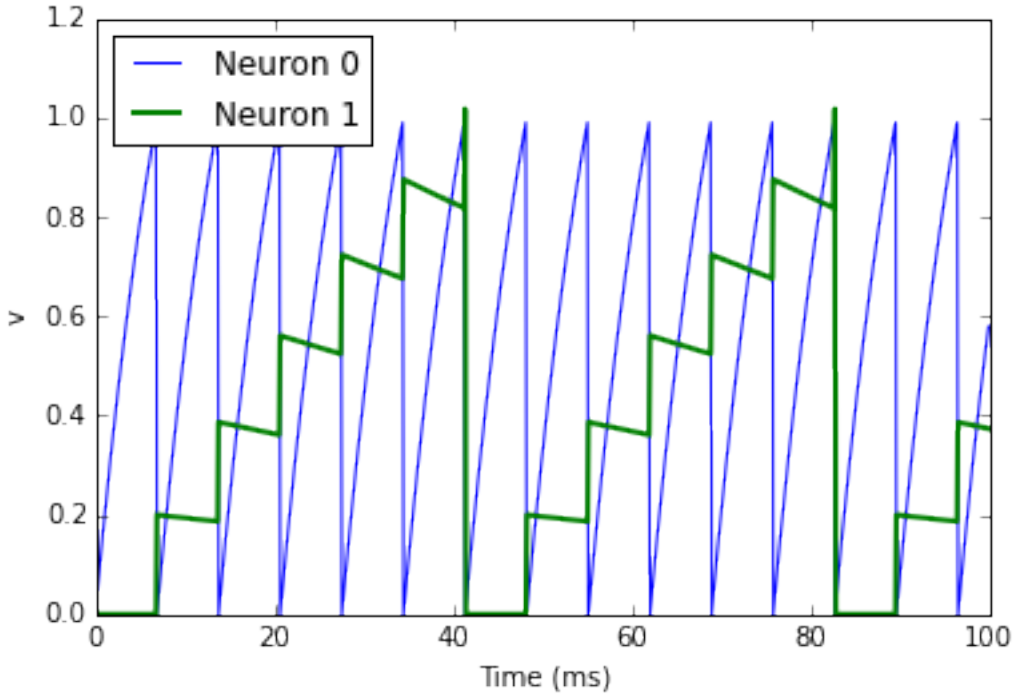
2.2.1 The simplest Synapse

Once you have some neurons, the next step is to connect them up via synapses. We'll start out with doing the simplest possible type of synapse that causes an instantaneous change in a variable after a spike.

```
start_scope()

eqs = '''
dv/dt = (I-v)/tau : 1
I : 1
tau : second
'''
G = NeuronGroup(2, eqs, threshold='v>1', reset='v = 0')
G.I = [2, 0]
G.tau = [10, 100]*ms
```

```
# Comment these two lines out to see what happens without Synapses
S = Synapses(G, G, pre='v_post += 0.2')
S.connect(0, 1)

M = StateMonitor(G, 'v', record=True)

run(100*ms)

plot(M.t/ms, M.v[0], '-b', label='Neuron 0')
plot(M.t/ms, M.v[1], '-g', lw=2, label='Neuron 1')
xlabel('Time (ms)')
ylabel('v')
legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x7f51b14442d0>
```



There are a few things going on here. First of all, let's recap what is going on with the `NeuronGroup`. We've created two neurons, each of which has the same differential equation but different values for parameters I and tau. Neuron 0 has `I=2` and `tau=10*ms` which means that is driven to repeatedly spike at a fairly high rate. Neuron 1 has `I=0` and `tau=100*ms` which means that on its own - without the synapses - it won't spike at all (the driving current I is 0). You can prove this to yourself by commenting out the two lines that define the synapse.

Next we define the synapses: `Synapses(source, target, ...)` means that we are defining a synaptic model that goes from `source` to `target`. In this case, the source and target are both the same, the group `G`. The syntax `pre='v_post += 0.2'` means that when a spike occurs in the presynaptic neuron (hence `pre`) it causes an instantaneous change to happen `v_post += 0.2`. The `_post` means that the value of `v` referred to is the post-synaptic value, and it is increased by 0.2. So in total, what this model says is that whenever two neurons in G are connected by a synapse, when the source neuron fires a spike the target neuron will have its value of `v` increased by 0.2.

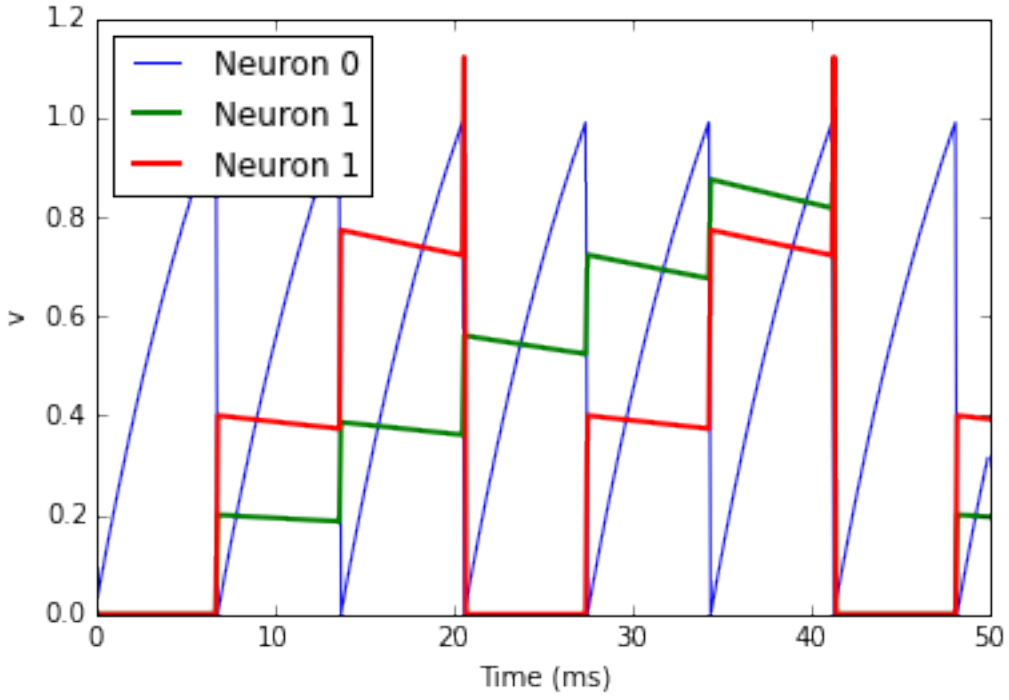
However, at this point we have only defined the synapse model, we haven't actually created any synapses. The next line `S.connect(0,1)` creates a synapse from neuron 0 to neuron 1.

2.2.2 Adding a weight

In the previous section, we hard coded the weight of the synapse to be the value 0.2, but often we would to allow this to be different for different synapses. We do that by introducing synapse equations.

```
start_scope()

eqs = '''
dv/dt = (I-v)/tau : 1
I : 1
tau : second
'''
G = NeuronGroup(3, eqs, threshold='v>1', reset='v = 0')
G.I = [2, 0, 0]
G.tau = [10, 100, 100]*ms

# Comment these two lines out to see what happens without Synapses
S = Synapses(G, G, 'w : 1', pre='v_post += w')
S.connect(0, [1, 2])
S.w = 'j*0.2'

M = StateMonitor(G, 'v', record=True)

run(50*ms)

plot(M.t/ms, M.v[0], '-b', label='Neuron 0')
plot(M.t/ms, M.v[1], '-g', lw=2, label='Neuron 1')
plot(M.t/ms, M.v[2], '-r', lw=2, label='Neuron 1')
xlabel('Time (ms)')
ylabel('v')
legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x7f51b06cef50>
```



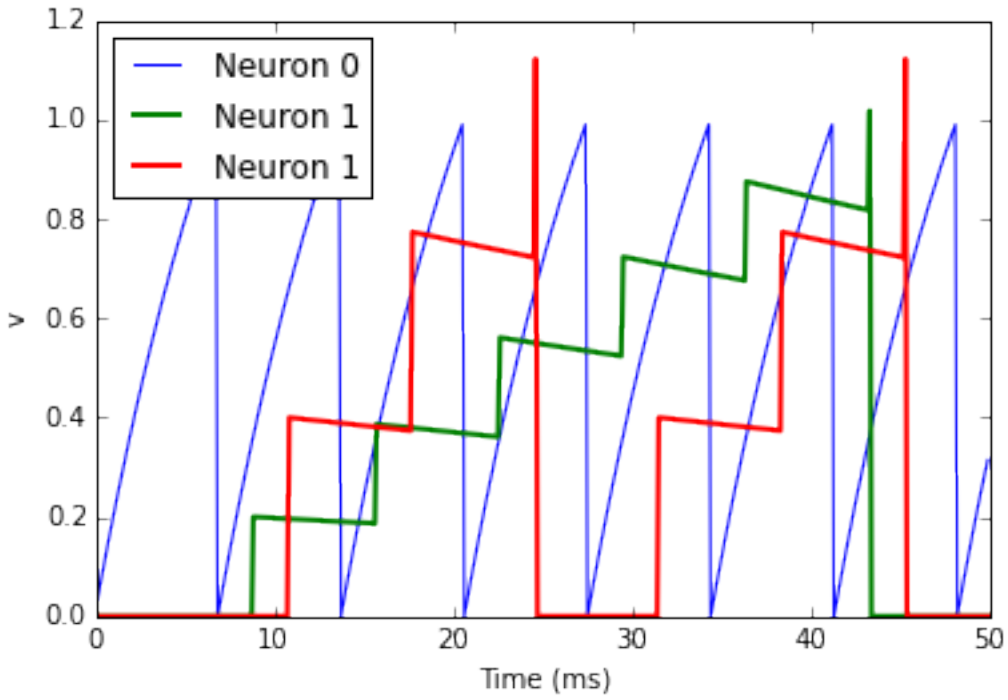
This example behaves very similarly to the previous example, but now there's a synaptic weight variable `w`. The string `'w : 1'` is an equation string, precisely the same as for neurons, that defines a single dimensionless parameter `w`. We changed the behaviour on a spike to `pre='v_post += w'` now, so that each synapse can behave differently depending on the value of `w`. To illustrate this, we've made a third neuron which behaves precisely the same as the second neuron, and connected neuron 0 to both neurons 1 and 2. We've also set the weights via `S.w = 'j*0.2'`. When `i` and `j` occur in the context of synapses, `i` refers to the source neuron index, and `j` to the target neuron index. So this will give a synaptic connection from 0 to 1 with weight `0.2=0.2*1` and from 0 to 2 with weight `0.4=0.2*2`.

2.2.3 Introducing a delay

So far, the synapses have been instantaneous, but we can also make them act with a certain delay.

```
start_scope()

eqs = '''
dv/dt = (I-v)/tau : 1
I : 1
tau : second
'''
G = NeuronGroup(3, eqs, threshold='v>1', reset='v = 0')
G.I = [2, 0, 0]
G.tau = [10, 100, 100]*ms

S = Synapses(G, G, 'w : 1', pre='v_post += w')
S.connect(0, [1, 2])
S.w = 'j*0.2'
S.delay = 'j*2*ms'

M = StateMonitor(G, 'v', record=True)

run(50*ms)

plot(M.t/ms, M.v[0], '-b', label='Neuron 0')
plot(M.t/ms, M.v[1], '-g', lw=2, label='Neuron 1')
plot(M.t/ms, M.v[2], '-r', lw=2, label='Neuron 1')
xlabel('Time (ms)')
ylabel('v')
legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x7f51b042a490>
```



As you can see, that's as simple as adding a line `S.delay = 'j*2*ms'` so that the synapse from 0 to 1 has a delay of 2 ms, and from 0 to 2 has a delay of 4 ms.

2.2.4 More complex connectivity

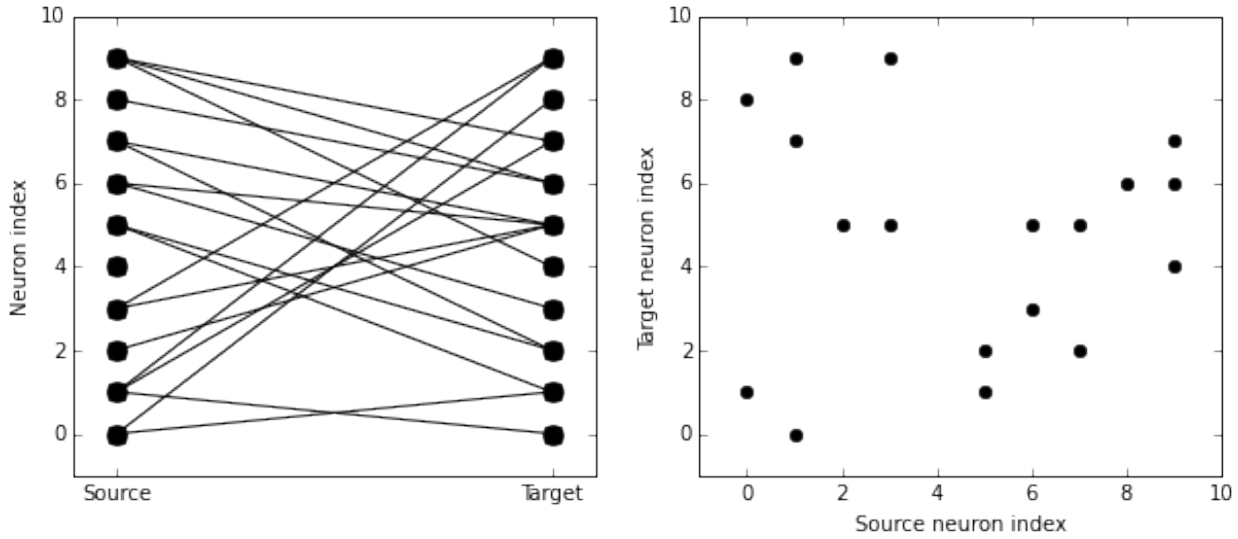
So far, we specified the synaptic connectivity explicitly, but for larger networks this isn't usually possible. For that, we usually want to specify some condition.

```
start_scope()

N = 10
G = NeuronGroup(N, 'v:1')
S = Synapses(G, G)
S.connect('i!=j', p=0.2)
```

Here we've created a dummy neuron group of N neurons and a dummy synapses model that doens't actually do anything just to demonstrate the connectivity. The line `S.connect('i!=j', p=0.2)` will connect all pairs of neurons `i` and `j` with probability 0.2 as long as the condition `i!=j` holds. You could equivalently write `S.connect('i!=j and rand()<0.2')` if you wanted to. So, how can we see that connectivity? Here's a little function that will let us visualise it.

```
def visualise_connectivity(S):
    Ns = len(S.source)
    Nt = len(S.target)
    figure(figsize=(10, 4))
    subplot(121)
    plot(zeros(Ns), arange(Ns), 'ok', ms=10)
    plot(ones(Nt), arange(Nt), 'ok', ms=10)
    for i, j in zip(S.i, S.j):
        plot([0, 1], [i, j], '-k')
    xticks([0, 1], ['Source', 'Target'])
    ylabel('Neuron index')
```

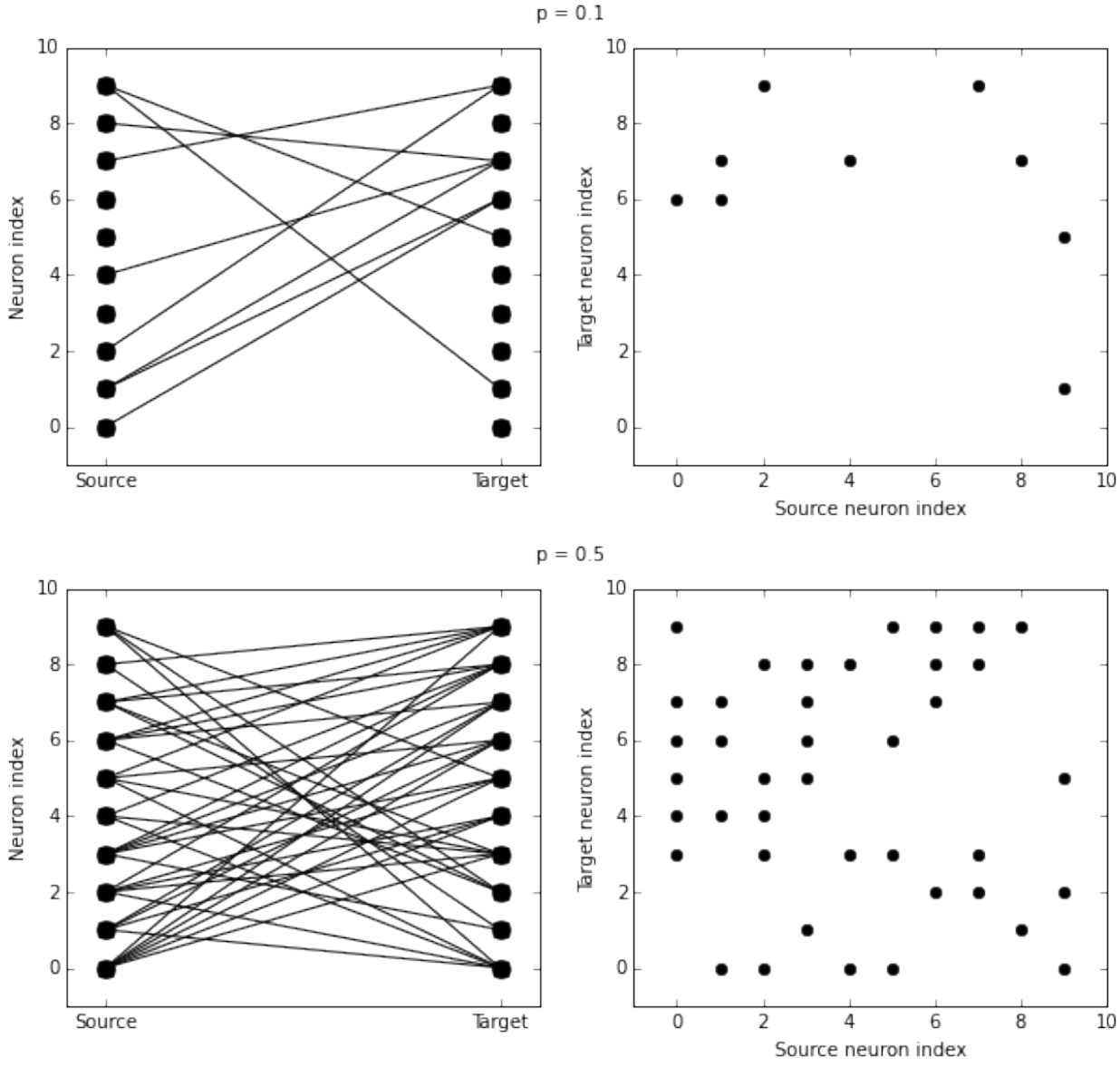
```
    xlim(-0.1, 1.1)
    ylim(-1, max(Ns, Nt))
    subplot(122)
    plot(S.i, S.j, 'ok')
    xlim(-1, Ns)
    ylim(-1, Nt)
    xlabel('Source neuron index')
    ylabel('Target neuron index')

visualise_connectivity(S)
```

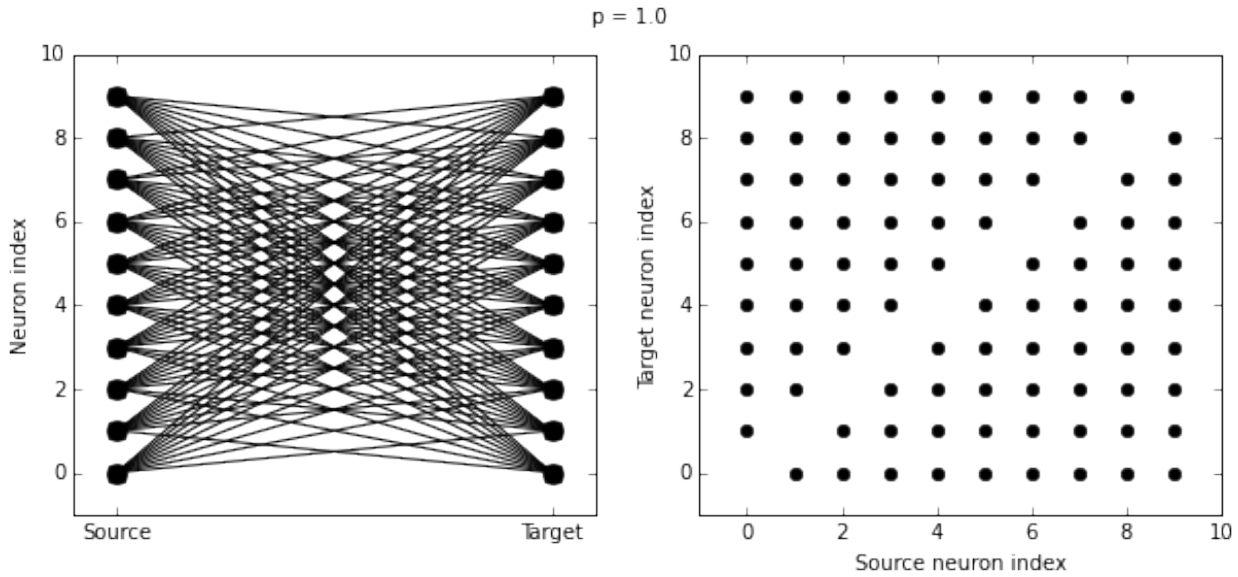


There are two plots here. On the left hand side, you see a vertical line of circles indicating source neurons on the left, and a vertical line indicating target neurons on the right, and a line between two neurons that have a synapse. On the right hand side is another way of visualising the same thing. Here each black dot is a synapse, with x value the source neuron index, and y value the target neuron index.

Let's see how these figures change as we change the probability of a connection:

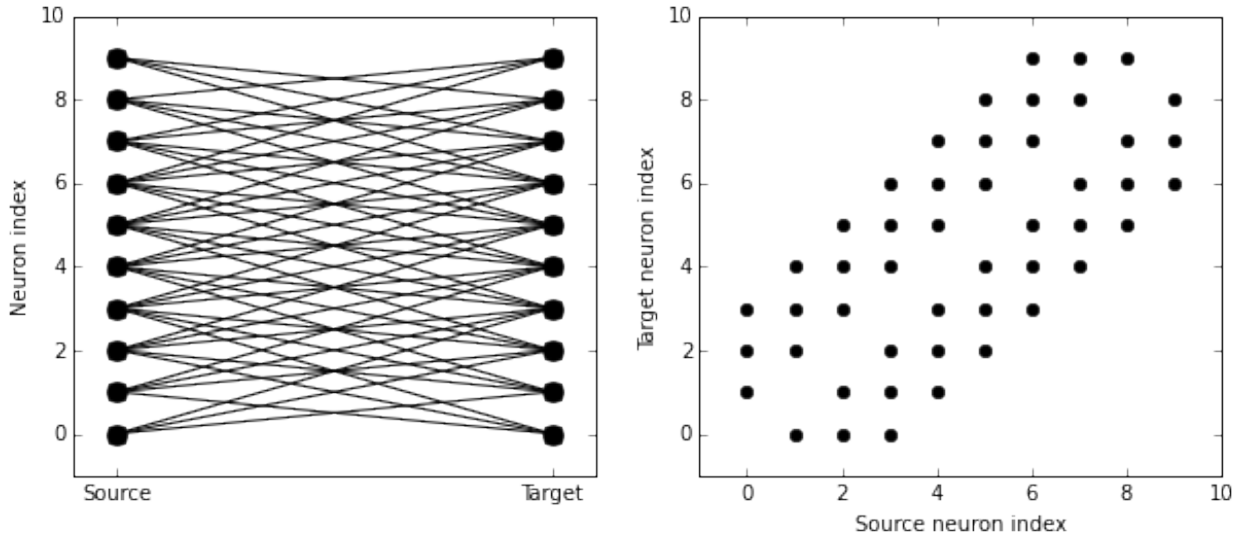
```
start_scope()

N = 10
G = NeuronGroup(N, 'v:1')

for p in [0.1, 0.5, 1.0]:
    S = Synapses(G, G)
    S.connect('i!=j', p=p)
    visualise_connectivity(S)
    suptitle('p = '+str(p))
```


p = 0.1
Neuron index
Source
Target
Target neuron index
Source neuron index
p = 0.5
Neuron index
Source
Target
Target neuron index
Source neuron index



And let's see what another connectivity condition looks like. This one will only connect neighbouring neurons.

```
start_scope()

N = 10
G = NeuronGroup(N, 'v:1')

S = Synapses(G, G)
S.connect('abs(i-j)<4 and i!=j')
visualise_connectivity(S)
```



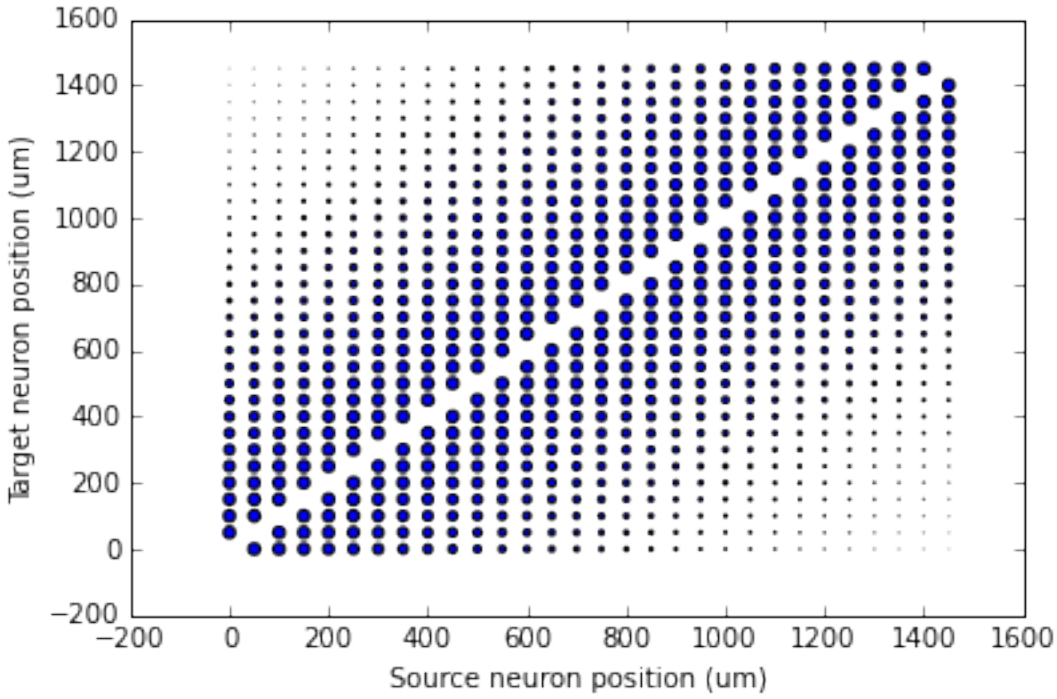
Try using that cell to see how other connectivity conditions look like.

This way of specifying connectivity is very general, and can also be used for specifying the value of weights for example. Let's see an example where we assign each neuron a spatial location and have a distance-dependent connectivity function. We visualise the weight of a synapse by the size of the marker.

```
start_scope()
```

```
N = 30
neuron_spacing = 50*umetre
width = N/4.0*neuron_spacing

# Neuron has one variable x, its position
G = NeuronGroup(N, 'x : metre')
G.x = 'i*neuron_spacing'

# All synapses are connected (excluding self-connections)
S = Synapses(G, G, 'w : 1')
S.connect('i!=j')
# Weight varies with distance
S.w = 'exp(-(x_pre-x_post)**2/(2*width**2))'

scatter(G.x[S.i]/um, G.x[S.j]/um, S.w*20)
xlabel('Source neuron position (um)')
ylabel('Target neuron position (um)')
```

```
<matplotlib.text.Text at 0x7f51ad6ebc10>
```



Now try changing that function and seeing how the plot changes.

2.2.5 More complex synapse models: STDP

Brian's synapse framework is very general and can do things like short-term plasticity (STP) or spike-timing dependent plasticity (STDP). Let's see how that works for STDP.

STDP is normally defined by an equation something like this:

$$\Delta w = \sum_{t_{pre}} \sum_{t_{post}} W(t_{post} - t_{pre})$$

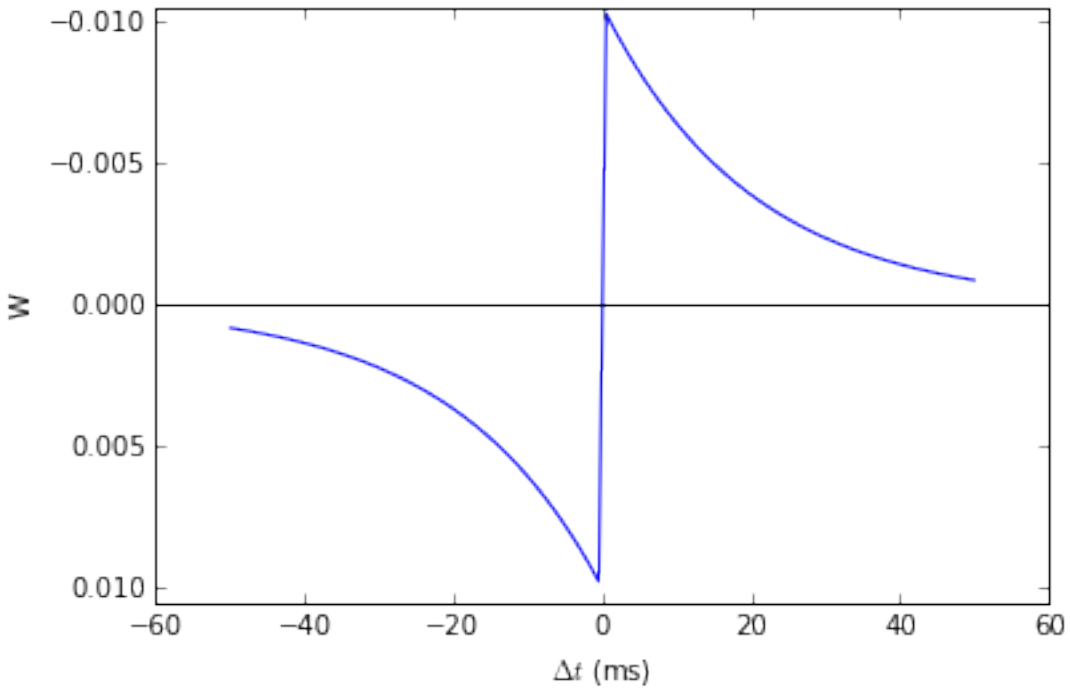
That is, the change in synaptic weight w is the sum over all presynaptic spike times t_{pre} and postsynaptic spike times t_{post} of some function W of the difference in these spike times. A commonly used function W is:

$$W(\Delta t) = \begin{cases} A_{pre} e^{-\Delta t/\tau_{pre}} & \Delta t > 0 \\ A_{post} - e^{\Delta t/\tau_{pre}} & \Delta t < 0 \end{cases}$$

This function looks like this:

```
tau_pre = tau_post = 20*ms
A_pre = 0.01
A_post = -A_pre*1.05
delta_t = linspace(-50, 50, 100)*ms
W = where(delta_t<0, A_pre*exp(delta_t/tau_pre), A_post*exp(-delta_t/tau_post))
plot(delta_t/ms, W)
xlabel(r'$\Delta t$ (ms)')
ylabel('W')
ylim(-A_post, A_post)
axhline(0, ls='-', c='k')
```

```
<matplotlib.lines.Line2D at 0x7f51af4b1310>
```



Simulating it directly using this equation though would be very inefficient, because we would have to sum over all pairs of spikes. That would also be physiologically unrealistic because the neuron cannot remember all its previous spike times. It turns out there is a more efficient and physiologically more plausible way to get the same effect.

We define two new variables a_{pre} and a_{post} which are "traces" of pre- and post-synaptic activity, governed by the

differential equations:

$$
\begin{aligned}
& & to \\
\tau_{pre}\frac{\mathrm{d}}{\mathrm{d}t}a_{pre} &= \\
& & -a_{pre} \\
\tau_{post}\frac{\mathrm{d}}{\mathrm{d}t}a_{post} &= \\
& & -a_{post}
\end{aligned}
\tag{2.1}
$$

$$
\begin{aligned}
&= \\
&=-a_{pre}\tau_{post}\frac{\mathrm{d}}{\mathrm{d}t}a_{post} \\
&\quad -a_{post}
\end{aligned}
\tag{2.1}
$$

When a presynaptic spike occurs, the presynaptic trace is updated and the weight is modified according to the rule:

$$
\begin{aligned}
& & to \\
& & a_{pre} \rightarrow \\
a_{pre} &+ A_{pre} \\
& & w \rightarrow \\
w &+ a_{post}
\end{aligned}
\tag{2.2}
$$

$a_{pre} + A_{pre} w$
$w + a_{post}$

When a postsynaptic spike occurs:

$$to$$

$$a_{post} \rightarrow a_{post} + A_{post}$$

$$w \rightarrow w + a_{pre} \quad (2.1)$$

$a_{post} + A_{post} w$
$w + a_{pre}$

To see that this formulation is equivalent, you just have to check that the equations sum linearly, and consider two cases: what happens if the presynaptic spike occurs before the postsynaptic spike, and vice versa. Try drawing a picture of it.

Now that we have a formulation that relies only on differential equations and spike events, we can turn that into Brian code.

```
start_scope()

taupre = taupost = 20*ms
wmax = 0.01
Apre = 0.01
Apost = -Apre*taupre/taupost*1.05

G = NeuronGroup(1, 'v:1')

S = Synapses(G, G,
             '''
             w : 1
             dapre/dt = -apre/taupre : 1 (event-driven)
             dapost/dt = -apost/taupost : 1 (event-driven)
             ''',
             pre='''
             v_post += w
             apre += Apre
             w = clip(w+apost, 0, wmax)
             ''',
             post='''
             apost += Apost
             w = clip(w+apre, 0, wmax)
             ''')
```

There are a few things to see there. Firstly, when defining the synapses we've given a more complicated multi-line string defining three synaptic variables (`w`, `apre` and `apost`). We've also got a new bit of syntax there,

`(event-driven)` after the definitions of `apre` and `apost`. What this means is that although these two variables evolve continuously over time, Brian should only update them at the time of an event (a spike). This is because we don't need the values of `apre` and `apost` except at spike times, and it is more efficient to only update them when needed.

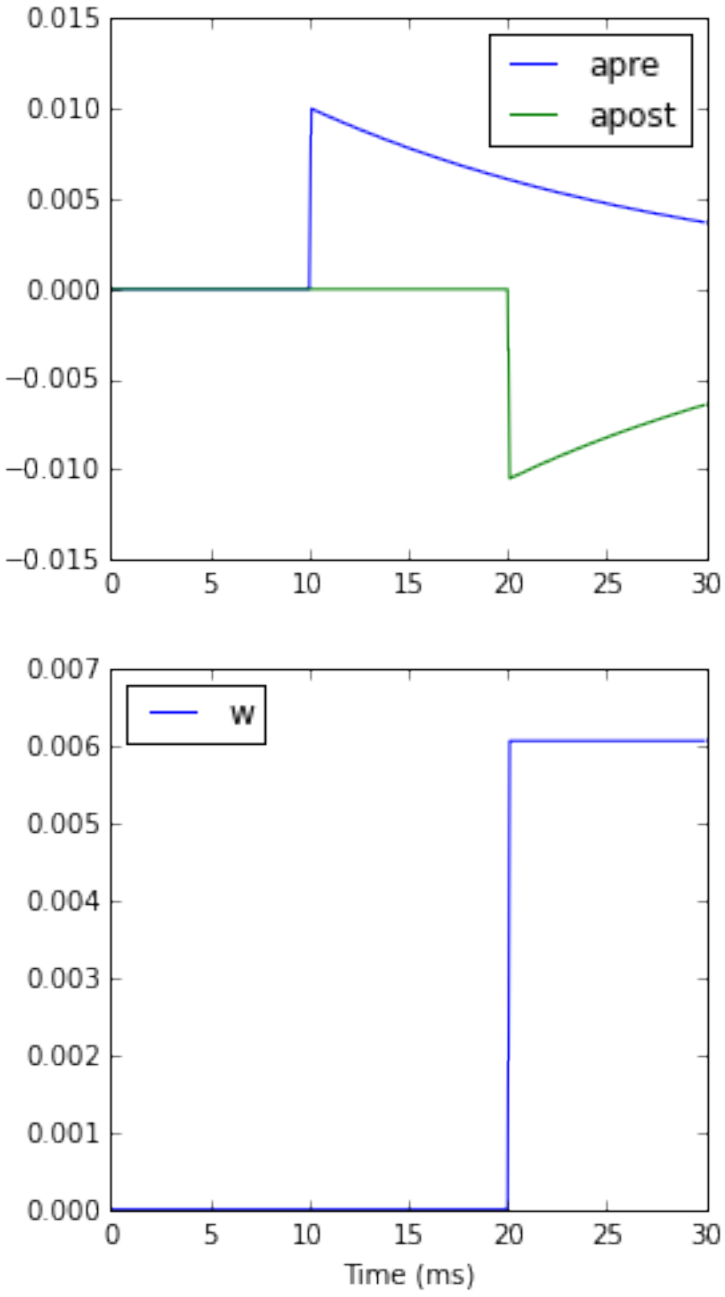
Next we have a `pre=...` argument. The first line is `v_post += w`: this is the line that actually applies the synaptic weight to the target neuron. The second line is `apre += Apre` which encodes the rule above. In the third line, we're also encoding the rule above but we've added one extra feature: we've clamped the synaptic weights between a minimum of 0 and a maximum of `wmax` so that the weights can't get too large or negative. The function `clip(x, low, high)` does this.

Finally, we have a `post=...` argument. This gives the statements to calculate when a post-synaptic neuron fires. Note that we do not modify `v` in this case, only the synaptic variables.

Now let's see how all the variables behave when a presynaptic spike arrives some time before a postsynaptic spike.

```
start_scope()

taupre = taupost = 20*ms
wmax = 0.01
Apre = 0.01
Apost = -Apre*taupre/taupost*1.05

G = NeuronGroup(2, 'v:1', threshold='t>(1+i)*10*ms', refractory=100*ms)

S = Synapses(G, G,
             '''
             w : 1
             dapre/dt = -apre/taupre : 1
             dapost/dt = -apost/taupost : 1
             ''',
             pre='''
             v_post += w
             apre += Apre
             w = clip(w+apost, 0, wmax)
             ''',
             post='''
             apost += Apost
             w = clip(w+apre, 0, wmax)
             ''')
S.connect(0, 1)
M = StateMonitor(S, ['w', 'apre', 'apost'], record=True)

run(30*ms)

figure(figsize=(4, 8))
subplot(211)
plot(M.t/ms, M.apre[0], label='apre')
plot(M.t/ms, M.apost[0], label='apost')
legend(loc='best')
subplot(212)
plot(M.t/ms, M.w[0], label='w')
legend(loc='best')
xlabel('Time (ms)')
```

```
<matplotlib.text.Text at 0x7f51af39f6d0>
```



A couple of things to note here. First of all, we've used a trick to make neuron 0 fire a spike at time 10 ms, and neuron 1 at time 20 ms. Can you see how that works?

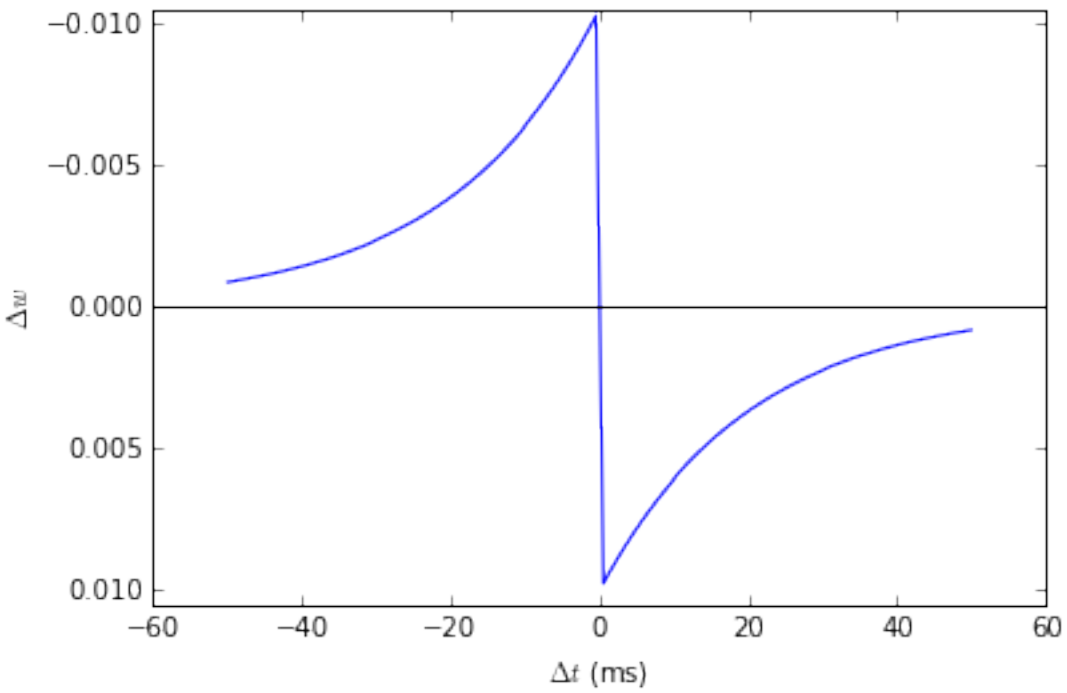
Secondly, we've removed the `(event-driven)` from the equations so you can see how `apre` and `apost` evolve over time. Try putting them back in and see what happens.

Try changing the times of the spikes to see what happens.

Finally, let's verify that this formulation is equivalent to the original one.

```
start_scope()

taupre = taupost = 20*ms
Apre = 0.01
Apost = -Apre*taupre/taupost*1.05
tmax = 50*ms
N = 100

# Presynaptic neurons G spike at times from 0 to tmax
# Postsynaptic neurons G spike at times from tmax to 0
# So difference in spike times will vary from -tmax to +tmax
G = NeuronGroup(N, 'tspike:second', threshold='t>tspike', refractory=100*ms)
H = NeuronGroup(N, 'tspike:second', threshold='t>tspike', refractory=100*ms)
G.tspike = 'i*tmax/(N-1)'
H.tspike = '(N-1-i)*tmax/(N-1)'

S = Synapses(G, H,
             '''
             w : 1
             dapre/dt = -apre/taupre : 1 (event-driven)
             dapost/dt = -apost/taupost : 1 (event-driven)
             ''',
             pre='''
             apre += Apre
             w = w+apost
             ''',
             post='''
             apost += Apost
             w = w+apre
             ''')
S.connect('i==j')

run(tmax+1*ms)

plot((H.tspike-G.tspike)/ms, S.w)
xlabel(r'$\Delta t$ (ms)')
ylabel(r'$\Delta w$')
ylim(-Apost, Apost)
axhline(0, ls='-', c='k')
```

```
<matplotlib.lines.Line2D at 0x7f51aff1d0d0>
```



Can you see how this works?

2.2.6 End of tutorial

2.3 Notebook files

- `Introduction to Brian part 1: Neurons`
- `Introduction to Brian part 2: Synapses`

CHAPTER 3

User's guide

3.1 Importing Brian

After installation, Brian is avaiable in the `brian2` package. By doing a wildcard import from this package, i.e.:

```
from brian2 import *
```

you will not only get access to the `brian2` classes and functions, but also to everything in the `pylab` package, which includes the plotting functions from matplotlib and everything included in numpy/scipy (e.g. functions such as `arange`, `linspace`, etc.). This is the style used in the documentation and in the examples but not in the Brian code itself (see Coding conventions).

If you want to use a wildcard import from Brian, but don't want to import all the additional symbols provided by `pylab`, you can use:

```
from brian2.only import *
```

Note that whenever you use something different from the most general `from brian2 import *` statement, you should be aware that Brian overwrites some numpy functions with their unit-aware equivalents (see Units). If you combine multiple wildcard imports, the Brian import should therefore be the last import. Similarly, you should not import and call overwritten numpy functions directly, e.g. by using `import numpy as np` followed by `np.sin` since this will not use the unit-aware versions. To make this easier, Brian provides a `brian2.numpy_` package that provides access to everything in numpy but overwrites certain functions. If you prefer to use prefixed names, the recommended way of doing the imports is therefore:

```
import brian2.numpy_ as np
import brian2.only as br2
```

Note that it is safe to use e.g. `np.sin` and `numpy.sin` after a `from brian2 import *`.

3.1.1 Dependency checks

Brian will check the dependency versions during import and raise an error for an outdated dependency. An outdated dependency does not necessarily mean that Brian cannot be run with it, it only means that Brian is untested on that version. If you want to force Brian to run despite the outdated dependency, set the *core.outdated_dependency_error* preference to `False`. Note that this cannot be done in a script, since you do not have access to the preferences before importing `brian2`. See Preferences for instructions how to set preferences in a file.

3.2 Physical units

Brian includes a system for defining physical units. These are defined by their standard SI unit names: amp, kilogram, second, metre/meter and the derived units coulomb, farad, gram/gramme, hertz, joule, pascal, ohm, siemens, volt, watt, together with prefixed versions (e.g. `msiemens = 0.001*siemens`) using the prefixes p, n, u, m, k, M, G, T (two exceptions: kilogram is not imported with any prefixes, metre and meter are additionaly defined with the "centi" prefix, i.e. cmetre/cmeter). In addition a couple of useful standard abbreviations like "cm" (instead of cmetre/cmeter), "nS" (instead of nsiemens), "ms" (instead of msecond), "Hz" (instead of hertz), etc. are included.

3.2.1 Using units

You can generate a physical quantity by multiplying a scalar or vector value with its physical unit:

```
>>> tau = 20*ms
>>> print tau
20. ms
>>> rates = [10, 20, 30] * Hz
>>> print rates
[ 10.  20.  30.] Hz
```

Brian will check the consistency of operations on units and raise an error for dimensionality mismatches:

```
>>> tau += 1  # ms? second?
Traceback (most recent call last):
...
DimensionMismatchError: Addition, dimensions were (s) (1)
>>> 3*gram + 3*amp
Traceback (most recent call last):
...
DimensionMismatchError: Addition, dimensions were (kg) (A)
```

Most Brian functions will also complain about non-specified or incorrect units:

```
>>> G = NeuronGroup(10, 'dv/dt = -v/tau: volt', dt=0.5)
Traceback (most recent call last):
...
DimensionMismatchError: Function "__init__" variable "dt" has wrong dimensions, dimensions were (1)
```

Numpy functions have been overwritten to correctly work with units (see the developer documentation for more details):

```
>>> print mean(rates)
20. Hz
>>> print rates.repeat(2)
[ 10.  10.  20.  20.  30.  30.] Hz
```

3.2.2 Removing units

There are various options to remove the units from a value (e.g. to use it with analysis functions that do not correctly work with units)

- Divide the value by its unit (most of the time the recommended option because it is clear about the scale)
- Transform it to a pure numpy array in the base unit by calling `asarray()` (no copy) or `array` (copy)
- Directly get the unitless value of a state variable by appending an underscore to the name

```
>>> tau/ms
20.0
>> asarray(rates)
array([ 10.,  20.,  30.])
>>> G = NeuronGroup(5, 'dv/dt = -v/tau: volt')
>>> print G.v_[:]
[ 0.,  0.,  0.,  0.,  0.]
```

3.2.3 Importing units

Brian generates standard names for units, combining the unit name (e.g. "siemens") with a prefixes (e.g. "m"), and also generates squared and cubed versions by appending a number. For example, the units "msiemens", "siemens2", "usiemens3" are all predefined. You can import these units from the package `brian2.units.allunits` – accordingly, an `from brian2.units.allunits import *` will result in everything from `Ylumen3` (cubed yotta lumen) to `ymol` (yocto mole) being imported.

A better choice is normally to do an `from brian2.units import *` or import everything `from brian2 import *`, this imports only the base units amp, kilogram, second, metre/meter and the derived units coulomb, farad, gram/gramme, hertz, joule, pascal, ohm, siemens, volt, watt, together with the prefixes p, n, u, m, k, M, G, T (two exceptions: kilogram is not imported with any prefixes, metre and meter are additionaly defined with the "centi" prefix, i.e. cmetre/cmeter).

In addition a couple of useful standard abbreviations like "cm" (instead of cmetre/cmeter), "nS" (instead of nsiemens), "ms" (instead of msecond), "Hz" (instead of hertz), etc. are added (they can be individually imported from `brian2.units.stdunits`).

3.2.4 In-place operations on quantities

In-place operations on quantity arrays change the underlying array, in the same way as for standard numpy arrays. This means, that any other variables referencing the same object will be affected as well:

```
>>> q = [1, 2] * mV
>>> r = q
>>> q += 1*mV
>>> q
array([ 2.,  3.]) * mvolt
>>> r
array([ 2.,  3.]) * mvolt
```

In contrast, scalar quantities will never change the underlying value but instead return a new value (in the same way as standard Python scalars):

```
>>> x = 1*mV
>>> y = x
>>> x *= 2
>>> x
2. * mvolt
>>> y
1. * mvolt
```

3.2.5 Comparison with Brian 1

Brian 1 did only support scalar quantities, units were not stored for arrays. Some expressions therefore have different values in Brian 1 and Brian 2:

Expression	Brian 1	Brian 2
1 * mV	1.0 * mvolt	1.0 * mvolt
np.array(1) * mV	0.001	1.0 * mvolt
np.array([1]) * mV	array([0.001])	array([1.]) * mvolt
np.mean(np.arange(5) * mV)	0.002	2.0 * mvolt
np.arange(2) * mV	array([0. , 0.001])	array([0., 1.]) * mvolt
(np.arange(2) * mV) >= 1 * mV	array([False, True], dtype=bool)	array([False, True], dtype=bool)
(np.arange(2) * mV)[0] >= 1 * mV	False	False
(np.arange(2) * mV)[1] >= 1 * mV	DimensionMismatchError	True

3.3 Models and neuron groups

The core of every simulation is a *NeuronGroup*, a group of neurons that share the same equations defining their properties. The minimum *NeuronGroup* specification contains the number of neurons and the model description in the form of equations:

```
G = NeuronGroup(10, 'dv/dt = -v/(10*ms) : volt')
```

This defines a group of 10 leaky integrators. The model description can be directly given as a (possibly multi-line) string as above, or as an *Equations* object. For more details on the form of equations, see Equations. Note that model descriptions can make reference to physical units, but also to scalar variables declared outside of the model description itself:

```
tau = 10*ms
G = NeuronGroup(10, 'dv/dt = -v/tau : volt')
```

If a variable should be taken as a *parameter* of the neurons, i.e. if it should be possible to vary its value across neurons, it has to be declared as part of the model description:

```
G = NeuronGroup(10, '''dv/dt = -v/tau : volt
                       tau : second''')
```

To make complex model descriptions more readable, named subexpressions can be used:

```
G = NeuronGroup(10, '''dv/dt = I_leak / Cm : volt
                       I_leak = g_L*(E_L - v) : amp''')
```

Sometimes it can also be useful to introduce shared variables or subexpressions, i.e. variables that have a common value for all neurons. In contrast to external variables (such as `Cm` above), such variables can change during a run, e.g. by using *run_regularly()*. This can be for example used for an external stimulus that changes in the course of a run:

```
G = NeuronGroup(10, '''shared_input : volt (shared)
                       dv/dt = (-v + shared_input)/tau : volt
                       tau : second''')
```

Note that there are several restrictions around the use of shared variables: they cannot be written to in contexts where statements apply only to a subset of neurons (e.g. reset statements, see below). If a code block mixes statements writing to shared and vector variables, then the shared statements have to come first.

3.3.1 Threshold and reset

To emit spikes, neurons need a *threshold*. Threshold and reset are given as strings in the *NeuronGroup* constructor:

```
tau = 10*ms
G = NeuronGroup(10, 'dv/dt = -v/tau : volt', threshold='v > -50*mV',
                reset='v = -70*mV')
```

Whenever the threshold condition is fulfilled, the reset statements will be executed. Again, both threshold and reset can refer to physical units, external variables and parameters, in the same way as model descriptions:

```
v_r = -70*mV  # reset potential
G = NeuronGroup(10, '''dv/dt = -v/tau : volt
                       v_th : volt  # neuron-specific threshold''',
                threshold='v > v_th', reset='v = v_r')
```

3.3.2 Refractoriness

To make a neuron non-excitable for a certain time period after a spike, the refractory keyword can be used:

```
G = NeuronGroup(10, 'dv/dt = -v/tau : volt', threshold='v > -50*mV',
                reset='v = -70*mV', refractory=5*ms)
```

This will not allow any threshold crossing for a neuron for 5ms after a spike. The refractory keyword allows for more flexible refractoriness specifications, see Refractoriness for details.

3.3.3 State variables

Differential equations and parameters in model descriptions are stored as *state variables* of the *NeuronGroup*. They can be accessed and set as an attribute of the group. To get the values without physical units (e.g. for analysing data with external tools), use an underscore after the name:

```
>>> G = NeuronGroup(10, '''dv/dt = (-v + shared_input)/tau : volt
                           shared_input : volt (shared)
...                        tau : second''')
>>> G.v = -70*mV
>>> print G.v
<neurongroup.v: array([-70., -70., -70., -70., -70., -70., -70., -70., -70., -70.]) * mvolt>
>>> print G.v_  # values without units
<neurongroup.v_: array([-0.07, -0.07, -0.07, -0.07, -0.07, -0.07, -0.07, -0.07, -0.07, -0.07])>
>>> G.shared_input = 5*mV
>>> print G.shared_input
<neurongroup.shared_input: 5.0 * mvolt>
```

The value of state variables can also be set using string expressions that can refer to units and external variables, other state variables, mathematical functions, and a special variable `i`, the index of the neuron:

```
>>> G.tau = '5*ms + 5*ms*rand() + i*5*ms'
>>> print G.tau
<neurongroup.tau: array([  5.03593449,  10.74914808,  19.01641896,  21.66813281,
        27.16243388,  31.13571924,  36.28173038,  40.04921519,
        47.28797921,  50.18913711]) * msecond>
```

For shared variables, such string expressions can only refer to shared values:

```
>>> G.shared_input = 'rand()*mV + 4*mV'
>>> print G.shared_input
<neurongroup.shared_input: 4.2579690100000001 * mvolt>
```

Sometimes it can be convenient to access multiple state variables at once, e.g. to set initial values from a dictionary of values or to store all the values of a group on disk. This can be done with the *Group.get_states()* and *Group.set_states()* methods:

```
>>> group = NeuronGroup(5, '''dv/dt = -v/tau : 1
...                           tau : second''')
>>> initial_values = {'v': [0, 1, 2, 3, 4],
...                   'tau': [10, 20, 10, 20, 10]*ms}
>>> group.set_states(initial_values)
>>> group.v[:]
array([ 0.,  1.,  2.,  3.,  4.])
>>> group.tau[:]
array([ 10.,  20.,  10.,  20.,  10.]) * msecond
>>> states = group.get_states()
>>> states['v']
array([ 0.,  1.,  2.,  3.,  4.])
>>> sorted(states.keys())
['N', 'dt', 'i', 't', 'tau', 'v']
```

3.3.4 Subgroups

It is often useful to refer to a subset of neurons, this can be achieved using slicing syntax:

```
G = NeuronGroup(10, '''dv/dt = -v/tau : volt
                       tau : second''',
                threshold='v > -50*mV',
                reset='v = -70*mV')
# Create subgroups
G1 = G[:5]
G2 = G[5:]

# This will set the values in the main group, subgroups are just "views"
G1.tau = 10*ms
G2.tau = 20*ms
```

Subgroups can be used in most places where regular groups are used, e.g. their state variables or spiking activity can be recorded using monitors, they can be connected via *Synapses*, etc. In such situations, indices (e.g. the indices of the neurons to record from in a *StateMonitor*) are relative to the subgroup, not to the main group

3.3.5 Linked variables

A *NeuronGroup* can define parameters that are not stored in this group, but are instead a reference to a state variable in another group. For this, a group defines a parameter as `linked` and then uses *linked_var()* to specify the linking. This can for example be useful to model shared noise between cells:

```
inp = NeuronGroup(1, 'dnoise/dt = -noise/tau + tau**-0.5*xi : 1')

neurons = NeuronGroup(100, '''noise : 1 (linked)
                              dv/dt = (-v + noise_strength*noise)/tau : volt''')
neurons.noise = linked_var(inp, 'noise')
```

If the two groups have the same size, the linking will be done in a 1-to-1 fashion. If the source group has the size one (as in the above example) or if the source parameter is a shared variable, then the linking will be done as 1-to-all. In all other cases, you have to specify the indices to use for the linking explicitly:

```
# two inputs with different phases
inp = NeuronGroup(2, '''phase : 1
                        dx/dt = 1*mV/ms*sin(2*pi*100*Hz*t-phase) : volt''')
inp.phase = [0, pi/2]

neurons = NeuronGroup(100, '''inp : volt (linked)
                              dv/dt = (-v + inp) / tau : volt''')
# Half of the cells get the first input, other half gets the second
neurons.inp = linked_var(inp, 'x', index=repeat([0, 1], 50))
```

3.3.6 Numerical integration

Differential equations are converted into a sequence of statements that integrate the equations numerically over a single time step. By default, Brian chooses an integration method automatically, trying to solve the equations exactly first (for linear equations) and then resorting to numerical algorithms. It will also take care of integrating stochastic differential equations appropriately. Each class defines its own list of algorithms it tries to apply, *NeuronGroup* and *Synapses* will use the first suitable method out of the methods `'linear'`, `'euler'`, and `'milstein'` while *SpatialNeuron* objects will use `'linear'`, `'exponential_euler'`, `'rk2'`, or `'milstein'`.

If you prefer to chose an integration algorithm yourself, you can do so using the `method` keyword for *NeuronGroup*, *Synapses*, or *SpatialNeuron*. The complete list of available methods is the following:

- `'linear'`: exact integration for linear equations
- `'independent'`: exact integration for a system of independent equations, where all the equations can be analytically solved independently
- `'exponential_euler'`: exponential Euler integration for conditionally linear equations
- `'euler'`: forward Euler integration (for additive stochastic differential equations using the Euler-Maruyama method)
- `'rk2'`: second order Runge-Kutta method (midpoint method)
- `'rk4'`: classical Runge-Kutta method (RK4)
- `'milstein'`: derivative-free Milstein method for solving stochastic differential equations with diagonal multiplicative noise

You can also define your own numerical integrators, see State update for details.

3.4 Equations

3.4.1 Equation strings

Equations are used both in *NeuronGroup* and *Synapses* to:

- define state variables
- define continuous-updates on these variables, through differential equations

Equations are defined by multiline strings.

An Equation is a set of single lines in a string:

1. `dx/dt = f :  unit` (differential equation)
2. `x = f :  unit` (subexpression)

3. `x : unit` (parameter)

The equations may be defined on multiple lines (no explicit line continuation with `\` is necessary). Comments using `#` may also be included. Subunits are not allowed, i.e., one must write `volt`, not `mV`. This is to make it clear that the values are internally always saved in the basic units, so no confusion can arise when getting the values out of a *NeuronGroup* and discarding the units. Compound units are of course allowed as well (e.g. `farad/meter**2`). There are also three special "units" that can be used: `1` denotes a dimensionless floating point variable, `boolean` and `integer` denote dimensionless variables of the respective kind.

Some special variables are defined: `t`, `dt` (time) and `xi` (white noise). Variable names starting with an underscore and a couple of other names that have special meanings under certain circumstances (e.g. names ending in `_pre` or `_post`) are forbidden.

For stochastic equations with several `xi` values it is now necessary to make clear whether they correspond to the same or different noise instantiations. To make this distinction, an arbitrary suffix can be used, e.g. using `xi_1` several times refers to the same variable, `xi_2` (or `xi_inh`, `xi_alpha`, etc.) refers to another. An error will be raised if you use more than one plain `xi`. Note that noise is always independent across neurons, you can only work around this restriction by defining your noise variable as a shared parameter and update it using a user-defined function (e.g. with `run_regularly`), or create a group that models the noise and link to its variable (see *Linked variables*).

Flags

A *flag* is a keyword in parentheses at the end of the line, which qualifies the equations. There are several keywords:

event-driven this is only used in Synapses, and means that the differential equation should be updated only at the times of events. This implies that the equation is taken out of the continuous state update, and instead a event-based state update statement is generated and inserted into event codes (pre and post). This can only qualify differential equations of synapses. Currently, only one-dimensional linear equations can be handled (see below).

unless refractory this means the variable is not updated during the refractory period. This can only qualify differential equations of neuron groups.

constant this means the parameter will not be changed during a run. This allows optimizations in state updaters. This can only qualify parameters.

shared this means that a parameter or subexpression is not neuron-/synapse-specific but rather a single value for the whole *NeuronGroup* or *Synapses*. A shared subexpression can only refer to other shared variables.

linked this means that a parameter refers to a parameter in another *NeuronGroup*. See *Linked variables* for more details.

Multiple flags may be specified as follows:

```
dx/dt = f : unit (flag1,flag2)
```

Event-driven equations

Equations defined as event-driven are completely ignored in the state update. They are only defined as variables that can be externally accessed. There are additional constraints:

- An event-driven variable cannot be used by any other equation that is not also event-driven.
- An event-driven equation cannot depend on a differential equation that is not event-driven (directly, or indirectly through subexpressions). It can depend on a constant parameter.

Currently, automatic event-driven updates are only possible for one-dimensional linear equations, but this may be extended in the future.

Equation objects

The model definitions for *NeuronGroup* and *Synapses* can be simple strings or *Equations* objects. Such objects can be combined using the add operator:

```
eqs = Equations('dx/dt = (y-x)/tau : volt')
eqs += Equations('dy/dt = -y/tau: volt')
```

Equations allow for the specification of values in the strings, but do this by simple string replacement, e.g. you can do:

```
eqs = Equations('dx/dt = x/tau : volt', tau=10*ms)
```

but this is exactly equivalent to:

```
eqs = Equations('dx/dt = x/(10*ms) : volt')
```

The *Equations* object does some basic syntax checking and will raise an error if two equations defining the same variable are combined. It does not however do unit checking, checking for unknown identifiers or incorrect flags – all this will be done during the instantiation of a *NeuronGroup* or *Synapses* object.

3.4.2 External variables and functions

Equations defining neuronal or synaptic equations can contain references to external parameters or functions. These references are looked up at the time that the simulation is run. If you don't specify where to look them up, it will look in the Python local/global namespace (i.e. the block of code where you call *run()*). If you want to override this, you can specify an explicit "namespace". This is a Python dictionary with keys being variable names as they appear in the equations, and values being the desired value of that variable. This namespace can be specified either in the creation of the group or when you can the *run()* function using the `namespace` keyword argument.

The following three examples show the different ways of providing external variable values, all having the same effect in this case:

```
# Explicit argument to the NeuronGroup
G = NeuronGroup(1, 'dv/dt = -v / tau : 1', namespace={'tau': 10*ms})
net = Network(G)
net.run(10*ms)

# Explicit argument to the run function
G = NeuronGroup(1, 'dv/dt = -v / tau : 1')
net = Network(G)
net.run(10*ms, namespace={'tau': 10*ms})

# Implicit namespace from the context
G = NeuronGroup(1, 'dv/dt = -v / tau : 1')
net = Network(G)
tau = 10*ms
net.run(10*ms)
```

See Namespaces for more details.

3.4.3 Examples

Equation objects

Concatenating equations

```
>>> membrane_eqs = Equations('dv/dt = -(v + I)/ tau : volt')
>>> eqs1 = membrane_eqs + Equations('''I = sin(2*pi*freq*t) : volt
...                                    freq : Hz''')
>>> eqs2 = membrane_eqs + Equations('''I : volt''')
>>> print eqs1
I = sin(2*pi*freq*t)  : V
dv/dt = -(v + I)/ tau  : V
freq : Hz
>>> print eqs2
dv/dt = -(v + I)/ tau  : V
I : V
```

Substituting variable names

```
>>> general_equation = 'dg/dt = -g / tau : siemens'
>>> eqs_exc = Equations(general_equation, g='g_e', tau='tau_e')
>>> eqs_inh = Equations(general_equation, g='g_i', tau='tau_i')
>>> print eqs_exc
dg_e/dt = -g_e / tau_e  : S
>>> print eqs_inh
dg_i/dt = -g_i / tau_i  : S
```

Inserting values

```
>>> eqs = Equations('dv/dt = mu/tau + sigma/tau**.5*xi : volt',
                    mu = -65*mV, sigma=3*mV, tau=10*ms)
>>> print eqs
dv/dt = (-0.065 * volt)/(10.0 * msecond) + (3.0 * mvolt)/(10.0 * msecond)**.5*xi  : V
```

3.5 Refractoriness

Brian allows you to model the absolute refractory period of a neuron in a flexible way. The definition of refractoriness consists of two components: the amount of time after a spike that a neuron is considered to be refractory, and what changes in the neuron during the refractoriness.

3.5.1 Defining the refractory period

The refractory period is specified by the `refractory` keyword in the *NeuronGroup* initializer. In the simplest case, this is simply a fixed time, valid for all neurons:

```
G = NeuronGroup(N, model='...', threshold='...', reset='...',
                refractory=2*ms)
```

Alternatively, it can be a string expression that evaluates to a time. This expression will be evaluated after every spike and allows for a changing refractory period. For example, the following will set the refractory period to a random duration between 1ms and 3ms after every spike:

```
G = NeuronGroup(N, model='...', threshold='...', reset='...',
                refractory='(1 + 2*rand())*ms')
```

In general, modelling a refractory period that varies across neurons involves declaring a state variable that stores the refractory period per neuron as a model parameter. The refractory expression can then refer to this parameter:

```
G = NeuronGroup(N, model='''...
                            refractory : second''', threshold='...',
```

```
                reset='...', refractory='refractory')
# Set the refractory period for each cell
G.refractory = ...
```

This state variable can also be a dynamic variable itself. For example, it can serve as an adaptation mechanism by increasing it after every spike and letting it relax back to a steady-state value between spikes:

```
refractory_0 = 2*ms
tau_refractory = 50*ms
G = NeuronGroup(N, model='''...
                           drefractory/dt = (refractory_0 - refractory) / tau_refractory : second''
                threshold='...', refractory='refractory',
                reset='''...
                         refractory += 1*ms''')
G.refractory = refractory_0
```

In some cases, the condition for leaving the refractory period is not easily expressed as a certain time span. For example, in a Hodgkin-Huxley type model the threshold is only used for *counting* spikes and the refractoriness is used to prevent to count multiple spikes for a single threshold crossing (the threshold condition would evaluate to `True` for several time points). When a neuron should leave the refractory period is not easily expressed as a time span but more naturally as a condition that the neuron should remain refractory for as long as it stays above the threshold. This can be achieved by using a string expression for the `refractory` keyword that evaluates to a boolean condition:

```
G = NeuronGroup(N, model='...', threshold='v > -20*mV',
                refractory='v >= -20*mV')
```

The `refractory` keyword should be read as "stay refractory as long as the condition remains true". In fact, specifying a time span for the refractoriness will be automatically transformed into a logical expression using the current time `t` and the time of the last spike `lastspike`. Specifying `refractory=2*ms` is equivalent to specifying `refractory='(t - lastspike) <= 2*ms'`.

3.5.2 Defining model behaviour during refractoriness

The refractoriness definition as described above only has a single effect by itself: threshold crossings during the refractory period are ignored. In the following model, the variable `v` continues to update during the refractory period but it does not elicit a spike if it crosses the threshold:

```
G = NeuronGroup(N, 'dv/dt = -v / tau : 1',
                threshold='v > 1', reset='v=0',
                refractory=2*ms)
```

There is also a second implementation of refractoriness that is supported by Brian, one or several state variables can be clamped during the refractory period. To model this kind of behaviour, variables that should stop being updated during refractoriness can be marked with the `(unless refractory)` flag:

```
G = NeuronGroup(N, '''dv/dt = -(v + w)/ tau_v : 1 (unless refractory)
                      dw/dt = -w / tau_w : 1''',
                threshold='v > 1', reset='v=0; w+=0.1', refractory=2*ms)
```

In the above model, the `v` variable is clamped at 0 for 2ms after a spike but the adaptation variable `w` continues to update during this time.

In fact, arbitrary behaviours can be defined using Brian's refractoriness mechanism. For more details, see the documentation on details of the refractoriness implementation.

3.6 Synapses

Note: *Synapses* is now the only class for defining synaptic interactions, it replaces *Connections*, *STDP*, etc.

3.6.1 Defining synaptic models

The most simple synapse (adding a fixed amount to the target membrane potential on every spike) is described as follows:

```
w = 1*mV
S = Synapses(P, Q, pre='v += w')
```

This defines a set of synapses between *NeuronGroup* P and *NeuronGroup* Q. If the target group is not specified, it is identical to the source group by default. The `pre` keyword defines what happens when a presynaptic spike arrives at a synapse. In this case, the constant `w` is added to variable `v`. Because `v` is not defined as a synaptic variable, it is assumed by default that it is a postsynaptic variable, defined in the target *NeuronGroup* Q. Note that this does not does create synapses (see *Creating Synapses*), only the synaptic models.

To define more complex models, models can be described as string equations, similar to the models specified in *NeuronGroup*:

```
S = Synapses(P, Q, model='w : volt', pre='v += w')
```

The above specifies a parameter `w`, i.e. a synapse-specific weight.

Synapses can also specify code that should be executed whenever a postsynaptic spike occurs (keyword `post`) and a fixed (pre-synaptic) delay for all synapses (keyword `delay`). See the reference documentation for *Synapses* for more details.

Model syntax

The model follows exactly the same syntax as for *NeuronGroup*. There can be parameters (e.g. synaptic variable `w` above), but there can also be named subexpressions and differential equations, describing the dynamics of synaptic variables. In all cases, synaptic variables are created, one value per synapse. Internally, these are stored as arrays. There are a few things worth noting:

- A variable with the `_post` suffix is looked up in the postsynaptic (target) neuron. That is, `v_post` means variable `v` in the postsynaptic neuron.
- A variable with the `_pre` suffix is looked up in the presynaptic (source) neuron.
- A variable not defined as a synaptic variable is considered to be postsynaptic.
- A variable not defined as a synaptic variable and not defined in the postsynaptic neuron is considered an external constant

For the integration of differential equations, one can use the same keywords as for *NeuronGroup*.

Event-driven updates

By default, differential equations are integrated in a clock-driven fashion, as for a *NeuronGroup*. This is potentially very time consuming, because all synapses are updated at every timestep. It is possible to ask Brian 2 to simulate differential equations in an event-driven fashion using the keyword `(event-driven)`. A typical example is pre- and postsynaptic traces in STDP:

```
model='''w:1
         dApre/dt=-Apre/taupre : 1 (event-driven)
         dApost/dt=-Apost/taupost : 1 (event-driven)'''
```

Here, Brian updates the value of `Apre` for a given synapse only when this synapse receives a spike, whether it is presynaptic or postsynaptic. More precisely, the variables are updated every time either the `pre` or `post` code is called for the synapse, so that the values are always up to date when these codes are executed.

Automatic event-driven updates are only possible for a subset of equations, in particular for one-dimensional linear equations. These equations must also be independent of the other ones, that is, a differential equation that is not event-driven cannot depend on an event-driven equation (since the values are not continuously updated). In other cases, the user can write event-driven code explicitly in the update codes (see below).

Pre and post codes

The `pre` code is executed at each synapse receiving a presynaptic spike. For example:

```
pre='v+=w'
```

adds the value of synaptic variable `w` to postsynaptic variable `v`. As for the model equations, the `_post` (`_pre`) suffix indicates a postsynaptic (presynaptic) variable, and variables not found in the synaptic variables are considered postsynaptic by default. Internally, the code is executed for all synapses receiving presynaptic spikes during the current timestep. Therefore, the code should be understood as acting on arrays rather than single values. Any sort of code can be executed. For example, the following code defines stochastic synapses, with a synaptic weight `w` and transmission probability `p`:

```
S=Synapses(input,neurons,model="""w : 1
                                 p : 1""",
                          pre="v+=w*(rand()<p)")
```

The code means that `w` is added to `v` with probability `p` (note that, internally, `rand()` is transformed to a instruction that outputs an array of random numbers). The code may also include multiple lines.

As mentioned above, it is possible to write event-driven update code for the synaptic variables. For this, two special variables are provided: `t` is the current time when the code is executed, and `lastupdate` is the last time when the synapse was updated (either through `pre` or `post` code). An example is short-term plasticity (in fact this could be done automatically with the use of the `(event-driven)` keyword mentioned above):

```
S=Synapses(input,neuron,
           model='''x : 1
                    u : 1
                    w : 1''',
           pre='''u=U+(u-U)*exp(-(t-lastupdate)/tauf)
                  x=1+(x-1)*exp(-(t-lastupdate)/taud)
                  i+=w*u*x
                  x*=(1-u)
                  u+=U*(1-u)''')
```

By default, the `pre` pathway is executed before the `post` pathway (both are executed in the `'synapses'` scheduling slot, but the `pre` pathway has the `order` attribute -1, wheras the `post` pathway has `order` 1. See *Scheduling* for more details).

Summed variables

In many cases, the postsynaptic neuron has a variable that represents a sum of variables over all its synapses. This is called a "summed variable". An example is nonlinear synapses (e.g. NMDA):

```
neurons = NeuronGroup(1, model="""dv/dt=(gtot-v)/(10*ms) : 1
                                  gtot : 1""")
S=Synapses(input,neurons,
           model='''dg/dt=-a*g+b*x*(1-g) : 1
                    gtot_post = g : 1  (summed)
                    dx/dt=-c*x : 1
                    w : 1 # synaptic weight
                 ''',
           pre='x+=w')
```

Here, each synapse has a conductance `g` with nonlinear dynamics. The neuron's total conductance is `gtot`. The line stating `gtot_post = g : 1 (summed)` specifies the link between the two: `gtot` in the postsynaptic group is the summer over all variables `g` of the corresponding synapses. What happens during the simulation is that at each time step, presynaptic conductances are summed for each neuron and the result is copied to the variable `gtot`. Another example is gap junctions:

```
neurons = NeuronGroup(N, model='''dv/dt=(v0-v+Igap)/tau : 1
                                  Igap : 1''')
S=Synapses(neurons,model='''w:1 # gap junction conductance
                            Igap_post = w*(v_pre-v_post): 1 (summed)''')
```

Here, `Igap` is the total gap junction current received by the postsynaptic neuron.

3.6.2 Creating synapses

Creating a *Synapses* instance does not create synapses, it only specifies their dynamics. The following command creates a synapse between neuron `i` in the source group and neuron `j` in the target group:

```
S.connect(i, j)
```

It is possible to create several synapses for a given pair of neurons:

```
S.connect(i, j, n=3)
```

This is useful for example if one wants to have multiple synapses with different delays. Multiple synaptic connections can be created in a single statement:

```
S.connect(True)
S.connect([1, 2], [1, 2])
S.connect(numpy.arange(10), 1)
```

The first statement connects all neuron pairs. The second statement creates synapses between neurons 1 and 1, and between neurons 2 and 2. The third statement creates synapses between the first ten neurons in the source group and neuron 1 in the target group.

One can also create synapses using code:

```
S.connect('i==j')
S.connect('j==((i+1)%N)')
```

The code is a boolean statement that should return True when a synapse must be created, where `i` is the presynaptic neuron index and `j` is the postsynaptic neuron index (special variables). Here the first statement creates one-to-one connections, the second statement creates connections with a ring structure (`N` is the number of neurons, assumed to defined elsewhere by the user as an external variable). This way of creating synapses is generally preferred.

The string expressions can also refer to pre- or postsynaptic variables. This can be useful for example for spatial connectivity: assuming that the pre- and postsynaptic groups have parameters `x` and `y`, storing their location, the following statement connects all cells in a 250 um radius:

```
S.connect('sqrt((x_pre-x_post)**2 + (y_pre-y_post)**2) < 250*umeter')
```

Synapse creation can also be probabilistic by providing a `p` argument, providing the connection probability for each pair of synapses:

```
S.connect(True, p=0.1)
```

This connects all neuron pairs with a probability of 10%. Probabilities can also be given as expressions, for example to implement a connection probability that depends on distance:

```
S.connect('i != j',
          p='p_max*exp(-(x_pre-x_post)**2+(y_pre-y_post)**2) / (2*(125*umeter)**2)')
```

If this statement is applied to a *Synapses* object that connects a group to itself, it prevents self-connections (`i != j`) and connects cells with a probability that is modulated according to a 2-dimensional Gaussian of the distance between the cells.

If conditions for connecting neurons are combined with both the `n` (number of synapses to create) and the `p` (probability of a synapse) keywords, they are interpreted in the following way:

For every pair i, j:
    if condition(i, j) is fulfilled:
        Evaluate p(i, j)
        If p(i, j) < uniform random number between 0 and 1:
            Create n(i, j) synapses for (i, j)

3.6.3 Accessing synaptic variables

Synaptic variables can be accessed in a similar way as *NeuronGroup* variables. They can be indexed with two indexes, corresponding to the indexes of pre and postsynaptic neurons, and optionally with a third index in the case of multiple synapses. Here are a few examples:

```
S.w[2, 5] = 1*nS
S.w[1, :] = 2*nS
S.w = 1*nS # all synapses assigned
w0 = S.w[2, 3, 1] # second synapse for connection 2->3
S.w[2, 3] = (1*nS, 2*nS)
S.w[group1, group2] = "(1+cos(i-j))*2*nS"
S.w[:, :] = 'rand()*nS'
S.w['abs(x_pre-x_post) < 250*umetre'] = 1*nS
```

Note that it is also possible to index synaptic variables with a single index (integer, slice, or array), but in this case synaptic indices have to be provided.

3.6.4 Delays

There is a special synaptic variable that is automatically created: `delay`. It is the propagation delay from the presynaptic neuron to the synapse, i.e., the presynaptic delay. This is just a convenience syntax for accessing the delay stored in the presynaptic pathway: `pre.delay`. When there is a postsynaptic code (keyword `post`), the delay of the postsynaptic pathway can be accessed as `post.delay`.

The delay variable(s) can be set and accessed in the same way as other synaptic variables.

3.6.5 Multiple pathways

It is possible to have multiple pathways with different update codes from the same presynaptic neuron group. This may be interesting in cases when different operations must be applied at different times for the same presynaptic spike. To do this, specify a dictionary of pathway names and codes:

```
pre={'pre_transmission': 'ge+=w',
     'pre_plasticity': '''w=clip(w+Apost,0,inf)
                          Apre+=dApre'''}
```

This creates two pathways with the given names (in fact, specifying `pre=code` is just a shorter syntax for `pre={'pre': code}`) through which the delay variables can be accessed. The following statement, for example, sets the delay of the synapse between the first neurons of the source and target groups in the `pre_plasticity` pathway:

```
S.pre_plasticity.delay[0,0] = 3*ms
```

As mentioned above, `pre` pathways are generally executed before `post` pathways. The order of execution of several `pre` (or `post`) pathways is however arbitrary, and simply based on the alphabetical ordering of their names (i.e. `pre_plasticity` will be executed before `pre_transmission`). To explicitly specify the order, set the `order` attribute of the pathway, e.g.:

```
S.pre_transmission.order = -2
```

will make sure that the `pre_transmission` code is executed before the `pre_plasticity` code in each time step.

3.6.6 Monitoring synaptic variables

A *StateMonitor* object can be used to monitor synaptic variables. For example, the following statement creates a monitor for variable `w` for the synapses 0 and 1:

```
M = StateMonitor(S,'w',record=[0,1])
```

Note that these are *synapse* indices, not neuron indices. More convenient is to directly index the *Synapses* object, Brian will automatically calculate the indices for you in this case:

```
M = StateMonitor(S,'w',record=S[0, :])  # all synapses originating from neuron 0
M = StateMonitor(S,'w',record=S['i!=j'])  # all synapses excluding autapses
M = StateMonitor(S,'w',record=S['w>0'])  # all synapses with non-zero weights (at this time)
```

The recorded traces can then be accessed in the usual way, again with the possibility to index the *Synapses* object:

```
plot(M.t / ms, M[0].w / nS)  # first synapse
plot(M.t / ms, M[0, :].w / nS)  # all synapses originating from neuron 0
plot(M.t / ms, M['w>0'].w / nS)  # all synapses with non-zero weights (at this time)
```

3.7 Input stimuli

There are various ways of providing "external" input to a network. Brian does not yet provide all the features of Brian1 in this regard, but there is already a range of options, detailed below.

3.7.1 Poisson input

For generating spikes according to a Poisson point process, *PoissonGroup* can be used. It takes a rate or an array of rates (one rate per neuron) as an argument and can be connected to a *NeuronGroup* via the usual *Synapses* mechanism. At the moment, using `PoissonGroup(N, rates)` is equivalent to `NeuronGroup(N, 'rates : Hz', threshold='rand()<rates*dt')` and setting the group's `rates` attribute. The explicit creation of such a *NeuronGroup* might be useful if the rates for the neurons are not constant in time, since it allows using the techniques mentioned below (formulating rates as equations or referring to a timed array). In the future, the implementation of *PoissonGroup* will change to a more efficient spike generation mechanism, based on the calculation of inter-spike intervals. Note that, as can be seen in its equivalent *NeuronGroup* formulation, a *PoissonGroup* does not work for high rates where more than one spike might fall into a single timestep. Use several units with lower rates in this case (e.g. use `PoissonGroup(10, 1000*Hz)` instead of `PoissonGroup(1, 10000*Hz)`).

Example use:

```
P = PoissonGroup(100, np.arange(100)*Hz + 10*Hz)
G = NeuronGroup(100, 'dv/dt = -v / (10*ms) : 1')
S = Synapses(P, G, pre='v+=0.1', connect='i==j')
```

For simulations where the *PoissonGroup* is just used as a source of input to a neuron (i.e., the individually generated spikes are not important, just their impact on the target cell), the *PoissonInput* class provides a more efficient alternative. Instead of generating spikes, it directly updates a target variable based on the sum of independent Poisson processes:

```
G = NeuronGroup(100, 'dv/dt = -v / (10*ms) : 1')
P = PoissonInput(G, 'v', 100, 100*Hz, weight=0.1)
```

The *PoissonInput* class is however more restrictive than *PoissonGroup*, it only allows for a constant rate across all neurons (but you can create several *PoissonInput* objects, targeting different subgroups). It internally uses *BinomialFunction* which will draw a random number each time step, either from a binomial distribution or from a normal distribution as an approximation to the binomial distribution if $np > 5 \wedge n(1-p) > 5$, where n is the number of inputs and $p = dt \cdot rate$ the spiking probability for a single input.

3.7.2 Spike generation

You can also generate an explicit list of spikes given via arrays using *SpikeGeneratorGroup*. This object behaves just like a *NeuronGroup* in that you can connect it to other groups via a *Synapses* object, but you specify three bits of information: `N` the number of neurons in the group; `indices` an array of the indices of the neurons that will fire; and `times` an array of the same length as `indices` with the times that the neurons will fire a spike. The `indices` and `times` arrays are matching, so for example `indices=[0,2,1]` and `times=[1*ms,2*ms,3*ms]` means that neuron 0 fires at time 1 ms, neuron 2 fires at 2 ms and neuron 1 fires at 3 ms. Example use:

```
indices = array([0, 2, 1])
times = array([1, 2, 3])*ms
G = SpikeGeneratorGroup(3, indices, times)
```

The spikes that will be generated by *SpikeGeneratorGroup* can be changed between runs with the *set_spikes* method. This can be useful if the input to a system should depend on its previous output or when running multiple trials with different input:

```
inp = SpikeGeneratorGroup(N, indices, times)
G = NeuronGroup(N, '...')
feedforward = Synapses(inp, G, '...', pre='...', connect='i==j')
recurrent = Synapses(G, G, '...', pre='...', connect='i!=j')
spike_mon = SpikeMonitor(G)
# ...
```

```
run(runtime)
# Replay the previous output of group G as input into the group
inp.set_spikes(spike_mon.i, spike_mon.t + runtime)
run(runtime)
```

3.7.3 Explicit equations

If the input can be explicitly expressed as a function of time (e.g. a sinusoidal input current), then its description can be directly included in the equations of the respective group:

```
G = NeuronGroup(100, '''dv/dt = (-v + I)/(10*ms) : 1
                        rates : Hz  # each neuron's input has a different rate
                        size : 1  # and a different amplitude
                        I = size*sin(2*pi*rates*t) : 1''')
G.rates = '10*Hz + i*Hz'
G.size = '(100-i)/100. + 0.1'
```

3.7.4 Timed arrays

If the time dependence of the input cannot be expressed in the equations in the way shown above, it is possible to create a *TimedArray*. Such an objects acts as a function of time where the values at given time points are given explicitly. This can be especially useful to describe non-continuous stimulation. For example, the following code defines a *TimedArray* where stimulus blocks consist of a constant current of random strength for 30ms, followed by no stimulus for 20ms. Note that in this particular example, numerical integration can use exact methods, since it can assume that the *TimedArray* is a constant function of time during a single integration time step. Also note that the semantics of *TimedArray* changed slightly compared to Brian1: for `TimedArray([x1, x2, ...], dt=my_dt)`, the value `x1` will be returned for all `0<=t<my_dt`, `x2` for `my_dt<=t<2*my_dt` etc., whereas Brian1 returned `x1` for `0<=t<0.5*my_dt`, `x2` for `0.5*my_dt<=t<1.5*my_dt`, etc.

```
stimulus = TimedArray(np.hstack([[c, c, c, 0, 0]
                                 for c in np.random.rand(1000)]),
                                dt=10*ms)
G = NeuronGroup(100, 'dv/dt = (-v + stimulus(t))/(10*ms) : 1',
                threshold='v>1', reset='v=0')
G.v = '0.5*rand()'  # different initial values for the neurons
```

TimedArray can take a one-dimensional value array (as above) and therefore return the same value for all neurons or it can take a two-dimensional array with time as the first and (neuron/synapse/...-)index as the second dimension.

In the following, this is used to implement shared noise between neurons, all the "even neurons" get the first noise instantiation, all the "odd neurons" get the second:

```
runtime = 1*second
stimulus = TimedArray(np.random.rand(int(runtime/defaultclock.dt), 2),
                      dt=defaultclock.dt)
G = NeuronGroup(100, 'dv/dt = (-v + stimulus(t, i % 2))/(10*ms) : 1',
                threshold='v>1', reset='v=0')
```

3.7.5 Regular operations

An alternative to specifying a stimulus in advance is to run explicitly specified code at certain points during a simulation. This can be achieved with *run_regularly()*. One can think of these statements as equivalent to reset statements but executed unconditionally (i.e. for all neurons) and possibly on a different clock than the rest of the group. The following code changes the stimulus strength of half of the neurons (randomly chosen) to a new random

value every 50ms. Note that the statement uses logical expressions to have the values only updated for the chosen subset of neurons (where the newly introduced auxiliary variable `change` equals 1):

```
G = NeuronGroup(100, '''dv/dt = (-v + I)/(10*ms) : 1
                        I : 1  # one stimulus per neuron''')
G.run_regularly('''change = int(rand() < 0.5)
                   I = change*(rand()*2) + (1-change)*I''',
                dt=50*ms)
```

3.7.6 Arbitrary Python code (network operations)

If none of the above techniques is general enough to fulfill the requirements of a simulation, Brian allows you to write a *NetworkOperation*, an arbitrary Python function that is executed every time step (possible on a different clock than the rest of the simulation). This function can do arbitrary operations, use conditional statements etc. and it will be executed as it is (i.e. as pure Python code even if weave code generation is active). Note that one cannot use network operations in combination with the C++ standalone mode. Network operations are particularly useful when some condition or calculation depends on operations across neurons, which is currently not possible to express in abstract code. The following code switches input on for a randomly chosen single neuron every 50 ms:

```
G = NeuronGroup(10, '''dv/dt = (-v + active*I)/(10*ms) : 1
                       I = sin(2*pi*100*Hz*t) : 1 (shared) #single input
                       active : 1  # will be set in the network operation''')
@network_operation(dt=50*ms)
def update_active():
    index = np.random.randint(10)  # index for the active neuron
    G.active_ = 0  # the underscore switches off unit checking
    G.active_[index] = 1
```

Note that the network operation (in the above example: `update_active`) has to be included in the *Network* object if one is constructed explicitly.

Only functions with zero or one arguments can be used as a *NetworkOperation*. If the function has one argument then it will be passed the current time `t`:

```
@network_operation(dt=1*ms)
def update_input(t):
    if t>50*ms and t<100*ms:
        pass # do something
```

Note that this is preferable to accessing `defaultclock.t` from within the function – if the network operation is not running on the *defaultclock* itself, then that value is not guaranteed to be correct.

Instance methods can be used as network operations as well, however in this case they have to be constructed explicitly, the *network_operation()* decorator cannot be used:

```
class Simulation(object):
    def __init__(self, data):
        self.data = data
        self.group = NeuronGroup(...)
        self.network_op = NetworkOperation(self.update_func, dt=10*ms)
        self.network = Network(self.group, self.network_op)

    def update_func(self):
        pass # do something

    def run(self, runtime):
        self.network.run(runtime)
```

3.8 Recording during a simulation

Recording variables during a simulation is done with "monitor" objects. Specifically, spikes are recorded with *SpikeMonitor*, the time evolution of variables with *StateMonitor* and the firing rate of a population of neurons with *PopulationRateMonitor*.

Note that all monitors are implement as "groups", so you can get all the stored values in a monitor with the *Group.get_states()* method, which can be useful to dump all recorded data to disk, for example.

3.8.1 Recording spikes

To record spikes from a group `G` simply create a *SpikeMonitor* via `SpikeMonitor(G)`. After the simulation, you can access the attributes `i`, `t`, `it`, `num_spikes` and `count` of the monitor. The `i` and `t` attributes give the array of neuron indices and times of the spikes. For example, if `M.i==[0, 2, 1]` and `M.t==[1*ms, 2*ms, 3*ms]` it means that neuron 0 fired a spike at 1 ms, neuron 2 fired a spike at 2 ms, and neuron 1 fired a spike at 3 ms. Alternatively, you can also call the *spike_trains* method to get a dictionary mapping neuron indices to arrays of spike times, i.e. in the above example, `spike_trains = M.spike_trains(); spike_trains[1]` would return `array([ 3.]) * msecond`. The `num_spikes` attribute gives the total number of spikes recorded, and `count` is an array of the length of the recorded group giving the total number of spikes recorded from each neuron. Finally, the `it` attribute is just the pair `(i, t)` for convenience.

Example:

```
G = NeuronGroup(N, model='...')
M = SpikeMonitor(G)
run(runtime)
plot(M.t/ms, M.i, '.')
```

3.8.2 Recording variables at spike time

By default, a *SpikeMonitor* only records the time of the spike and the index of the neuron that spiked. Sometimes it can be useful to addtionaly record other variables, e.g. the membrane potential for models where the threshold is not at a fixed value. This can be done by providing an extra `variables` argument, the recorded variable can then be accessed as an attribute of the *SpikeMonitor*. To conveniently access the values of a recorded variable for a single neuron, the *SpikeMonitor.values()* method can be used that returns a dictionary with the values for each neuron.:

```
G = NeuronGroup(N, '''dv/dt = (1-v)/(10*ms) : 1
                      v_th : 1''',
                threshold='v > v_th',
                # randomly change the threshold after a spike:
                reset='''v=0
                         v_th = clip(v_th + rand()*0.2 - 0.1, 0.1, 0.9)''')
G.v_th = 0.5
spike_mon = SpikeMonitor(G, variables='v')
run(1*second)
v_values = spike_mon.values('v')
print('Threshold crossing values for neuron 0: {}'.format(v_values[0]))
hist(spike_mon.v, np.arange(0, 1, .1))
show()
```

Note: Spikes are not the only events that can trigger recordings, see Custom events.

3.8.3 Recording variables continuously

To record how a variable evolves over time, use a *StateMonitor*. To use this, you specify the group, variables and indices you want to record from. You specify the variables with a string or list of strings, and the indices either as an array of indices or `True` to record all indices (but beware because this may take a lot of memory).

After the simulation, you can access these variables as attributes of the monitor. They are 2D arrays with shape `(num_indices, num_times)`. The special attribute `t` is an array of length `num_times` with the corresponding times at which the values were recorded.

Note that you can also use *StateMonitor* to record from *Synapses* where the indices are the synapse indices rather than neuron indices.

In this example, we record two variables v and u, and record from indices 0, 10 and 100. Afterwards, we plot the recorded values of v and u from neuron 0:

```
G = NeuronGroup(...)
M = StateMonitor(G, ('v', 'u'), record=[0, 10, 100])
run(...)
plot(M.t/ms, M.v[0]/mV, label='v')
plot(M.t/ms, M.u[0]/mV, label='u')
```

There are two subtly different ways to get the values for specific neurons: you can either index the 2D array stored in the attribute with the variable name (as in the example above) or you can index the monitor itself. The former will use an index relative to the recorded neurons (e.g. `M.v[1]` will return the values for the second *recorded* neuron which is the neuron with the index 10 whereas `M.v[10]` would raise an error because only three neurons have been recorded), whereas the latter will use an absolute index corresponding to the recorded group (e.g. `M[1].v` will raise an error because the neuron with the index 1 has not been recorded and `M[10].v` will return the values for the neuron with the index 10). If all neurons have been recorded (e.g. with `record=True`) then both forms give the same result.

Note that for plotting all recorded values at once, you have to transpose the variable values:

```
plot(M.t/ms, M.v.T/mV)
```

In contrast to previous versions of Brian, the values are recorded at the beginning of a time step and not at the end (you can set the `when` argument when creating a *StateMonitor*, details about scheduling can be found here: Scheduling and custom progress reporting).

3.8.4 Recording population rates

To record the time-varying firing rate of a population of neurons use *PopulationRateMonitor*. After the simulation the monitor will have two attributes `t` and `rate`, the latter giving the firing rate at each time step corresponding to the time in `t`. For example:

```
G = NeuronGroup(...)
M = PopulationRateMonitor(G)
run(...)
plot(M.t/ms, M.rate/Hz)
```

3.9 Running a simulation

To run a simulation, one either constructs a new *Network* object and calls its *Network.run()* method, or uses the "magic" system and a plain *run()* call, collecting all the objects in the current namespace.

Note that Brian has several different ways of running the actual computations, and choosing the right one can make orders of magnitude of difference in terms of simplicity and efficiency. See Computational methods and efficiency for more details.

3.9.1 Magic networks

In most straightforward simulations, you do not have to explicitly create a *Network* object but instead can simply call *run()* to run a simulation. This is what is called the "magic" system, because Brian figures out automatically what you want to do.

When calling *run()*, Brian runs the *collect()* function to gather all the objects in the current context. It will include all the objects that are "visible", i.e. that you could refer to with an explicit name:

```
G = NeuronGroup(10, 'dv/dt = -v / tau : volt')
S = Synapses(G, G, model='w:1', pre='v+=w', connect='i!=j')
mon = SpikeMonitor(G)

run(10*ms)  # will include G, S, mon
```

Note that it will not automatically include objects that are "hidden" in containers, e.g. if you store several monitors in a list. Use an explicit *Network* object in this case. It might be convenient to use the *collect()* function when creating the *Network* object in that case:

```
G = NeuronGroup(10, 'dv/dt = -v / tau : volt')
S = Synapses(G, G, model='w:1', pre='v+=w', connect='i!=j')
monitors = [SpikeMonitor(G), StateMonitor(G, 'v', record=True)]

# a simple run would not include the monitors
net = Network(collect())  # automatically include G and S
net.add(monitors)  # manually add the monitors
```

When you use more than a single *run()* statement, the magic system tries to detect which of the following two situations applies:

1. You want to continue a previous simulation
2. You want to start a new simulation

For this, it uses the following heuristic: if a simulation consists only of objects that have not been run, it will start a new simulation starting at time 0 (corresponding to the creation of a new *Network* object). If a simulation only consists of objects that have been simulated in the previous *run()* call, it will continue that simulation at the previous time.

If neither of these two situations apply, i.e., the network consists of a mix of previously run objects and new objects, an error will be raised. If this is not a mistake but intended (e.g. when a new input source and synapses should be added to a network at a later stage), use an explicit *Network* object.

In these checks, "non-invalidating" objects (i.e. objects that have *BrianObject.invalidates_magic_network* set to `False`) are ignored, e.g. creating new monitors is always possible.

3.9.2 Progress reporting

Especially for long simulations it is useful to get some feedback about the progress of the simulation. Brian offers a few built-in options and an extensible system to report the progress of the simulation. In the *Network.run()* or *run()* call, two arguments determine the output: `report` and `report_period`. When `report` is set to `'text'` or `'stdout'`, the progress will be printed to the standard output, when it is set to `'stderr'`, it will be printed to

"standard error". There will be output at the start and the end of the run, and during the run in `report_period` intervals. It is also possible to do *custom progress reporting*.

3.9.3 Profiling

To get an idea which parts of a simulation take the most time, Brian offers a basic profiling mechanism. If a simulation is run with the `profile=True` keyword argument, it will collect information about the total simulation time for each *`CodeObject`*. This information can then be retrieved from *`Network.profiling_info`*, which contains a list of `(name, time)` tuples or a string summary can be obtained by calling *`profiling_summary()`*. The following example shows profiling output after running the CUBA example (where the neuronal state updates take up the most time):

```
>>> profiling_summary(show=5)  # show the 5 objects that took the longest
Profiling summary
=================
neurongroup_stateupdater    5.54 s    61.32 %
synapses_pre                1.39 s    15.39 %
synapses_1_pre              1.03 s    11.37 %
spikemonitor                0.59 s     6.55 %
neurongroup_thresholder     0.33 s     3.66 %
```

3.9.4 Scheduling

Every simulated object in Brian has three attributes that can be specified at object creation time: `dt`, `when`, and `order`. The time step of the simulation is determined by `dt`, if it is specified, or otherwise by `defaultclock.dt`. Changing this will therefore change the `dt` of all objects that don't specify one.

During a single time step, objects are updated in an order according first to their `when` argument's position in the schedule. This schedule is determined by *`Network.schedule`* which is a list of strings, determining "execution slots" and their order. It defaults to: `['start', 'groups', 'thresholds', 'synapses', 'resets', 'end']`. In addition to the names provided in the schedule, names such as `before_thresholds` or `after_synapses` can be used that are understood as slots in the respective positions. The default for the `when` attribute is a sensible value for most objects (resets will happen in the `reset` slot, etc.) but sometimes it make sense to change it, e.g. if one would like a *`StateMonitor`*, which by default records in the `end` slot, to record the membrane potential before a reset is applied (otherwise no threshold crossings will be observed in the membrane potential traces).

Finally, if during a time step two objects fall in the same execution slot, they will be updated in ascending order according to their `order` attribute, an integer number defaulting to 0. If two objects have the same `when` and `order` attribute then they will be updated in an arbitrary but reproducible order (based on the lexicographical order of their names).

Every new *`Network`* starts a simulation at time 0; *`Network.t`* is a read-only attribute, to go back to a previous moment in time (e.g. to do another trial of a simulation with a new noise instantiation) use the mechanism described below.

For more details, including finer control over the scheduling of operations and changing the value of `dt` between runs see Scheduling and custom progress reporting.

3.9.5 Continuing/repeating simulations

To store the current state of a network, including the time of the simulation, internal variables like triggered but not yet delivered spikes, etc., call *`Network.store()`* which will store the state of all the objects in the network (use a plain *`store()`* if you are using the magic system). You can store more than one snapshot of a system by providing a

name for the snapshot; if *Network.store()* is called without a specified name, `'default'` is used as the name. To restore a network's state, use *Network.restore()*.

The following simple example shows how this system can be used to run several trials of an experiment:

```
# set up the network
G = NeuronGroup(...)
S = Synapses(...)
G.v = ...
S.connect(...)
S.w = ...
spike_monitor = SpikeMonitor(G)
# Snapshot the state
store()

# Run the trials
spike_counts = []
for trial in range(3):
    restore()  # Restore the initial state
    run(...)
    # store the results
    spike_counts.append(spike_monitor.count)
```

The following schematic shows how multiple snapshots can be used to run a network with a separate "train" and "test" phase. After training, the test is run several times based on the trained network. The whole process of training and testing is repeated several times as well:

```
# set up the network
G = NeuronGroup(..., '''...
                     test_input : amp
                     ...''')
S = Synapses(..., '''...
                  plastic : boolean (shared)
                  ...''')
G.v = ...
S.connect(...)
S.w = ...

# First snapshot at t=0
store('initialized')

# Run 3 complete trials
for trial in range(3):
    # Simulate training phase
    restore('initialized')
    S.plastic = True
    run(...)

    # Snapshot after learning
    store('after_learning')

    # Run 5 tests after the training
    for test_number in range(5):
        restore('after_learning')
        S.plastic = False  # switch plasticity off
        G.test_input = test_inputs[test_number]
        # monitor the activity now
        spike_mon = SpikeMonitor(G)
        run(...)
```

```
        # Do something with the result
        # ...
```

Note that *Network.run()*, *Network.store()* and *Network.restore()* (or *run()*, *store()*, *restore()*) are the only way of affecting the time of the clocks. In contrast to Brian1, it is no longer necessary (nor possible) to directly set the time of the clocks or call a `reinit` function.

3.10 Computational methods and efficiency

Brian has several different methods for running the computations in a simulation. In particular, Brian uses "runtime code generation" for efficient computation. This means that it takes the Python code and strings in your model and generates code in one of several possible different languages and actually executes that. The target language for this code generation process is set in the *codegen.target* preference. By default, this preference is set to `'auto'`, meaning that it will chose a compiled language target if possible and fall back to Python otherwise. There are two compiled language targets for Python 2.x, `'weave'` (needing a working installation of a C++ compiler) and `'cython'` (needing the Cython package in addition); for Python 3.x, only `'cython'` is available. If you want to chose a code generation target explicitly (e.g. because you want to get rid of the warning that only the Python fallback is available), set the preference to `'numpy'`, `'weave'` or `'cython'` at the beginning of your script:

```
from brian2 import *
prefs.codegen.target = 'numpy'  # use the Python fallback
```

See Preferences for different ways of setting preferences. If you are using a compiled language target, also see the *Compiler settings for maximum speed* section below.

Both of these code generation targets are still run via Python, which means that there are still overheads due to Python. The fastest way to run Brian is in "standalone mode" (see Devices), although this won't work for every possible simulation. Note that you can also use multiple threads with standalone mode, which is not possible in the modes described above. This doesn't always lead to a huge speed improvement, but can occasionally give a higher than linear speed up relative to the number of cores.

You might find that running simulations in weave or Cython modes won't work or is not as efficient as you were expecting. This is probably because you're using Python functions which are not compatible with weave or Cython. For example, if you wrote something like this it would not be efficient:

```
from brian2 import *
prefs.codegen.target = 'cython'
def f(x):
    return abs(x)
G = NeuronGroup(10000, 'dv/dt = -x*f(x) : 1')
```

The reason is that the function `f(x)` is a Python function and so cannot be called from C++ directly. To solve this problem, you need to provide an implementation of the function in the target language. See Functions.

3.10.1 Compiler settings for maximum speed

If using C++ code generation (either via weave, cython or standalone), you can maximise the efficiency of the generated code in various ways, described below. These can be set in the global preferences file as described in Preferences.

GCC

For the GCC compiler, the fastest options are:

```
codegen.cpp.extra_compile_args_gcc = ['-w', '-Ofast', '-march=native']
```

The `-Ofast` optimisation allows the compiler to disregard strict IEEE standards compliance. In our usage this has never been a problem, but we don't do this by default for safety. Note that not all versions of gcc include this switch, older versions might require you to write `'-O3', '-ffast-math'`.

The `-march=native` sets the computer architecture to be the one available on the machine you are compiling on. This allows the compiler to make use of as many advanced instructions as possible, but reduces portability of the generated executable (which is not usually an issue). Again, this option is not available on all versions of gcc so on an older version you might have to put your architecture in explicitly (check the gcc docs for your version).

MSVC

For the MSVC compiler, the fastest options are:

```
codegen.cpp.extra_compile_args_msvc = ['/Ox', '/EHsc', '/w', '/arch:AVX2', '/fp:fast']
```

Note that as above for `-Ofast` on gcc, `/fp:fast` will enable the compiler to disregard strict IEEE standards compliance, which has never been a problem in our usage but we leave this off by default for safety.

The `/arch:AVX2` option may not be available on your version of MSVC and your computer architecture. The available options (in order from best to worst) are: `AVX2`, `AVX`, `SSE2`, `SSE` and `IA32`.

3.11 Functions

All equations, expressions and statements in Brian can make use of mathematical functions. However, functions have to be prepared for use with Brian for two reasons: 1) Brian is strict about checking the consistency of units, therefore every function has to specify how it deals with units; 2) functions need to be implemented differently for different code generation targets.

Brian provides a number of default functions that are already prepared for use with numpy and C++ and also provides a mechanism for preparing new functions for use (see below).

3.11.1 Default functions

The following functions (stored in the `DEFAULT_FUNCTIONS` dictionary) are ready for use:

- Random numbers: `rand()`, `randn()` (Note that these functions should be called without arguments, the code generation process will take care of generating an array of numbers for numpy).
- Elementary functions: `sqrt`, `exp`, `log`, `log10`, `abs`, `sign`
- Trigonometric functions: `sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`, `arcsin`, `arccos`, `arctan`
- General utility functions: `clip`, `floor`, `ceil`

Brian also provides a special purpose function `int`, which can be used to convert a boolean expression or variable into an integer value of 0 or 1. This is useful to have a conditional evaluation as part of an equation or statement. This sometimes allows to circumvent the lack of an `if` statement. For example, the following reset statement resets the variable `v` to either `v_r1` or `v_r2`, depending on the value of `w`: `'v = v_r1 * int(w <= 0.5) + v_r2 * int(w > 0.5)'`

3.11.2 User-provided functions

Python code generation

If a function is only used in contexts that use Python code generation, preparing a function for use with Brian only means specifying its units. The simplest way to do this is to use the *check_units()* decorator:

```
@check_units(x1=meter, y1=meter, x2=meter, y2=meter, result=meter)
def distance(x1, y1, x2, y2):
    return sqrt((x1 - x2)**2 + (y1 - y2)**2)
```

Another option is to wrap the function in a *Function* object:

```
def distance(x1, y1, x2, y2):
    return sqrt((x1 - x2)**2 + (y1 - y2)**2)
# wrap the distance function
distance = Function(distance, arg_units=[meter, meter, meter, meter],
                    return_unit=meter)
```

The use of Brian's unit system has the benefit of checking the consistency of units for every operation but at the expense of performance. Consider the following function, for example:

```
@check_units(I=amp, result=Hz)
def piecewise_linear(I):
    return clip((I-1*nA) * 50*Hz/nA, 0*Hz, 100*Hz)
```

When Brian runs a simulation, the state variables are stored and passed around without units for performance reasons. If the above function is used, however, Brian adds units to its input argument so that the operations inside the function do not fail with dimension mismatches. Accordingly, units are removed from the return value so that the function output can be used with the rest of the code. For better performance, Brian can alter the namespace of the function when it is executed as part of the simulation and remove all the units, then pass values without units to the function. In the above example, this means making the symbol `nA` refer to `1e-9` and `Hz` to `1`. To use this mechanism, add the decorator *implementation()* with the `discard_units` keyword:

```
@implementation('numpy', discard_units=True)
@check_units(I=amp, result=Hz)
def piecewise_linear(I):
    return clip((I-1*nA) * 50*Hz/nA, 0*Hz, 100*Hz)
```

Note that the use of the function *outside of simulation runs* is not affected, i.e. using `piecewise_linear` still requires a current in Ampere and returns a rate in Hertz. The `discard_units` mechanism does not work in all cases, e.g. it does not work if the function refers to units as `brian2.nA` instead of `nA`, if it uses imports inside the function (e.g. `from brian2 import nA`), etc. The `discard_units` can also be switched on for all functions without having to use the *implementation()* decorator by setting the *codegen.runtime.numpy.discard_units* preference.

Other code generation targets

To make a function available for other code generation targets (e.g. C++), implementations for these targets have to be added. This can be achieved using the *implementation()* decorator. The form of the code (e.g. a simple string or a dictionary of strings) necessary is target-dependent, for C++ both options are allowed, a simple string will be interpreted as filling the `'support_code'` block. Note that both `'cpp'` and `'weave'` can be used to provide C++ implementations, the first should be used for generic C++ implementations, and the latter if weave-specific code is necessary. An implementation for the C++ target could look like this:

```
@implementation('cpp', '''
    double piecewise_linear(double I) {
        if (I < 1e-9)
```

```
            return 0;
        if (I > 3e-9)
            return 100;
        return (I/1e-9 - 1) * 50;
    }
    ''')
@check_units(I=amp, result=Hz)
def piecewise_linear(I):
    return clip((I-1*nA) * 50*Hz/nA, 0*Hz, 100*Hz)
```

Alternatively, `FunctionImplementation` objects can be added to the *Function* object.

The same sort of approach as for C++ works for Cython using the `'cython'` target. The example above would look like this:

```
@implementation('cython', '''
    cdef double piecewise_linear(double I):
        if I<1e-9:
            return 0.0
        elif I>3e-9:
            return 100.0
        return (I/1e-9-1)*50
    ''')
@check_units(I=amp, result=Hz)
def piecewise_linear(I):
    return clip((I-1*nA) * 50*Hz/nA, 0*Hz, 100*Hz)
```

Arrays vs. scalar values in user-provided functions

Equations, expressions and abstract code statements are always implicitly referring to all the neurons in a *NeuronGroup*, all the synapses in a *Synapses* object, etc. Therefore, function calls also apply to more than a single value. The way in which this is handled differs between code generation targets that support vectorized expressions (e.g. the `numpy` target) and targets that don't (e.g. the `weave` target or the `cpp_standalone` mode). If the code generation target supports vectorized expressions, it will receive an array of values. For example, in the `piecewise_linear` example above, the argument `I` will be an array of values and the function returns an array of values. For code generation without support for vectorized expressions, all code will be executed in a loop (over neurons, over synapses, ...), the function will therefore be called several times with a single value each time.

In both cases, the function will only receive the "relevant" values, meaning that if for example a function is evaluated as part of a reset statement, it will only receive values for the neurons that just spiked.

Additional namespace

Some functions need additional data to compute a result, e.g. a *TimedArray* needs access to the underlying array. For the `numpy` target, a function can simply use a reference to an object defined outside the function, there is no need to explicitly pass values in a namespace. For the other code language targets, values can be passed in the `namespace` argument of the *implementation()* decorator or the *add_implementation* method. The namespace values are then accessible in the function code under the given name, prefixed with `_namespace`. Note that this mechanism should only be used for numpy arrays or general objects (e.g. function references to call Python functions from weave or Cython code). Scalar values should be directly included in the function code, by using a "dynamic implemention" (see *add_dynamic_implementation*).

See *TimedArray* and *BinomialFunction* for examples that use this mechanism.

3.12 Devices

Brian supports generating standalone code for multiple devices. In this mode, running a Brian script generates source code in a project tree for the target device/language. This code can then be compiled and run on the device, and modified if needed. At the moment, the only 'device' supported is standalone C++ code. In some cases, the speed gains can be impressive, in particular for smaller networks with complicated spike propagation rules (such as STDP).

3.12.1 C++ standalone

To use the C++ standalone mode, make the following changes to your script:

1. At the beginning of the script, i.e. after the import statements, add:

```
set_device('cpp_standalone')
```

2. After `run(duration)` in your script, add:

```
device.build(directory='output', compile=True, run=True, debug=False)
```

The `build` function has several arguments to specify the output directory, whether or not to compile and run the project after creating it (using `gcc`) and whether or not to compile it with debugging support or not.

Not all features of Brian will work with C++ standalone, in particular Python based network operations and some array based syntax such as `S.w[0, :] = ...` will not work. If possible, rewrite these using string based syntax and they should work. Also note that since the Python code actually runs as normal, code that does something like this may not behave as you would like:

```
results = []
for val in vals:
    # set up a network
    run()
    results.append(result)
```

The current C++ standalone code generation only works for a fixed number of `run` statements, not with loops. If you need to do loops or other features not supported automatically, you can do so by inspecting the generated C++ source code and modifying it, or by inserting code directly into the main loop as follows:

```
device.insert_code('main', '''
cout << "Testing direct insertion of code." << endl;
''')
```

After a simulation has been run (using the `run` keyword in the `Device.build` call), state variables and monitored variables can be accessed using standard syntax, with a few exceptions (e.g. string expressions for indexing).

Multi-threading with OpenMP

Warning: OpenMP code has not yet been well tested and so may be inaccurate.

When using the C++ standalone mode, you have the opportunity to turn on multi-threading, if your C++ compiler is compatible with OpenMP. By default, this option is turned off and only one thread is used. However, by changing the preferences of the codegen.cpp_standalone object, you can turn it on. To do so, just add the following line in your python script:

```
prefs.devices.cpp_standalone.openmp_threads = XX
```

XX should be a positive value representing the number of threads that will be used during the simulation. Note that the speedup will strongly depend on the network, so there is no guarantee that the speedup will be linear as a function of the number of threads. However, this is working fine for networks with not too small timestep (dt > 0.1ms), and results do not depend on the number of threads used in the simulation.

3.13 Brian 1 Hears bridge

The "Brian Hears" library is being rewritten for Brian 2 in a more flexible way to allow it to work on multiple devices. Until this work is complete, it is possible to run the version of Brian Hears from Brian 1.x using the "bridge". To make this work, you must have a copy of Brian 1 installed (preferably the latest version), and import Brian Hears using:

```
from brian2.hears import *
```

Many scripts will run without any changes, but there are a few caveats to be aware of. Mostly, the problems are due to the fact that the units system in Brian 2 is not 100% compatible with the units system of Brian 1.

FilterbankGroup now follows the rules for *NeuronGroup* in Brian 2, which means some changes may be necessary to match the syntax of Brian 2, for example, the following would work in Brian 1 Hears:

```
# Leaky integrate-and-fire model with noise and refractoriness
eqs = '''
dv/dt = (I-v)/(1*ms)+0.2*xi*(2/(1*ms))**.5 : 1
I : 1
'''
anf = FilterbankGroup(ihc, 'I', eqs, reset=0, threshold=1, refractory=5*ms)
```

However, in Brian 2 Hears you would need to do:

```
# Leaky integrate-and-fire model with noise and refractoriness
eqs = '''
dv/dt = (I-v)/(1*ms)+0.2*xi*(2/(1*ms))**.5 : 1 (unless refractory)
I : 1
'''
anf = FilterbankGroup(ihc, 'I', eqs, reset='v=0', threshold='v>1', refractory=5*ms)
```

Slicing sounds no longer works. Previously you could do, e.g. `sound[:20*ms]` but with Brian 2 you would need to do `sound.slice(0*ms, 20*ms)`.

In addition, some functions may not work correctly with Brian 2 units. In most circumstances, Brian 2 units can be used interchangeably with Brian 1 units in the bridge, but in some cases it may be necessary to convert units from one format to another, and to do that you can use the functions *convert_unit_b1_to_b2* and *convert_unit_b2_to_b1*.

3.14 Multicompartment models

It is possible to create neuron models with a spatially extended morphology, using the *SpatialNeuron* class. A *SpatialNeuron* is a single neuron with many compartments. Essentially, it works as a *NeuronGroup* where elements are compartments instead of neurons. A *SpatialNeuron* is specified mainly by a set of equations for transmembrane currents (ionic channels) and a morphology.

3.14.1 Creating a neuron morphology

Morphologies are stored and manipulated with *Morphology* objects. A new morphology can be loaded from a `.swc` file (a standard format for neuronal morphologies):

```
morpho = Morphology('corticalcell.swc')
```

There is a large database of morphologies in the swc format at http://neuromorpho.org/neuroMorpho.

Morphologies can also be created manually by combining different standard geometrical objects:

```
morpho = Soma(diameter = 30*um)
morpho = Cylinder(length = 100*um, diameter = 1*um, n = 10)
```

The first statement creates a sphere (as a single compartment), the second one a cylinder with 10 compartments (default is 1 compartment). It is also possible to specify the type of process, which is `soma`, `axon` or `dendrite`, and the relative coordinates of the end point. For example:

```
morpho = Cylinder(diameter = 1*um, n = 10, type = 'axon', x = 50*um, y = 100*um, z = 0*um)
```

In this case, length must not be specified, as it is calculated from the coordinates. These coordinates are mostly helpful for visualization. If they are not specified, 3D direction is chosen at random.

A tree is created by attaching *`Morphology`* objects together:

```
morpho = Soma(diameter = 30*um)
morpho.axon = Cylinder(length = 100*um, diameter = 1*um, n = 10)
morpho.dendrite = Cylinder(length = 50*um, diameter = 2*um, n = 5)
```

These statements create a morphology consisting of a cylindrical axon and a dendrite attached to a spherical soma. Note that the names `axon` and `dendrite` are arbitrary and chosen by the user. For example, the same morphology can be created as follows:

```
morpho = Soma(diameter = 30*um)
morpho.output_process = Cylinder(length = 100*um, diameter = 1*um, n = 10)
morpho.input_process = Cylinder(length = 50*um, diameter = 2*um, n = 5)
```

The syntax is recursive, for example two branches can be added at the end of the dendrite as follows:

```
morpho.dendrite.branch1 = Cylinder(length = 50*um, diameter = 1*um, n = 3)
morpho.dendrite.branch2 = Cylinder(length = 50*um, diameter = 1*um, n = 3)
```

Equivalently, one can use an indexing syntax:

```
morpho['dendrite']['branch1'] = Cylinder(length = 50*um, diameter = 1*um, n = 3)
morpho['dendrite']['branch2'] = Cylinder(length = 50*um, diameter = 1*um, n = 3)
```

Finally there is a special shorter syntax for quickly creating trees, using `L` (for left), `R` (for right), and digits from 1 to 9. These can be simply concatenated (without using the dot):

```
morpho.L=Cylinder(length = 10*um, diameter = 1 *um, n = 3)
morpho.L1=Cylinder(length = 5*um, diameter = 1 *um, n = 3)
morpho.L2=Cylinder(length = 5*um, diameter = 1 *um, n = 3)
morpho.L3=Cylinder(length = 5*um, diameter = 1 *um, n = 3)
morpho.R=Cylinder(length = 10*um, diameter = 1 *um, n = 3)
morpho.RL=Cylinder(length = 5*um, diameter = 1 *um, n = 3)
morpho.RR=Cylinder(length = 5*um, diameter = 1 *um, n = 3)
```

These instructions create a dendritic tree with two main branches, 3 subbranches on the first branch and 2 on the second branch. After these instructions are executed, `morpho.L` contains the entire subtree. To retrieve only the primary branch of this subtree, use the `main` attribute:

```
mainbranch = morpho.L.main
```

The number of compartments in the entire tree can be obtained with `len(morpho)`. Finally, the morphology can be displayed as a 3D plot:

```
morpho.plot()
```

Complex morphologies

Neuronal morphologies can be created by assembling cylinders and spheres, but also more complex processes with variable diameter. This can be done by directly setting the attributes of a *Morphology* object. A *Morphology* object stores the `diameter`, `x`, `y`, `z`, `length` and `area` of all the compartments of the main branch (i.e., not children) as arrays. For example, `morpho.length` is an array containing the length of each of its compartments. When creating a cylinder, the length of each compartment is obtained by dividing the total length provided at creation by the number of compartments. The area is calculated automatically. Complex processes can be created manually by directly specifying the diameter and length of each compartment:

```
morpho.axon = Morphology(n = 5)
morpho.axon.diameter = ones(5) * 1 * um
morpho.axon.length = [1, 2, 1, 3, 1] * um
morpho.axon.set_coordinates()
morpho.axon.set_area()
```

Note the last two statements: `set_coordinates()` creates x-y-z coordinates and is required for plotting; `set_area()` calculates the area of each compartment (considered as a cylinder) and is required for using the morphology in simulations. Alternatively the coordinates can be specified, instead of the lengths of compartments, and then `set_length()` must be called. Note that these methods only apply to the main branch of the morphology, not the children (subtrees).

3.14.2 Creating a spatially extended neuron

A *SpatialNeuron* is a spatially extended neuron. It is created by specifying the morphology as a *Morphology* object, the equations for transmembrane currents, and optionally the specific membrane capacitance `Cm` and intracellular resistivity `Ri`:

```
gL=1e-4*siemens/cm**2
EL=-70*mV
eqs='''
Im=gL*(EL-v) : amp/meter**2
I : amp (point current)
'''
neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=1 * uF / cm ** 2, Ri=100 * ohm * cm)
neuron.v = EL+10*mV
```

Several state variables are created automatically: all the variables of the morphology object are linked to state variables of the neuron (`diameter`, `x`, `y`, `z`, `length` and `area`). Additionally, a state variable `Cm` is created. It is initialized with the value given at construction, but it can be modified on a compartment per compartment basis (which is useful to model myelinated axons). Finally the membrane potential is stored in state variable `v`. The integration method can be specified as for a *NeuronGroup* with the `method` keyword. In general, for models with nonlinear conductances, the exponential Euler method should be used: `method = "exponential_euler"`.

The key state variable, which must be specified at construction, is `Im`. It is the total transmembrane current, expressed in units of current per area. This is a mandatory line in the definition of the model. The rest of the string description may include other state variables (differential equations or subexpressions) or parameters, exactly as in *NeuronGroup*. At every timestep, Brian integrates the state variables, calculates the transmembrane current at every point on the neuronal morphology, and updates `v` using the transmembrane current and the diffusion current, which is calculated based on the morphology and the intracellular resistivity. Note that the transmembrane current is a surfacic current, not the total current in the compartement. This choice means that the model equations are independent of the number of compartments chosen for the simulation. The space constant can obtained for any point of the neuron with the `space_constant` attribute:

```
l = neuron.space_constant[0]
```

The calculation is based on the local total conductance (not just the leak conductance). Therefore, it can potentially vary during a simulation (e.g. decrease during an action potential).

To inject a current `I` at a particular point (e.g. through an electrode or a synapse), this current must be divided by the area of the compartment when inserted in the transmembrane current equation. This is done automatically when the flag `point current` is specified, as in the example above. This flag can apply only to subexpressions or parameters with amp units. Internally, the expression of the transmembrane current `Im` is simply augmented with `+I/area`. A current can then be injected in the first compartment of the neuron (generally the soma) as follows:

```
neuron.I[0]=1*nA
```

State variables of the *SpatialNeuron* include all the compartments of that neuron (including subtrees). Therefore, the statement `neuron.v=EL+10*mV` sets the membrane potential of the entire neuron at -60 mV.

Subtrees can be accessed by attribute (in the same way as in *Morphology* objects):

```
neuron.axon.gNa = 10*gL
```

Note that the state variables correspond to the entire subtree, not just the main branch. That is, if the axon had branches, then the above statement would change `gNa` on the main branch and all the subbranches. To access the main branch only, use the attribute `main`:

```
neuron.axon.main.gNa = 10*gL
```

A typical use case is when one wants to change parameter values at the soma only. For example, inserting an electrode current at the soma is done as follows:

```
neuron.main.I = 1*nA
```

A part of a branch can be accessed as follows:

```
initial_segment = neuron.axon[10*um:50*um]
```

3.14.3 Synaptic inputs

There are two methods to have synapses on *SpatialNeuron*. The first one to insert synaptic equations directly in the neuron equations:

```
eqs='''
Im = gL*(EL-v) : amp/meter**2
Is = gs*(Es-v) : amp (point current)
dgs/dt = -gs/taus : siemens
'''
neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=1 * uF / cm ** 2, Ri=100 * ohm * cm)
```

Note that, as for electrode stimulation, the synaptic current must be defined as a point current. Then we use a *Synapses* object to connect a spike source to the neuron:

```
S = Synapses(stimulation,neuron,pre = 'gs += w')
S.connect(0,50)
S.connect(1,100)
```

This creates two synapses, on compartments 50 and 100. One can specify the compartment number with its spatial position by indexing the morphology:

```
S.connect(0,morpho[25*um])
S.connect(1,morpho.axon[30*um])
```

In this method for creating synapses, there is a single value for the synaptic conductance in any compartment. This means that it will fail if there are several synapses onto the same compartment and synaptic equations are nonlinear. The second method, which works in such cases, is to have synaptic equations in the *Synapses* object:

```
eqs='''
Im = gL*(EL-v) : amp/meter**2
Is = gs*(Es-v) : amp (point current)
gs : siemens
'''
neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=1 * uF / cm ** 2, Ri=100 * ohm * cm)
S = Synapses(stimulation,neuron,model='''dg/dt = -g/taus : siemens
                                         gs_post = g : siemens (summed)''',pre = 'g += w')
```

Here each synapse (instead of each compartment) has an associated value `g`, and all values of `g` for each compartment (i.e., all synapses targeting that compartment) are collected into the compartmental variable `gs`.

3.14.4 Detecting spikes

To detect and record spikes, we must specify a threshold condition, essentially in the same way as for a *NeuronGroup*:

```
neuron = SpatialNeuron(morphology=morpho, model=eqs, threshold = "v > 0*mV", refractory = "v > -10*mV
```

Here spikes are detected when the membrane potential `v` reaches 0 mV. Because there is generally no explicit reset in this type of model (although it is possible to specify one), `v` remains above 0 mV for some time. To avoid detecting spikes during this entire time, we specify a refractory period. In this case no spike is detected as long as `v` is greater than -10 mV. Another possibility could be:

```
neuron = SpatialNeuron(morphology=morpho, model=eqs, threshold = "m > 0.5", refractory = "m > 0.4")
```

where `m` is the state variable for sodium channel activation (assuming this has been defined in the model). Here a spike is detected when half of the sodium channels are open.

With the syntax above, spikes are detected in all compartments of the neuron. To detect them in a single compartment, use the `threshold_location` keyword:

```
neuron = SpatialNeuron(morphology=morpho, model=eqs, threshold = "m > 0.5", threshold_location = 30,
                       refractory = "m > 0.4")
```

In this case, spikes are only detecting in compartment number 30. Reset then applies locally to that compartment (if a reset statement is defined). Again the location of the threshold can be specified with spatial position:

```
neuron = SpatialNeuron(morphology=morpho, model=eqs, threshold = "m > 0.5",
                       threshold_location = morpho.axon[30*um],
                       refractory = "m > 0.4")
```

CHAPTER 4

Advanced guide

This section has additional information on details not covered in the User's guide.

4.1 Preferences

Brian has a system of global preferences that affect how certain objects behave. These can be set either in scripts by using the *prefs* object or in a file. Each preference looks like `codegen.c.compiler`, i.e. dotted names.

4.1.1 Accessing and setting preferences

Preferences can be accessed and set either keyword-based or attribute-based. The following are equivalent:

```
prefs['codegen.c.compiler'] = 'gcc'
prefs.codegen.c.compiler = 'gcc'
```

Using the attribute-based form can be particulary useful for interactive work, e.g. in ipython, as it offers autocompletion and documentation. In ipython, `prefs.codegen.c?` would display a docstring with all the preferences available in the `codegen.c` category.

4.1.2 Preference files

Preferences are stored in a hierarchy of files, with the following order (each step overrides the values in the previous step but no error is raised if one is missing):

- The global defaults are stored in the installation directory.
- The user default are stored in `~/.brian/user_preferences` (which works on Windows as well as Linux). The ~ symbol refers to the user directory.
- The file `brian_preferences` in the current directory.

The preference files are of the following form:

```
a.b.c = 1
# Comment line
[a]
b.d = 2
[a.b]
b.e = 3
```

This would set preferences `a.b.c=1`, `a.b.d=2` and `a.b.e=3`.

4.1.3 List of preferences

Brian itself defines the following preferences (including their default values):

codegen

Code generation preferences `codegen.loop_invariant_optimisations = True`

Whether to pull out scalar expressions out of the statements, so that they are only evaluated once instead of once for every neuron/synapse/... Can be switched off, e.g. because it complicates the code (and the same optimisation is already performed by the compiler) or because the code generation target does not deal well with it. Defaults to `True`.

`codegen.string_expression_target = 'numpy'`

Default target for the evaluation of string expressions (e.g. when indexing state variables). Should normally not be changed from the default numpy target, because the overhead of compiling code is not worth the speed gain for simple expressions.

Accepts the same arguments as *codegen.target*, except for `'auto'`

`codegen.target = 'auto'`

Default target for code generation.

Can be a string, in which case it should be one of:

- `'auto'` the default, automatically chose the best code generation target available.
- `'weave'` uses `scipy.weave` to generate and compile C++ code, should work anywhere where `gcc` is installed and available at the command line.
- `'cython'`, uses the Cython package to generate C++ code. Needs a working installation of Cython and a C++ compiler.
- `'numpy'` works on all platforms and doesn't need a C compiler but is often less efficient.

Or it can be a `CodeObject` class.

codegen.cpp

C++ compilation preferences `codegen.cpp.compiler = ''`

Compiler to use (uses default if empty)

Should be gcc or msvc.

`codegen.cpp.define_macros = []`

List of macros to define; each macro is defined using a 2-tuple, where 'value' is either the string to define it to or None to define it without a particular value (equivalent of "#define FOO" in source or -DFOO on Unix C compiler command line).

`codegen.cpp.extra_compile_args = None`

Extra arguments to pass to compiler (if None, use either `extra_compile_args_gcc` or `extra_compile_args_msvs`).

`codegen.cpp.extra_compile_args_gcc = ['-w', '-O3']`

Extra compile arguments to pass to GCC compiler

`codegen.cpp.extra_compile_args_msvc = ['/Ox', '/EHsc', '/w']`

Extra compile arguments to pass to MSVC compiler

`codegen.cpp.extra_link_args = []`

Any extra platform- and compiler-specific information to use when linking object files together.

`codegen.cpp.headers = []`

A list of strings specifying header files to use when compiling the code. The list might look like ["<vector>","'my_header'"]. Note that the header strings need to be in a form than can be pasted at the end of a #include statement in the C++ code.

`codegen.cpp.include_dirs = []`

Include directories to use. Note that `$prefix/include` will be appended to the end automatically, where `$prefix` is Python's site-specific directory prefix as returned by `sys.prefix`.

`codegen.cpp.libraries = []`

List of library names (not filenames or paths) to link against.

`codegen.cpp.library_dirs = []`

List of directories to search for C/C++ libraries at link time.

`codegen.cpp.msvc_architecture = ''`

MSVC architecture name (or use system architectue by default).

Could take values such as x86, amd64, etc.

`codegen.cpp.msvc_vars_location = ''`

Location of the MSVC command line tool (or search for best by default).

`codegen.cpp.runtime_library_dirs = []`

List of directories to search for C/C++ libraries at run time.

codegen.generators

Codegen generator preferences (see subcategories for individual languages)

codegen.generators.cpp

C++ codegen preferences `codegen.generators.cpp.flush_denormals = False`

Adds code to flush denormals to zero.

The code is gcc and architecture specific, so may not compile on all platforms. The code, for reference is:

```
#define CSR_FLUSH_TO_ZERO         (1 << 15)
unsigned csr = __builtin_ia32_stmxcsr();
csr |= CSR_FLUSH_TO_ZERO;
__builtin_ia32_ldmxcsr(csr);
```

Found at http://stackoverflow.com/questions/2487653/avoiding-denormal-values-in-c.

`codegen.generators.cpp.restrict_keyword = '__restrict'`

The keyword used for the given compiler to declare pointers as restricted.

This keyword is different on different compilers, the default works for gcc and MSVS.

codegen.runtime

Runtime codegen preferences (see subcategories for individual targets)

codegen.runtime.cython

Cython runtime codegen preferences `codegen.runtime.cython.multiprocess_safe = True`

Whether to use a lock file to prevent simultaneous write access to cython .pyx and .so files.

codegen.runtime.numpy

Numpy runtime codegen preferences `codegen.runtime.numpy.discard_units = False`

Whether to change the namespace of user-specifed functions to remove units.

core

Core Brian preferences `core.default_float_dtype = float64`

Default dtype for all arrays of scalars (state variables, weights, etc.).

`core.default_integer_dtype = int32`

Default dtype for all arrays of integer scalars.

`core.outdated_dependency_error = True`

Whether to raise an error for outdated dependencies (`True`) or just a warning (`False`).

core.network

Network preferences `core.network.default_schedule = ['start', 'groups', 'thresholds', 'synapses', 'resets', 'end']`

Default schedule used for networks that don't specify a schedule.

devices

Device preferences

devices.cpp_standalone

C++ standalone preferences `devices.cpp_standalone.openmp_threads = 0`

The number of threads to use if OpenMP is turned on. By default, this value is set to 0 and the C++ code is generated without any reference to OpenMP. If greater than 0, then the corresponding number of threads are used to launch the simulation.

logging

Logging system preferences `logging.console_log_level = 'WARNING'`

What log level to use for the log written to the console.

Has to be one of CRITICAL, ERROR, WARNING, INFO or DEBUG.

`logging.delete_log_on_exit = True`

Whether to delete the log and script file on exit.

If set to `True` (the default), log files (and the copy of the main script) will be deleted after the brian process has exited, unless an uncaught exception occured. If set to `False`, all log files will be kept.

`logging.file_log = True`

Whether to log to a file or not.

If set to `True` (the default), logging information will be written to a file. The log level can be set via the *logging.file_log_level* preference.

`logging.file_log_level = 'DEBUG'`

What log level to use for the log written to the log file.

In case file logging is activated (see *logging.file_log*), which log level should be used for logging. Has to be one of CRITICAL, ERROR, WARNING, INFO or DEBUG.

`logging.save_script = True`

Whether to save a copy of the script that is run.

If set to `True` (the default), a copy of the currently run script is saved to a temporary location. It is deleted after a successful run (unless *logging.delete_log_on_exit* is `False`) but is kept after an uncaught exception occured. This can be helpful for debugging, in particular when several simulations are running in parallel.

`logging.std_redirection = True`

Whether or not to redirect stdout/stderr to null at certain places.

This silences a lot of annoying compiler output, but will also hide error messages making it harder to debug problems. You can always temporarily switch it off when debugging. In any case, the output is saved to a file and if an error occurs the name of this file will be printed.

4.2 Namespaces

Equations can contain references to external parameters or functions. During the initialisation of a *NeuronGroup* or a *Synapses* object, this *namespace* can be provided as an argument. This is a group-specific namespace that will only be used for names in the context of the respective group. Note that units and a set of standard functions are always provided and should not be given explicitly. This namespace does not necessarily need to be exhaustive at the time of the creation of the *NeuronGroup*/*Synapses*, entries can be added (or modified) at a later stage via the `namespace` attribute (e.g. `G.namespace['tau'] = 10*ms`).

At the point of the call to the *Network.run()* namespace, any group-specific namespace will be augmented by the "run namespace". This namespace can be either given explicitly as an argument to the `run` method or it will be taken from the locals and globals surrounding the call. A warning will be emitted if a name is defined in more than one namespace.

To summarize: an external identifier will be looked up in the context of an object such as *NeuronGroup* or *Synapses*. It will follow the following resolution hierarchy:

1. Default unit and function names.
2. Names defined in the explicit group-specific namespace.
3. Names in the run namespace which is either explicitly given or the implicit namespace surrounding the run call.

Note that if you completely specify your namespaces at the *Group* level, you should probably pass an empty dictionary as the namespace argument to the `run` call – this will completely switch off the "implicit namespace" mechanism.

The following three examples show the different ways of providing external variable values, all having the same effect in this case:

```
# Explicit argument to the NeuronGroup
G = NeuronGroup(1, 'dv/dt = -v / tau : 1', namespace={'tau': 10*ms})
net = Network(G)
net.run(10*ms)

# Explicit argument to the run function
G = NeuronGroup(1, 'dv/dt = -v / tau : 1')
net = Network(G)
```

```
net.run(10*ms, namespace={'tau': 10*ms})

# Implicit namespace from the context
G = NeuronGroup(1, 'dv/dt = -v / tau : 1')
net = Network(G)
tau = 10*ms
net.run(10*ms)
```

External variables are free to change between runs (but not during one run), the value at the time of the *run()* call is used in the simulation.

4.3 Refractoriness

Internally, a *NeuronGroup* with refractoriness has a boolean variable `not_refractory` added to the equations, and this is used to implement the refractoriness behaviour. Specifically, the `threshold` condition is replaced by `threshold and not_refractory` and differential equations that are marked as `(unless refractory)` are multiplied by `int(not_refractory)` (so that they have the value 0 when the neuron is refractory).

This `not_refractory` variable is also available to the user to define more sophisticated refractoriness behaviour. For example, the following code updates the `w` variable with a different time constant during refractoriness:

```
G = NeuronGroup(N, '''dv/dt = -(v + w)/ tau_v : 1 (unless refractory)
                      dw/dt = (-w / tau_active)*int(not_refractory) + (-w / tau_ref)*(1 - int(not_ref
                threshold='v > 1', reset='v=0; w+=0.1', refractory=2*ms)
```

4.4 Scheduling and custom progress reporting

4.4.1 Scheduling

Every simulated object in Brian has three attributes that can be specified at object creation time: `dt`, `when`, and `order`. The time step of the simulation is determined by `dt`, if it is specified, a new *Clock* with the given `dt` will be created for the object. Alternatively, a `clock` object can be specified directly, this can be useful if a clock should be shared between several objects – under most circumstances, however, a user should not have to deal with the creation of *Clock* objects and just define `dt`. If neither a `dt` nor a `clock` argument is specified, the object will use the *defaultclock*. Setting `defaultclock.dt` will therefore change the `dt` of all objects that use the *defaultclock*.

Note that directly changing the `dt` attribute of an object is not allowed, neither it is possible to assign to `dt` in abstract code statements. To change `dt` between runs, change the `dt` attribute of the respective *Clock* object (which is also accessible as the `clock` attribute of each *BrianObject*). The `when` and the `order` attributes can be changed by setting the respective attributes of a *BrianObject*.

During a single time step, objects are updated according to their `when` argument's position in the schedule. This schedule is determined by *Network.schedule* which is a list of strings, determining "execution slots" and their order. It defaults to: `['start', 'groups', 'thresholds', 'synapses', 'resets', 'end']`. In addition to the names provided in the schedule, names such as `before_thresholds` or `after_synapses` can be used that are understood as slots in the respective positions. The default for the `when` attribute is a sensible value for most objects (resets will happen in the `reset` slot, etc.) but sometimes it make sense to change it, e.g. if one would like a *StateMonitor*, which by default records in the `end` slot, to record the membrane potential before a reset is applied (otherwise no threshold crossings will be observed in the membrane potential traces). Note that you can also add new slots to the schedule and refer to them in the `when` argument of an object.

Finally, if during a time step two objects fall in the same execution slot, they will be updated in ascending order according to their `order` attribute, an integer number defaulting to 0. If two objects have the same `when` and `order` attribute then they will be updated in an arbitrary but reproducible order (based on the lexicographical order of their names).

Note that objects that don't do any computation by themselves but only act as a container for other objects (e.g. a *NeuronGroup* which contains a `StateUpdater`, a `Resetter` and a `Thresholder`), don't have any value for `when`, but pass on the given values for `dt` and `order` to their containing objects.

Every new *Network* starts a simulation at time 0; *Network.t* is a read-only attribute, to go back to a previous moment in time (e.g. to do another trial of a simulation with a new noise instantiation) use the mechanism described below.

Note that while it is allowed to change the `dt` of an object between runs (e.g. to simulate/monitor an initial phase with a bigger time step than a later phase), this change has to be compatible with the internal representation of clocks as an integer value (the number of elapsed time steps). For example, you can simulate an object for 100ms with a time step of 0.1ms (i.e. for 1000 steps) and then switch to a `dt` of 0.5ms, the time will then be internally represented as 200 steps. You cannot, however, switch to a dt of 0.3ms, because 100ms are not an integer multiple of 0.3ms.

4.4.2 Progress reporting

For custom progress reporting (e.g. graphical output, writing to a file, etc.), the `report` keyword accepts a callable (i.e. a function or an object with a `__call__` method) that will be called with three parameters:

- `elapsed`: the total (real) time since the start of the run
- `completed`: the fraction of the total simulation that is completed, i.e. a value between 0 and 1
- `duration`: the total duration (in biological time) of the simulation

The function will be called every `report_period` during the simulation, but also at the beginning and end with `completed` equal to 0.0 and 1.0, respectively.

For the C++ standalone mode, the same standard options are available. It is also possible to implement custom progress reporting by directly passing the code (as a multi-line string) to the `report` argument. This code will be filled into a progress report function template, it should therefore only contain a function body. The simplest use of this might look like:

```
net.run(duration, report='std::cout << (int)(completed*100.) << "% completed" << std::endl;')
```

Examples of custom reporting

Progress printed to a file

```
from brian2.core.network import TextReport
report_file = open('report.txt', 'w')
file_reporter = TextReport(report_file)
net.run(duration, report=file_reporter)
report_file.close()
```

"Graphical" output on the console

This needs a "normal" Linux console, i.e. it might not work in an integrated console in an IDE.

Adapted from http://stackoverflow.com/questions/3160699/python-progress-bar

```
import sys

class ProgressBar(object):
```

```
    def __init__(self, toolbar_width):
        self.toolbar_width = toolbar_width
        self.ticks = 0

    def __call__(self, elapsed, complete, duration):
        if complete == 0.0:
            # setup toolbar
            sys.stdout.write("[%s]" % (" " * self.toolbar_width))
            sys.stdout.flush()
            sys.stdout.write("\b" * (self.toolbar_width + 1)) # return to start of line, after '['
        else:
            ticks_needed = int(round(complete * 40))
            if self.ticks < ticks_needed:
                sys.stdout.write("-" * (ticks_needed-self.ticks))
                sys.stdout.flush()
                self.ticks = ticks_needed
        if complete == 1.0:
            sys.stdout.write("\n")

net.run(duration, report=progress_bar, report_period=1*second)
```

4.5 Custom events

In most simulations, a *NeuronGroup* defines a threshold on its membrane potential that triggers a spike event. This event can be monitored by a *SpikeMonitor*, it is used in synaptic interactions, and in integrate-and-fire models it also leads to the execution of one or more reset statements.

Sometimes, it can be useful to define additional events, e.g. when an ion concentration in the cell crosses a certain threshold. This can be done with the `events` keyword in the *NeuronGroup* initializer:

```
group = NeuronGroup(N, '...', threshold='...', reset='...',
                    events={'custom_event': 'x > x_th'})
```

In this example, we define an event with the name `custom_event` that is triggered when the `x` variable crosses the threshold `x_th`. Such events can be recorded with an *EventMonitor*:

```
event_mon = EventMonitor(group, 'custom_event')
```

Such an *EventMonitor* can be used in the same way as a *SpikeMonitor* – in fact, creating the *SpikeMonitor* is basically identical to recording the `spike` event with an *EventMonitor*. An *EventMonitor* is not limited to record the event time/neuron index, it can also record other variables of the model:

```
event_mon = EventMonitor(group, 'custom_event', variables['var1', 'var2'])
```

If the event should trigger a series of statements (i.e. the equivalent of `reset` statements), this can be added by calling `run_on_event`:

```
group.run_on_event('custom_event', 'x=0')
```

When neurons are connected by *Synapses*, the `pre` and `post` pathways are triggered by spike events by default. It is possible to change this by providing an `on_event` keyword that either specifies which event to use for all pathways, or a specific event for each pathway (where non-specified pathways use the default `spike` event):

```
synapse_1 = Synapses(group, another_group, '...', pre='...', on_event='custom_event')
synapse_2 = Synapses(group, another_group, '...', pre='...', post='...',
                     on_event={'pre': 'custom_event'})
```

4.5.1 Scheduling

By default, custom events are checked after the spiking threshold (in the `after_thresholds` slots) and statements are executed after the reset (in the `after_resets` slots). The slot for the execution of custom event-triggered statements can be changed when it is added with the usual `when` and `order` keyword arguments (see Scheduling and custom progress reporting for details). To change the time when the condition is checked, use *`NeuronGroup.set_event_schedule()`*.

4.6 State update

In Brian, a state updater transforms a set of equations into an abstract state update code (and therefore is automatically target-independent). In general, any function (or callable object) that takes an *`Equations`* object and returns abstract code (as a string) can be used as a state updater and passed to the *`NeuronGroup`* constructor as a `method` argument.

The more common use case is to specify no state updater at all or chose one by name, see *Choice of state updaters* below.

4.6.1 Explicit state update

Explicit state update schemes can be specified in mathematical notation, using the *`ExplicitStateUpdater`* class. A state updater scheme contains of a series of statements, defining temporary variables and a final line (starting with `x_new =`), giving the updated value for the state variable. The description can make reference to `t` (the current time), `dt` (the size of the time step), `x` (value of the state variable), and `f(x, t)` (the definition of the state variable `x`, assuming `dx/dt = f(x, t)`. In addition, state updaters supporting stochastic equations additionally make use of `dW` (a normal distributed random variable with variance `dt`) and `g(x, t)`, the factor multiplied with the noise variable, assuming `dx/dt = f(x, t) + g(x, t) * xi`.

Using this notation, simple forward Euler integration is specified as:

```
x_new = x + dt * f(x, t)
```

A Runge-Kutta 2 (midpoint) method is specified as:

```
k = dt * f(x,t)
x_new = x + dt * f(x + k/2, t + dt/2)
```

When creating a new state updater using *`ExplicitStateUpdater`*, you can specify the `stochastic` keyword argument, determining whether this state updater does not support any stochastic equations (`None`, the default), stochastic equations with additive noise only (`'additive'`), or arbitrary stochastic equations (`'multiplicative'`). The provided state updaters use the Stratonovich interpretation for stochastic equations (which is the correct interpretation if the white noise source is seen as the limit of a coloured noise source with a short time constant). As a result of this, the simple Euler-Maruyama scheme (`x_new = x + dt*f(x, t) + dW*g(x, t)`) will only be used for additive noise. You can enforce the Ito interpretation, however, by simply directly passing such a state updater. For example, if you specify *`euler`* for a system with multiplicative noise it will generate a warning (because the state updater does not give correct results under the Stratonovich interpretation) but will work (and give the correct result under the Ito interpretation).

An example for a general state updater that handles arbitrary multiplicative noise (under Stratonovich interpretation) is the derivative-free Milstein method:

```
x_support = x + dt*f(x, t) + dt**.5 * g(x, t)
g_support = g(x_support, t)
k = 1/(2*dt**.5)*(g_support - g(x, t))*(dW**2)
x_new = x + dt*f(x,t) + g(x, t) * dW + k
```

Note that a single line in these descriptions is only allowed to mention `g(x, t)`, respectively `f(x, t)` only once (and you are not allowed to write, for example, `g(f(x, t), t)`). You can work around these restrictions by using intermediate steps, defining temporary variables, as in the above examples for *milstein* and *rk2*.

4.6.2 Choice of state updaters

As mentioned in the beginning, you can pass arbitrary callables to the method argument of a *NeuronGroup*, as long as this callable converts an *Equations* object into abstract code. The best way to add a new state updater, however, is to register it with brian and provide a method to determine whether it is appropriate for a given set of equations. This way, it can be automatically chosen when no method is specified and it can be referred to with a name (i.e. you can pass a string like `'euler'` to the method argument instead of importing *euler* and passing a reference to the object itself).

If you create a new state updater using the *ExplicitStateUpdater* class, you have to specify what kind of stochastic equations it supports. The keyword argument `stochastic` takes the values `None` (no stochastic equation support, the default), `'additive'` (support for stochastic equations with additive noise), `'multiplicative'` (support for arbitrary stochastic equations).

After creating the state updater, it has to be registered with *StateUpdateMethod*:

```
new_state_updater = ExplicitStateUpdater('...', stochastic='additive')
StateUpdateMethod.register('mymethod', new_state_updater)
```

The preferred way to do write new general state updaters (i.e. state updaters that cannot be described using the explicit syntax described above) is to extend the *StateUpdateMethod* class (but this is not strictly necessary, all that is needed is an object that implements a `can_integrate` and a `__call__` method). The new class's `can_integrate` method gets an *Equations* object, a `namespace` dictionary for the external variables/functions and a `specifier` dictionary for the internal state variables. It has to return `True` or `False`, depending on whether it can integrate the given equations. The method would typically make use of *Equations.is_stochastic* or *Equations.stochastic_type*, check whether any external functions are used, etc.. Finally, the state updater has to be registered with *StateUpdateMethod* as shown above.

4.6.3 Implicit state updates

Note: All of the following is just here for future reference, it's not implemented yet.

Implicit schemes often use Newton-Raphson or fixed point iterations. These can also be defined by mathematical statements, but the number of iterations is dynamic and therefore not easily vectorised. However, this might not be a big issue in C, GPU or even with Numba.

Backward Euler

Backward Euler is defined as follows:

```
x(t+dt)=x(t)+dt*f(x(t+dt),t+dt)
```

This is not a executable statement because the RHS depends on the future. A simple way is to perform fixed point iterations:

```
x(t+dt)=x(t)
x(t+dt)=x(t)+dt*dx=f(x(t+dt),t+dt)     until increment<tolerance
```

This includes a loop with a different number of iterations depending on the neuron.

4.7 How Brian works

In this section we will briefly cover some of the internals of how Brian works. This is included here to understand the general process that Brian goes through in running a simulation, but it will not be sufficient to understand the source code of Brian itself or to extend it to do new things. For a more detailed view of this, see the documentation in the Developer's guide

The user-visible part of Brian consists of a number of objects such as *NeuronGroup*, *Synapses*, *Network*, etc. These are all written in pure Python and essentially work to translate the user specified model into the computational engine. The end state of this translation is a collection of short blocks of code operating on a namespace, which are called in a sequence by the *Network*. Examples of these short blocks of code are the "state updaters" which perform numerical integration, or the synaptic propagation step. The namespaces consist of a mapping from names to values, where the possible values can be scalar values, fixed-length or dynamically sized arrays, and functions.

4.7.1 Syntax layer

The syntax layer consists of everything that is independent of the way the final simulation is computed (i.e. the language and device it is running on). This includes things like *NeuronGroup*, *Synapses*, *Network*, *Equations*, etc.

The user-visible part of this is documented fully in the User's guide and the Advanced guide. In particular, things such as the analysis of equations and assignment of numerical integrators. The end result of this process, which is passed to the computational engine, is a specification of the simulation consisting of the following data:

- A collection of variables which are scalar values, fixed-length arrays, dynamically sized arrays, and functions. These are handled by `Variable` objects detailed in Variables and indices. Examples: each state variable of a *NeuronGroup* is assigned an `ArrayVariable`; the list of spike indices stored by a *SpikeMonitor* is assigned a `DynamicArrayVariable`; etc.
- A collection of code blocks specified via an "abstract code block" and a template name. The "abstract code block" is a sequence of statements such as `v = vr` which are to be executed. In the case that say, `v` and `vr` are arrays, then the statement is to be executed for each element of the array. These abstract code blocks are either given directly by the user (in the case of neuron threshold and reset, and synaptic pre and post codes), or generated from differential equations combined with a numerical integrator. The template name is one of a small set (around 20 total) which give additional context. For example, the code block `a = b` when considered as part of a "state update" means execute that for each neuron index. In the context of a reset statement, it means execute it for each neuron index of a neuron that has spiked. Internally, these templates need to be implemented for each target language/device, but there are relatively few of them.
- The order of execution of these code blocks, as defined by the *Network*.

4.7.2 Computational engine

The computational engine covers everything from generating to running code in a particular language or on a particular device. It starts with the abstract definition of the simulation resulting from the syntax layer described above.

The computational engine is described by a `Device` object. This is used for allocating memory, generating and running code. There are two types of device, "runtime" and "standalone". In runtime mode, everything is managed by Python, even if individual code blocks are in a different language. Memory is managed using numpy arrays (which can be passed as pointers to use in other languages). In standalone mode, the output of the process (after calling `Device.build`) is a complete source code project that handles everything, including memory management, and is independent of Python.

For both types of device, one of the key steps that works in the same way is code generation, the creation of a compilable and runnable block of code from an abstract code block and a collection of variables. This happens in two

stages: first of all, the abstract code block is converted into a code snippet, which is a syntactically correct block of code in the target language, but not one that can run on its own (it doesn't handle accessing the variables from memory, etc.). This code snippet typically represents the inner loop code. This step is handled by a *CodeGenerator* object. In some cases it will involve a syntax translation (e.g. the Python syntax `x**y` in C++ should be `pow(x, y)`). The next step is to insert this code snippet into a template to form a compilable code block. This code block is then passed to a runtime *CodeObject*. In the case of standalone mode, this doesn't do anything, but for runtime devices it handles compiling the code and then running the compiled code block in the given namespace.

4.8 Interfacing with external code

Some neural simulations benefit from a direct connections to external libraries, e.g. to support real-time input from a sensor (but note that Brian currently does not offer facilities to assure real-time processing) or to perform complex calculations during a simulation run.

If the external library is written in Python (or is a library with Python bindings), then the connection can be made either using the mechanism for *User-provided functions*, or using a *network operation*.

In case of C/C++ libraries, only the *User-provided functions* mechanism can be used. On the other hand, such simulations can use the same user-provided C++ code to run both with the runtime `weave` target and with the *C++ standalone* mode. In addition to that code, one generally needs to include additional header files and use compiler/linker options to interface with the external code. For this, several preferences can be used that will be taken into account for `weave`, `cython` and the `cpp_standalone` device. These preferences are mostly equivalent to the respective keyword arguments for Python's `distutils.core.Extension` class, see the documentation of the *cpp_prefs* module for more details.

CHAPTER 5

Examples

5.1 Example: COBAHH

```
# coding: latin-1
"""
This is an implementation of a benchmark described
in the following review paper:

Simulation of networks of spiking neurons: A review of tools and strategies (2006).
Brette, Rudolph, Carnevale, Hines, Beeman, Bower, Diesmann, Goodman, Harris, Zirpe,
Natschl?ger, Pecevski, Ermentrout, Djurfeldt, Lansner, Rochel, Vibert, Alvarez, Muller,
Davison, El Boustani and Destexhe.
Journal of Computational Neuroscience

Benchmark 3: random network of HH neurons with exponential synaptic conductances

Clock-driven implementation
(no spike time interpolation)

R. Brette - Dec 2007
"""

from brian2 import *

# Parameters
area = 20000*umetre**2
Cm = (1*ufarad*cm**-2) * area
gl = (5e-5*siemens*cm**-2) * area

El = -60*mV
EK = -90*mV
ENa = 50*mV
g_na = (100*msiemens*cm**-2) * area
g_kd = (30*msiemens*cm**-2) * area
VT = -63*mV
# Time constants
taue = 5*ms
taui = 10*ms
# Reversal potentials
Ee = 0*mV
Ei = -80*mV
we = 6*nS  # excitatory synaptic weight
```

```
wi = 67*nS  # inhibitory synaptic weight

# The model
eqs = Equations('''
dv/dt = (gl*(El-v)+ge*(Ee-v)+gi*(Ei-v)-
         g_na*(m*m*m)*h*(v-ENa)-
         g_kd*(n*n*n*n)*(v-EK))/Cm : volt
dm/dt = alpha_m*(1-m)-beta_m*m : 1
dn/dt = alpha_n*(1-n)-beta_n*n : 1
dh/dt = alpha_h*(1-h)-beta_h*h : 1
dge/dt = -ge*(1./taue) : siemens
dgi/dt = -gi*(1./taui) : siemens
alpha_m = 0.32*(mV**-1)*(13*mV-v+VT)/
          (exp((13*mV-v+VT)/(4*mV))-1.)/ms : Hz
beta_m = 0.28*(mV**-1)*(v-VT-40*mV)/
         (exp((v-VT-40*mV)/(5*mV))-1)/ms : Hz
alpha_h = 0.128*exp((17*mV-v+VT)/(18*mV))/ms : Hz
beta_h = 4./(1+exp((40*mV-v+VT)/(5*mV)))/ms : Hz
alpha_n = 0.032*(mV**-1)*(15*mV-v+VT)/
          (exp((15*mV-v+VT)/(5*mV))-1.)/ms : Hz
beta_n = .5*exp((10*mV-v+VT)/(40*mV))/ms : Hz
''')

P = NeuronGroup(4000, model=eqs, threshold='v>-20*mV', refractory=3*ms,
                method='exponential_euler')
Pe = P[:3200]
Pi = P[3200:]
Ce = Synapses(Pe, P, pre='ge+=we', connect='rand()<0.02')
Ci = Synapses(Pi, P, pre='gi+=wi', connect='rand()<0.02')

# Initialization
P.v = 'El + (randn() * 5 - 5)*mV'
P.ge = '(randn() * 1.5 + 4) * 10.*nS'
P.gi = '(randn() * 12 + 20) * 10.*nS'

# Record a few traces
trace = StateMonitor(P, 'v', record=[1, 10, 100])
run(1 * second, report='text')
plot(trace.t/ms, trace[1].v/mV)
plot(trace.t/ms, trace[10].v/mV)
plot(trace.t/ms, trace[100].v/mV)
xlabel('t (ms)')
ylabel('v (mV)')
show()
```

5.2 Example: CUBA

This is a Brian script implementing a benchmark described in the following review paper:

Simulation of networks of spiking neurons: A review of tools and strategies (2007). Brette, Rudolph, Carnevale, Hines, Beeman, Bower, Diesmann, Goodman, Harris, Zirpe, Natschlager, Pecevski, Ermentrout, Djurfeldt, Lansner, Rochel, Vibert, Alvarez, Muller, Davison, El Boustani and Destexhe. Journal of Computational Neuroscience 23(3):349-98

Benchmark 2: random network of integrate-and-fire neurons with exponential synaptic currents.

Clock-driven implementation with exact subthreshold integration (but spike times are aligned to the grid).

```
from brian2 import *

taum = 20*ms
taue = 5*ms
taui = 10*ms
Vt = -50*mV
Vr = -60*mV
El = -49*mV

eqs = '''
dv/dt  = (ge+gi-(v-El))/taum : volt (unless refractory)
dge/dt = -ge/taue : volt (unless refractory)
dgi/dt = -gi/taui : volt (unless refractory)
'''

P = NeuronGroup(4000, eqs, threshold='v>Vt', reset='v = Vr', refractory=5*ms)
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, P, pre='ge += we')
Ci = Synapses(P, P, pre='gi += wi')
Ce.connect('i<3200', p=0.02)
Ci.connect('i>=3200', p=0.02)

s_mon = SpikeMonitor(P)

run(1 * second)

plot(s_mon.t/ms, s_mon.i, '.k')
xlabel('Time (ms)')
ylabel('Neuron index')
show()
```

5.3 Example: IF_curve_Hodgkin_Huxley

Input-Frequency curve of a HH model. Network: 100 unconnected Hodgin-Huxley neurons with an input current I. The input is set differently for each neuron.

This simulation should use exponential Euler integration.

```
from brian2 import *

num_neurons = 100
duration = 2*second

# Parameters
area = 20000*umetre**2
Cm = 1*ufarad*cm**-2 * area
gl = 5e-5*siemens*cm**-2 * area
El = -65*mV
EK = -90*mV
ENa = 50*mV
g_na = 100*msiemens*cm**-2 * area
```

```
g_kd = 30*msiemens*cm**-2 * area
VT = -63*mV

# The model
eqs = Equations('''
dv/dt = (gl*(El-v) - g_na*(m*m*m)*h*(v-ENa) - g_kd*(n*n*n*n)*(v-EK) + I)/Cm : volt
dm/dt = 0.32*(mV**-1)*(13.*mV-v+VT)/
    (exp((13.*mV-v+VT)/(4.*mV))-1.)/ms*(1-m)-0.28*(mV**-1)*(v-VT-40.*mV)/
    (exp((v-VT-40.*mV)/(5.*mV))-1.)/ms*m : 1
dn/dt = 0.032*(mV**-1)*(15.*mV-v+VT)/
    (exp((15.*mV-v+VT)/(5.*mV))-1.)/ms*(1.-n)-.5*exp((10.*mV-v+VT)/(40.*mV))/ms*n : 1
dh/dt = 0.128*exp((17.*mV-v+VT)/(18.*mV))/ms*(1.-h)-4./(1+exp((40.*mV-v+VT)/(5.*mV)))/ms*h : 1
I : amp
''')
# Threshold and refractoriness are only used for spike counting
group = NeuronGroup(num_neurons, eqs,
                    threshold='v > -40*mV',
                    refractory='v > -40*mV',
                    method='exponential_euler')
group.v = El
group.I = '0.7*nA * i / num_neurons'

monitor = SpikeMonitor(group)

run(duration)

plot(group.I/nA, monitor.count / duration)
xlabel('I (nA)')
ylabel('Firing rate (sp/s)')
show()
```

5.4 Example: IF_curve_LIF

Input-Frequency curve of a IF model. Network: 1000 unconnected integrate-and-fire neurons (leaky IF) with an input parameter v0. The input is set differently for each neuron.

```
from brian2 import *

n = 1000
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)
group.v = 0*mV
group.v0 = '20*mV * i / (n-1)'

monitor = SpikeMonitor(group)

run(duration)
plot(group.v0/mV, monitor.count / duration)
xlabel('v0 (mV)')
ylabel('Firing rate (sp/s)')
```

```
show()
```

5.5 Example: adaptive_threshold

```
from brian2 import *
'''
A model with adaptive threshold (increases with each spike)
'''
#set_device('cpp_standalone_simple')
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')
IF.vt = 10*mV
PG = PoissonGroup(1, 500 * Hz)

C = Synapses(PG, IF, pre='v += 3*mV', connect=True)

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, report='text')
# print M_crossings.codeobj.code

subplot(1, 2, 1)
plot(Mv.t / ms, Mv[0].v / mV)
plot(Mvt.t / ms, Mvt[0].vt / mV)
ylabel('v (mV)')
xlabel('t (ms)')
# zoom in on the first 100ms
xlim(0, 100)
subplot(1, 2, 2)
hist(M_crossings.v / mV, bins=np.arange(10, 20, 0.5))
xlabel('v at threshold crossing (mV)')
show()
```

5.6 Example: non_reliability

Reliability of spike timing. See e.g. Mainen & Sejnowski (1995) for experimental results in vitro.

Here: a constant current is injected in all trials.

```
from brian2 import *

N = 25
tau = 20*ms
```

```
sigma = .015
eqs_neurons = '''
dx/dt = (1.1 - x) / tau + sigma * (2 / tau)**.5 * xi : 1 (unless refractory)
'''
neurons = NeuronGroup(N, model=eqs_neurons, threshold='x > 1', reset='x = 0',
                      refractory=5*ms)
spikes = SpikeMonitor(neurons)

run(500*ms)
plot(spikes.t/ms, spikes.i, '.k')
xlabel('Time (ms)')
ylabel('Neuron index')
show()
```

5.7 Example: phase_locking

Phase locking of IF neurons to a periodic input.

```
from brian2 import *

tau = 20*ms
n = 100
b = 1.2 # constant current mean, the modulation varies
freq = 10*Hz

eqs = '''
dv/dt = (-v + a * sin(2 * pi * freq * t) + b) / tau : 1
a : 1
'''
neurons = NeuronGroup(n, model=eqs, threshold='v > 1', reset='v = 0')
neurons.v = 'rand()'
neurons.a = '0.05 + 0.7*i/n'
S = SpikeMonitor(neurons)
trace = StateMonitor(neurons, 'v', record=50)

run(1000*ms)
subplot(211)
plot(S.t/ms, S.i, '.k')
xlabel('Time (ms)')
ylabel('Neuron index')
subplot(212)
plot(trace.t/ms, trace.v.T)
xlabel('Time (ms)')
ylabel('v')
show()
```

5.8 Example: reliability

Reliability of spike timing. See e.g. Mainen & Sejnowski (1995) for experimental results in vitro.

```
from brian2 import *

# The common noisy input
N = 25
```

```
tau_input = 5*ms
input = NeuronGroup(1, 'dx/dt = -x / tau_input + (2 /tau_input)**.5 * xi : 1')

# The noisy neurons receiving the same input
tau = 10*ms
sigma = .015
eqs_neurons = '''
dx/dt = (0.9 + .5 * I - x) / tau + sigma * (2 / tau)**.5 * xi : 1
I : 1 (linked)
'''
neurons = NeuronGroup(N, model=eqs_neurons, threshold='x > 1',
                      reset='x = 0', refractory=5*ms)
neurons.x = 'rand()'
neurons.I = linked_var(input, 'x') # input.x is continuously fed into neurons.I
spikes = SpikeMonitor(neurons)

run(500*ms)
plt.plot(spikes.t/ms, spikes.i, '.k')
xlabel('Time (ms)')
ylabel('Neuron index')
show()
```

5.9 advanced

5.9.1 Example: opencv_movie

An example that uses a function from external C library (OpenCV in this case). Works for all C-based code generation targets (i.e. for weave and cpp_standalone device) and for numpy (using the Python bindings).

This example needs a working installation of OpenCV2 and its Python bindings. It has been tested on Ubuntu 14.04 with OpenCV 2.4.8 (libopencv-dev and python-opencv packages).

```
import os
import urllib2
import cv2  # Import OpenCV2
import cv2.cv as cv  # Import the cv subpackage, needed for some constants

from brian2 import *

defaultclock.dt = 1*ms
prefs.codegen.target = 'weave'
prefs.logging.std_redirection = False
set_device('cpp_standalone')
filename = os.path.abspath('Megamind.avi')

if not os.path.exists(filename):
    print('Downloading the example video file')
    response = urllib2.urlopen('http://docs.opencv.org/_downloads/Megamind.avi')
    data = response.read()
    with open(filename, 'wb') as f:
        f.write(data)

video = cv2.VideoCapture(filename)
width, height, frame_count = (int(video.get(cv.CV_CAP_PROP_FRAME_WIDTH)),
                              int(video.get(cv.CV_CAP_PROP_FRAME_HEIGHT)),
                              int(video.get(cv.CV_CAP_PROP_FRAME_COUNT)))
```

```
fps = 24
time_between_frames = 1*second/fps

# Links the necessary libraries
prefs.codegen.cpp.libraries += ['opencv_core',
                                'opencv_highgui']

# Includes the header files in all generated files
prefs.codegen.cpp.headers += ['<opencv2/core/core.hpp>',
                              '<opencv2/highgui/highgui.hpp>']

# Pass in values as macros
# Note that in general we could also pass in the filename this way, but to get
# the string quoting right is unfortunately quite difficult
prefs.codegen.cpp.define_macros += [('VIDEO_WIDTH', width),
                                    ('VIDEO_HEIGHT', height)]
@implementation('cpp', '''
double* get_frame(bool new_frame)
{
    // The following initializations will only be executed once
    static cv::VideoCapture source("VIDEO_FILENAME");
    static cv::Mat frame;
    static double* grayscale_frame = (double*)malloc(VIDEO_WIDTH*VIDEO_HEIGHT*sizeof(double));
    if (new_frame)
    {
        source >> frame;
        double mean_value = 0;
        for (int row=0; row<VIDEO_HEIGHT; row++)
            for (int col=0; col<VIDEO_WIDTH; col++)
            {
                const double grayscale_value = (frame.at<cv::Vec3b>(row, col)[0] +
                                                frame.at<cv::Vec3b>(row, col)[1] +
                                                frame.at<cv::Vec3b>(row, col)[2])/(3.0*128);
                mean_value += grayscale_value / (VIDEO_WIDTH * VIDEO_HEIGHT);
                grayscale_frame[row*VIDEO_WIDTH + col] = grayscale_value;
            }
        // subtract the mean
        for (int i=0; i<VIDEO_HEIGHT*VIDEO_WIDTH; i++)
            grayscale_frame[i] -= mean_value;
    }
    return grayscale_frame;
}

double video_input(const int x, const int y)
{
    // Get the current frame (or a new frame in case we are asked for the first
    // element
    double *frame = get_frame(x==0 && y==0);
    return frame[y*VIDEO_WIDTH + x];
}
'''.replace('VIDEO_FILENAME', filename))
@check_units(x=1, y=1, result=1)
def video_input(x, y):
    # we assume this will only be called in the custom operation (and not for
    # example in a reset or synaptic statement), so we don't need to do indexing
    # but we can directly return the full result
    _, frame = video.read()
    grayscale = frame.mean(axis=2)
```

```
    grayscale /= 128.  # scale everything between 0 and 2
    return grayscale.ravel() - grayscale.ravel().mean()


N = width * height
tau, tau_th = 10*ms, time_between_frames
G = NeuronGroup(N, '''dv/dt = (-v + I)/tau : 1
                      dv_th/dt = -v_th/tau_th : 1
                      row : integer (constant)
                      column : integer (constant)
                      I : 1 # input current''',
                threshold='v>v_th', reset='v=0; v_th = 3*v_th + 1.0')
G.v_th = 1
G.row = 'i/width'
G.column = 'i%width'

G.run_regularly('I = video_input(column, row)',
                dt=time_between_frames)
mon = SpikeMonitor(G)
runtime = frame_count*time_between_frames
run(runtime, report='text')
device.build(compile=True, run=True)

# Avoid going through the whole Brian2 indexing machinery too much
i, t, row, column = mon.i[:], mon.t[:], G.row[:], G.column[:]

import matplotlib.animation as animation

# TODO: Use overlapping windows
stepsize = 100*ms
def next_spikes():
    step = next_spikes.step
    if step*stepsize > runtime:
        next_spikes.step=0
        raise StopIteration()
    spikes = i[(t>=step*stepsize) & (t<(step+1)*stepsize)]
    next_spikes.step += 1
    yield column[spikes], row[spikes]
next_spikes.step = 0

fig, ax = plt.subplots()
dots, = ax.plot([], [], 'k.', markersize=2, alpha=.25)
ax.set_xlim(0, width)
ax.set_ylim(0, height)
ax.invert_yaxis()
def run(data):
    x, y = data
    dots.set_data(x, y)

ani = animation.FuncAnimation(fig, run, next_spikes, blit=False, repeat=True,
                              repeat_delay=1000)
plt.show()
```

5.9.2 Example: stochastic_odes

Demonstrate the correctness of the "derivative-free Milstein method" for multiplicative noise.

```
from brian2 import *

# setting a random seed makes all variants use exactly the same Wiener process
seed = 12347

X0 = 1
mu = 0.5/second # drift
sigma = 0.1/second #diffusion

runtime = 1*second


def simulate(method, dt):
    '''
    simulate geometrical Brownian with the given method
    '''
    np.random.seed(seed)
    G = NeuronGroup(1, 'dX/dt = (mu - 0.5*second*sigma**2)*X + X*sigma*xi*second**.5: 1',
                    dt=dt, method=method)
    G.X = X0
    mon = StateMonitor(G, 'X', record=True)
    net = Network(G, mon)
    net.run(runtime)
    return mon.t_[:], mon.X.flatten()


def exact_solution(t, dt):
    '''
    Return the exact solution for geometrical Brownian motion at the given
    time points
    '''
    # Remove units for simplicity
    my_mu = float(mu)
    my_sigma = float(sigma)
    dt = float(dt)
    t = asarray(t)

    np.random.seed(seed)
    # We are calculating the values at the *end* of a time step, as when using
    # a StateMonitor. Therefore also the Brownian motion starts not with zero
    # but with a random value.
    brownian = cumsum(sqrt(dt) * np.random.randn(len(t)))

    return (X0 * exp((my_mu - 0.5*my_sigma**2)*(t+dt) + my_sigma*brownian))

figure(1, figsize=(16, 7))
figure(2, figsize=(16, 7))

methods = ['milstein']
dts = [1*ms, 0.5*ms, 0.2*ms, 0.1*ms, 0.05*ms, 0.025*ms, 0.01*ms, 0.005*ms]

rows = floor(sqrt(len(dts)))
cols = ceil(1.0 * len(dts) / rows)
errors = dict([(method, zeros(len(dts))) for method in methods])
for dt_idx, dt in enumerate(dts):
    print 'dt: ', dt
    trajectories = {}
    # Test the numerical methods
```

```
    for method in methods:
        t, trajectories[method] = simulate(method, dt)
    # Calculate the exact solution
    exact = exact_solution(t, dt)

    for method in methods:
        # plot the trajectories
        figure(1)
        subplot(rows, cols, dt_idx+1)
        plot(t, trajectories[method], label=method, alpha=0.75)

        # determine the mean absolute error
        errors[method][dt_idx] = mean(abs(trajectories[method] - exact))
        # plot the difference to the real trajectory
        figure(2)
        subplot(rows, cols, dt_idx+1)
        plot(t, trajectories[method] - exact, label=method, alpha=0.75)

    figure(1)
    plot(t, exact, color='gray', lw=2, label='exact', alpha=0.75)
    title('dt = %s' % str(dt))
    xticks([])

figure(1)
legend(frameon=False, loc='best')
tight_layout()

figure(2)
legend(frameon=False, loc='best')
tight_layout()

figure(3)
for method in methods:
    plot(array(dts) / ms, errors[method], 'o', label=method)
legend(frameon=False, loc='best')
xscale('log')
yscale('log')
xlabel('dt (ms)')
ylabel('Mean absolute error')
tight_layout()

show()
```

5.10 compartmental

5.10.1 Example: bipolar_cell

A pseudo MSO neuron, with two dendrites and one axon (fake geometry).

```
from brian2 import *

# Morphology
morpho = Soma(30*um)
morpho.axon = Cylinder(diameter=1*um, length=300*um, n=100)
morpho.L = Cylinder(diameter=1*um, length=100*um, n=50)
```

```
morpho.R = Cylinder(diameter=1*um, length=150*um, n=50)

# Passive channels
gL = 1e-4*siemens/cm**2
EL = -70*mV
eqs='''
Im = gL * (EL - v) : amp/meter**2
I : amp (point current)
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs,
                       Cm=1*uF/cm**2, Ri=100*ohm*cm)
neuron.v = EL
neuron.I = 0*amp

# Monitors
mon_soma = StateMonitor(neuron, 'v', record=[0])
mon_L = StateMonitor(neuron.L, 'v', record=True)
mon_R = StateMonitor(neuron, 'v', record=morpho.R[75*um])

run(1*ms)
neuron.I[morpho.L[50*um]] = 0.2*nA  # injecting in the left dendrite
run(5*ms)
neuron.I = 0*amp
run(50*ms, report='text')

subplot(211)
plot(mon_L.t/ms, mon_soma[0].v/mV, 'k')
plot(mon_L.t/ms, mon_L[morpho.L[50*um]].v/mV, 'r')
plot(mon_L.t/ms, mon_R[morpho.R[75*um]].v/mV, 'b')
ylabel('v (mV)')
subplot(212)
for x in linspace(0*um, 100*um, 10, endpoint=False):
    plot(mon_L.t/ms, mon_L[morpho.L[x]].v/mV)
xlabel('Time (ms)')
ylabel('v (mV)')
show()
```

5.10.2 Example: bipolar_with_inputs

A pseudo MSO neuron, with two dendrites (fake geometry). There are synaptic inputs.

```
from brian2 import *

# Morphology
morpho = Soma(30*um)
morpho.L = Cylinder(diameter=1*um, length=100*um, n=50)
morpho.R = Cylinder(diameter=1*um, length=100*um, n=50)

# Passive channels
gL = 1e-4*siemens/cm**2
EL = -70*mV
Es = 0*mV
eqs='''
Im = gL*(EL-v) : amp/meter**2
Is = gs*(Es-v) : amp (point current)
gs : siemens
```

```
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs,
                       Cm=1*uF/cm**2, Ri=100*ohm*cm)
neuron.v = EL

# Regular inputs
stimulation = NeuronGroup(2, 'dx/dt = 300*Hz : 1', threshold='x>1', reset='x=0')
stimulation.x = [0, 0.5]  # Asynchronous

# Synapses
taus = 1*ms
w = 20*nS
S = Synapses(stimulation, neuron, model='''dg/dt = -g/taus : siemens
                                           gs_post = g : siemens (summed)''',
             pre='g += w')

S.connect(0, morpho.L[-1])
S.connect(1, morpho.R[-1])

# Monitors
mon_soma = StateMonitor(neuron, 'v', record=[0])
mon_L = StateMonitor(neuron.L, 'v', record=True)
mon_R = StateMonitor(neuron.R, 'v',
                     record=morpho.R[-1])

run(50*ms, report='text')

subplot(211)
plot(mon_L.t/ms, mon_soma[0].v/mV, 'k')
plot(mon_L.t/ms, mon_L[morpho.L[-1]].v/mV, 'r')
plot(mon_L.t/ms, mon_R[morpho.R[-1]].v/mV, 'b')
ylabel('v (mV)')
subplot(212)
for x in linspace(0*um, 100*um, 10, endpoint=False):
    plot(mon_L.t/ms, mon_L[morpho.L[x]].v/mV)
xlabel('Time (ms)')
ylabel('v (mV)')
show()
```

5.10.3 Example: bipolar_with_inputs2

A pseudo MSO neuron, with two dendrites (fake geometry). There are synaptic inputs. Second method.

```
from brian2 import *

# Morphology
morpho = Soma(30*um)
morpho.L = Cylinder(diameter=1*um, length=100*um, n=50)
morpho.R = Cylinder(diameter=1*um, length=100*um, n=50)

# Passive channels
gL = 1e-4*siemens/cm**2
EL = -70*mV
Es = 0*mV
taus = 1*ms
eqs='''
```

```
Im = gL*(EL-v) : amp/meter**2
Is = gs*(Es-v) : amp (point current)
dgs/dt = -gs/taus : siemens
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs,
                       Cm=1*uF/cm**2, Ri=100*ohm*cm)
neuron.v = EL

# Regular inputs
stimulation = NeuronGroup(2, 'dx/dt = 300*Hz : 1', threshold='x>1', reset='x=0')
stimulation.x = [0, 0.5] # Asynchronous

# Synapses
w = 20*nS
S = Synapses(stimulation, neuron, pre = 'gs += w')
S.connect(0, morpho.L[99.9*um])
S.connect(1, morpho.R[99.9*um])

# Monitors
mon_soma = StateMonitor(neuron, 'v', record=[0])
mon_L = StateMonitor(neuron.L, 'v', record=True)
mon_R = StateMonitor(neuron, 'v', record=morpho.R[99.9*um])

run(50*ms, report='text')

subplot(211)
plot(mon_L.t/ms, mon_soma[0].v/mV, 'k')
plot(mon_L.t/ms, mon_L[morpho.L[99.9*um]].v/mV, 'r')
plot(mon_L.t/ms, mon_R[morpho.R[99.9*um]].v/mV, 'b')
ylabel('v (mV)')
subplot(212)
for i in [0, 5, 10, 15, 20, 25, 30, 35, 40, 45]:
    plot(mon_L.t/ms, mon_L.v[i, :]/mV)
xlabel('Time (ms)')
ylabel('v (mV)')
show()
```

5.10.4 Example: cylinder

A short cylinder with constant injection at one end.

```
from brian2 import *

defaultclock.dt = 0.01*ms

# Morphology
diameter = 1*um
length = 300*um
Cm = 1*uF/cm**2
Ri = 150*ohm*cm
N = 200
morpho = Cylinder(diameter=diameter, length=length, n=N)

# Passive channels
gL = 1e-4*siemens/cm**2
EL = -70*mV
```

```
eqs = '''
Im = gL * (EL - v) : amp/meter**2
I : amp (point current)
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri)
neuron.v = EL

la = neuron.space_constant[0]
print "Electrotonic length", la

neuron.I[0] = 0.02*nA # injecting at the left end
run(100*ms, report='text')

plot(neuron.distance/um, neuron.v/mV, 'k')
# Theory
x = neuron.distance
ra = la * 4 * Ri / (pi * diameter**2)
theory = EL + ra * neuron.I[0] * cosh((length - x) / la) / sinh(length / la)
plot(x/um, theory/mV, 'r')
xlabel('x (um)')
ylabel('v (mV)')
show()
```

5.10.5 Example: hh_with_spikes

Hodgkin-Huxley equations (1952). Spikes are recorded along the axon, and then velocity is calculated.

```
from brian2 import *
from scipy import stats

defaultclock.dt = 0.01*ms

morpho = Cylinder(length=10*cm, diameter=2*238*um, n=1000, type='axon')

El = 10.613*mV
ENa = 115*mV
EK = -12*mV
gl = 0.3*msiemens/cm**2
gNa0 = 120*msiemens/cm**2
gK = 36*msiemens/cm**2

# Typical equations
eqs = '''
# The same equations for the whole neuron, but possibly different parameter values
# distributed transmembrane current
Im = gl * (El-v) + gNa * m**3 * h * (ENa-v) + gK * n**4 * (EK-v) : amp/meter**2
I : amp (point current) # applied current
dm/dt = alpham * (1-m) - betam * m : 1
dn/dt = alphan * (1-n) - betan * n : 1
dh/dt = alphah * (1-h) - betah * h : 1
alpham = (0.1/mV) * (-v+25*mV) / (exp((-v+25*mV) / (10*mV)) - 1)/ms : Hz
betam = 4 * exp(-v/(18*mV))/ms : Hz
alphah = 0.07 * exp(-v/(20*mV))/ms : Hz
betah = 1/(exp((-v+30*mV) / (10*mV)) + 1)/ms : Hz
alphan = (0.01/mV) * (-v+10*mV) / (exp((-v+10*mV) / (10*mV)) - 1)/ms : Hz
betan = 0.125*exp(-v/(80*mV))/ms : Hz
```

```
gNa : siemens/meter**2
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, method="exponential_euler",
                       refractory="m > 0.4", threshold="m > 0.5",
                       Cm=1*uF/cm**2, Ri=35.4*ohm*cm)
neuron.v = 0*mV
neuron.h = 1
neuron.m = 0
neuron.n = .5
neuron.I = 0*amp
neuron.gNa = gNa0
M = StateMonitor(neuron, 'v', record=True)
spikes = SpikeMonitor(neuron)

run(50*ms, report='text')
neuron.I[0] = 1*uA # current injection at one end
run(3*ms)
neuron.I = 0*amp
run(50*ms, report='text')

# Calculation of velocity
slope, intercept, r_value, p_value, std_err = stats.linregress(spikes.t/second,
                                                    neuron.distance[spikes.i]/meter)
print "Velocity = ", slope, "m/s"

subplot(211)
for i in range(10):
    plot(M.t/ms, M.v.T[:, i*100]/mV)
ylabel('v')
subplot(212)
plot(spikes.t/ms, spikes.i*neuron.length[0]/cm, '.k')
plot(spikes.t/ms, (intercept+slope*(spikes.t/second))/cm, 'r')
xlabel('Time (ms)')
ylabel('Position (cm)')
show()
```

5.10.6 Example: hodgkin_huxley_1952

Hodgkin-Huxley equations (1952).

```
from brian2 import *

morpho = Cylinder(length=10*cm, diameter=2*238*um, n=1000, type='axon')

El = 10.613*mV
ENa = 115*mV
EK = -12*mV
gl = 0.3*msiemens/cm**2
gNa0 = 120*msiemens/cm**2
gK = 36*msiemens/cm**2

# Typical equations
eqs = '''
# The same equations for the whole neuron, but possibly different parameter values
# distributed transmembrane current
Im = gl * (El-v) + gNa * m**3 * h * (ENa-v) + gK * n**4 * (EK-v) : amp/meter**2
```

```
I : amp (point current) # applied current
dm/dt = alpham * (1-m) - betam * m : 1
dn/dt = alphan * (1-n) - betan * n : 1
dh/dt = alphah * (1-h) - betah * h : 1
alpham = (0.1/mV) * (-v+25*mV) / (exp((-v+25*mV) / (10*mV)) - 1)/ms : Hz
betam = 4 * exp(-v/(18*mV))/ms : Hz
alphah = 0.07 * exp(-v/(20*mV))/ms : Hz
betah = 1/(exp((-v+30*mV) / (10*mV)) + 1)/ms : Hz
alphan = (0.01/mV) * (-v+10*mV) / (exp((-v+10*mV) / (10*mV)) - 1)/ms : Hz
betan = 0.125*exp(-v/(80*mV))/ms : Hz
gNa : siemens/meter**2
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=1*uF/cm**2,
                       Ri=35.4*ohm*cm, method="exponential_euler")
neuron.v = 0*mV
neuron.h = 1
neuron.m = 0
neuron.n = .5
neuron.I = 0
neuron.gNa = gNa0
neuron[5*cm:10*cm].gNa = 0*siemens/cm**2
M = StateMonitor(neuron, 'v', record=True)

run(50*ms, report='text')
neuron.I[0] = 1*uA  # current injection at one end
run(3*ms)
neuron.I = 0*amp
run(100*ms, report='text')
for i in xrange(75, 125, 1):
    plot(cumsum(neuron.length)/cm, i+(1./60)*M.v[:, i*5]/mV, 'k')
yticks([])
ylabel('Time [major] v (mV) [minor]')
xlabel('Position (cm)')
axis('tight')
show()
```

5.10.7 Example: infinite_cable

An (almost) infinite cable with pulse injection in the middle.

```
from brian2 import *

defaultclock.dt = 0.001*ms

# Morphology
diameter = 1*um
Cm = 1*uF/cm**2
Ri = 100*ohm*cm
N = 500
morpho = Cylinder(diameter=diameter, length=3*mm, n=N)

# Passive channels
gL = 1e-4*siemens/cm**2
EL = -70*mV
eqs = '''
Im = gL * (EL-v) : amp/meter**2
```

```
I : amp (point current)
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri)
neuron.v = EL

taum = Cm  /gL  # membrane time constant
print "Time constant", taum
la = neuron.space_constant[0]
print "Characteristic length", la

# Monitors
mon = StateMonitor(neuron, 'v', record=range(0, N/2, 20))

neuron.I[len(neuron) / 2] = 1*nA  # injecting in the middle
run(0.02*ms)
neuron.I = 0*amp
run(10*ms, report='text')

t = mon.t
plot(t/ms, mon.v.T/mV, 'k')
# Theory (incorrect near cable ends)
for i in range(0, len(neuron)/2, 20):
    x = (len(neuron)/2 - i) * morpho.length[0]*meter
    theory = (1/(la*Cm*pi*diameter) * sqrt(taum / (4*pi*(t + defaultclock.dt))) *
              exp(-(t+defaultclock.dt)/taum -
                  taum / (4*(t+defaultclock.dt))*(x/la)**2))
    theory = EL + theory * 1*nA * 0.02*ms
    plot(t/ms, theory/mV, 'r')
xlabel('Time (ms)')
ylabel('v (mV')
show()
```

5.10.8 Example: lfp

Hodgkin-Huxley equations (1952)

We calculate the extracellular field potential at various places. Shape looks approximately ok!

```
from brian2 import *

morpho = Cylinder(x=10*cm, y=0*cm, z=0*cm, diameter=2*238*um, n=1000, type='axon')

El = 10.613* mV
ENa = 115*mV
EK = -12*mV
gl = 0.3*msiemens/cm**2
gNa0 = 120*msiemens/cm**2
gK = 36*msiemens/cm**2

# Typical equations
eqs = '''
# The same equations for the whole neuron, but possibly different parameter values
# distributed transmembrane current
Im = gl * (El-v) + gNa * m**3 * h * (ENa-v) + gK * n**4 * (EK-v) : amp/meter**2
I : amp (point current) # applied current
dm/dt = alpham * (1-m) - betam * m : 1
```

```
dn/dt = alphan * (1-n) - betan * n : 1
dh/dt = alphah * (1-h) - betah * h : 1
alpham = (0.1/mV) * (-v+25*mV) / (exp((-v+25*mV) / (10*mV)) - 1)/ms : Hz
betam = 4 * exp(-v/(18*mV))/ms : Hz
alphah = 0.07 * exp(-v/(20*mV))/ms : Hz
betah = 1/(exp((-v+30*mV) / (10*mV)) + 1)/ms : Hz
alphan = (0.01/mV) * (-v+10*mV) / (exp((-v+10*mV) / (10*mV)) - 1)/ms : Hz
betan = 0.125*exp(-v/(80*mV))/ms : Hz
gNa : siemens/meter**2
previous_v : volt
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=1*uF/cm**2,
                       Ri=35.4*ohm*cm, method="exponential_euler")
neuron.v = 0*mV
neuron.h = 1
neuron.m = 0
neuron.n = .5
neuron.I = 0
neuron.gNa = gNa0
neuron[5*cm:10*cm].gNa = 0*siemens/cm**2
M = StateMonitor(neuron, 'v', record=True)

neuron.run_regularly('previous_v = v', when='start')

# LFP recorder
Ne = 5 # Number of electrodes
sigma = 0.3*siemens/meter # Resistivity of extracellular field (0.3-0.4 S/m)
lfp = NeuronGroup(Ne,model='''v : volt
                              x : meter
                              y : meter
                              z : meter''')
lfp.x = 7*cm # Off center (to be far from stimulating electrode)
lfp.y = [1*mm, 2*mm, 4*mm, 8*mm, 16*mm]
# Synapses are normally executed after state update, so v-previous_v = dv
S = Synapses(neuron,lfp,model='''w : ohm*meter**2 (constant) # Weight in the LFP calculation
                                 v_post = w*(Cm_pre*(v_pre-previous_v_pre)/dt-Im_pre) : volt (summed)
S.w = 'area_pre/(4*pi*sigma)/((x_pre-x_post)**2+(y_pre-y_post)**2+(z_pre-z_post)**2)**.5'

Mlfp = StateMonitor(lfp,'v',record=True)

run(50*ms, report='text')
neuron.I[0] = 1*uA  # current injection at one end
run(3*ms)
neuron.I = 0*amp
run(100*ms, report='text')

subplot(211)
for i in range(10):
    plot(M.t/ms,M.v[i*100]/mV)
ylabel('LFP (mV)')
subplot(212)
for i in range(5):
    plot(M.t/ms,Mlfp.v[i]/mV)
ylabel('LFP (mV)')
xlabel('Time (ms)')
show()
```

5.10.9 Example: morphotest

```
from brian2 import *

# Morphology
morpho = Soma(30*um)
morpho.L = Cylinder(diameter=1*um, length=100*um, n=5)
morpho.LL = Cylinder(diameter=1*um, length=20*um, n=2)
morpho.R = Cylinder(diameter=1*um, length=100*um, n=5)

# Passive channels
gL = 1e-4*siemens/cm**2
EL = -70*mV
eqs = '''
Im = gL * (EL-v) : amp/meter**2
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs,
                       Cm=1*uF/cm**2, Ri=100*ohm*cm)
neuron.v = arange(0, 13)*volt

print neuron.v
print neuron.L.v
print neuron.LL.v
print neuron.L.main.v
```

5.10.10 Example: rall

A cylinder plus two branches, with diameters according to Rall's formula

```
from brian2 import *

defaultclock.dt = 0.01*ms

# Passive channels
gL = 1e-4*siemens/cm**2
EL = -70*mV

# Morphology
diameter = 1*um
length = 300*um
Cm = 1*uF/cm**2
Ri = 150*ohm*cm
N = 500
rm = 1 / (gL * pi * diameter)  # membrane resistance per unit length
ra = (4 * Ri)/(pi * diameter**2)  # axial resistance per unit length
la = sqrt(rm / ra) # space length
morpho = Cylinder(diameter=diameter, length=length, n=N)
d1 = 0.5*um
L1 = 200*um
rm = 1 / (gL * pi * d1) # membrane resistance per unit length
ra = (4 * Ri) / (pi * d1**2) # axial resistance per unit length
l1 = sqrt(rm / ra) # space length
morpho.L = Cylinder(diameter=d1, length=L1, n=N)
d2 = (diameter**1.5 - d1**1.5)**(1. / 1.5)
rm = 1/(gL * pi * d2) # membrane resistance per unit length
ra = (4 * Ri) / (pi * d2**2) # axial resistance per unit length
```

```
l2 = sqrt(rm / ra) # space length
L2 = (L1 / l1) * l2
morpho.R = Cylinder(diameter=d2, length=L2, n=N)

eqs='''
Im = gL * (EL-v) : amp/meter**2
I : amp (point current)
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri)
neuron.v = EL

neuron.I[0] = 0.02*nA # injecting at the left end
run(100*ms, report='text')

plot(neuron.main.distance/um, neuron.main.v/mV, 'k')
plot(neuron.L.distance/um, neuron.L.v/mV, 'k')
plot(neuron.R.distance/um, neuron.R.v/mV, 'k')
# Theory
x = neuron.main.distance
ra = la * 4 * Ri/(pi * diameter**2)
l = length/la + L1/l1
theory = EL + ra*neuron.I[0]*cosh(l - x/la)/sinh(l)
plot(x/um, theory/mV, 'r')
x = neuron.L.distance
theory = (EL+ra*neuron.I[0]*cosh(l - neuron.main.distance[-1]/la -
                                 (x - neuron.main.distance[-1])/l1)/sinh(l))
plot(x/um, theory/mV, 'r')
x = neuron.R.distance
theory = (EL+ra*neuron.I[0]*cosh(l - neuron.main.distance[-1]/la -
                                 (x - neuron.main.distance[-1])/l2)/sinh(l))
plot(x/um, theory/mV, 'r')
xlabel('x (um)')
ylabel('v (mV)')
show()
```

5.10.11 Example: spike_initiation

Ball and stick with Na and K channels

```
from brian2 import *

defaultclock.dt = 0.025*ms

# Morphology
morpho = Soma(30*um)
morpho.axon = Cylinder(diameter=1*um, length=300*um, n=100)

# Channels
gL = 1e-4*siemens/cm**2
EL = -70*mV
ENa = 50*mV
ka = 6*mV
ki = 6*mV
va = -30*mV
vi = -50*mV
EK = -90*mV
```

```
vk = -20*mV
kk = 8*mV
eqs = '''
Im = gL*(EL-v)+gNa*m*h*(ENa-v)+gK*n*(EK-v) : amp/meter**2
dm/dt = (minf-m)/(0.3*ms) : 1 # simplified Na channel
dh/dt = (hinf-h)/(3*ms) : 1 # inactivation
dn/dt = (ninf-n)/(5*ms) : 1 # K+
minf = 1/(1+exp((va-v)/ka)) : 1
hinf = 1/(1+exp((v-vi)/ki)) : 1
ninf = 1/(1+exp((vk-v)/kk)) : 1
I : amp (point current)
gNa : siemens/meter**2
gK : siemens/meter**2
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs,
                       Cm=1*uF/cm**2, Ri=100*ohm*cm)
neuron.v = -65*mV
neuron.I = 0*amp
neuron.axon[30*um:60*um].gNa = 700*gL
neuron.axon[30*um:60*um].gK = 700*gL

# Monitors
mon=StateMonitor(neuron, 'v', record=True)

run(1*ms)
neuron.main.I = 0.15*nA
run(50*ms)
neuron.I = 0*amp
run(95*ms, report='text')

plot(mon.t/ms, mon.v[0]/mV, 'r')
plot(mon.t/ms, mon.v[20]/mV, 'g')
plot(mon.t/ms, mon.v[40]/mV, 'b')
plot(mon.t/ms, mon.v[60]/mV, 'k')
plot(mon.t/ms, mon.v[80]/mV, 'y')
xlabel('Time (ms)')
ylabel('v (mV)')
show()
```

5.11 frompapers

5.11.1 Example: Brette_2004

Phase locking in leaky integrate-and-fire model

Fig. 2A from: Brette R (2004). Dynamics of one-dimensional spiking neuron models. J Math Biol 48(1): 38-56.

This shows the phase-locking structure of a LIF driven by a sinusoidal current. When the current crosses the threshold (a<3), the model almost always phase locks (in a measure-theoretical sense).

```
from brian2 import *

# defaultclock.dt = 0.01*ms  # for a more precise picture
N = 2000
```

```
tau = 100*ms
freq = 1/tau

eqs = '''
dv/dt = (-v + a + 2*sin(2*pi*t/tau))/tau : 1
a : 1
'''

neurons = NeuronGroup(N, eqs, threshold='v>1', reset='v=0')
neurons.a = linspace(2, 4, N)

run(5*second, report='text')  # discard the first spikes (wait for convergence)
S = SpikeMonitor(neurons)
run(5*second, report='text')

i, t = S.it
plot((t % tau)/tau, neurons.a[i], '.')
xlabel('Spike phase')
ylabel('Parameter a')
show()
```

5.11.2 Example: Brette_Gerstner_2005

Adaptive exponential integrate-and-fire model. http://www.scholarpedia.org/article/Adaptive_exponential_integrate-and-fire_model

Introduced in Brette R. and Gerstner W. (2005), Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity, J. Neurophysiol. 94: 3637 - 3642.

```
from brian2 import *

# Parameters
C = 281 * pF
gL = 30 * nS
taum = C / gL
EL = -70.6 * mV
VT = -50.4 * mV
DeltaT = 2 * mV
Vcut = VT + 5 * DeltaT

# Pick an electrophysiological behaviour
tauw, a, b, Vr = 144*ms, 4*nS, 0.0805*nA, -70.6*mV # Regular spiking (as in the paper)
#tauw,a,b,Vr=20*ms,4*nS,0.5*nA,VT+5*mV # Bursting
#tauw,a,b,Vr=144*ms,2*C/(144*ms),0*nA,-70.6*mV # Fast spiking

eqs = """
dvm/dt = (gL*(EL - vm) + gL*DeltaT*exp((vm - VT)/DeltaT) + I - w)/C : volt
dw/dt = (a*(vm - EL) - w)/tauw : amp
I : amp
"""

neuron = NeuronGroup(1, model=eqs, threshold='vm>Vcut',
                     reset="vm=Vr; w+=b")
neuron.vm = EL
trace = StateMonitor(neuron, 'vm', record=0)
spikes = SpikeMonitor(neuron)
```

```
run(20 * ms)
neuron.I = 1*nA
run(100 * ms)
neuron.I = 0*nA
run(20 * ms)

# We draw nicer spikes
vm = trace[0].vm[:]
for t in spikes.t:
    i = int(t / defaultclock.dt)
    vm[i] = 20*mV

plot(trace.t / ms, vm / mV)
xlabel('time (ms)')
ylabel('membrane potential (mV)')
show()
```

5.11.3 Example: Brette_Guigon_2003

Reliability of spike timing

Adapted from Fig. 10D,E of Brette R and E Guigon (2003). Reliability of Spike Timing Is a General Property of Spiking Model Neurons. Neural Computation 15, 279-308.

This shows that reliability of spike timing is a generic property of spiking neurons, even those that are not leaky. This is a non-physiological model which can be leaky or anti-leaky depending on the sign of the input I.

All neurons receive the same fluctuating input, scaled by a parameter p that varies across neurons. This shows:

1. reproducibility of spike timing
2. robustness with respect to deterministic changes (parameter)
3. increased reproducibility in the fluctuation-driven regime (input crosses the threshold)

```
from brian2 import *

N = 500
tau = 33*ms
taux = 20*ms
sigma = 0.02

eqs_input = '''
dx/dt = -x/taux + (2/taux)**.5*xi : 1
'''

eqs = '''
dv/dt = (v*I + 1)/tau + sigma*(2/tau)**.5*xi : 1
I = 0.5 + 3*p*B : 1
B = 2./(1 + exp(-2*x)) - 1 : 1 (shared)
p : 1
x : 1 (linked)
'''

input = NeuronGroup(1, eqs_input)
neurons = NeuronGroup(N, eqs, threshold='v>1', reset='v=0')
neurons.p = '1.0*i/N'
neurons.v = 'rand()'
```

```
neurons.x = linked_var(input, 'x')

M = StateMonitor(neurons, 'B', record=0)
S = SpikeMonitor(neurons)

run(1000*ms, report='text')

subplot(211)  # The input
plot(M.t/ms, M[0].B)
xticks([])
title('shared input')
subplot(212)
plot(S.t/ms, neurons.p[S.i], '.')
plot([0, 1000], [.5, .5], 'r')
xlabel('time (ms)')
ylabel('p')
title('spiking activity')
show()
```

5.11.4 Example: Brunel_Hakim_1999

Dynamics of a network of sparsely connected inhibitory current-based integrate-and-fire neurons. Individual neurons fire irregularly at low rate but the network is in an oscillatory global activity regime where neurons are weakly synchronized.

Reference: "Fast Global Oscillations in Networks of Integrate-and-Fire Neurons with Low Firing Rates" Nicolas Brunel & Vincent Hakim Neural Computation 11, 1621-1671 (1999)

```
from brian2 import *

N = 5000
Vr = 10*mV
theta = 20*mV
tau = 20*ms
delta = 2*ms
taurefr = 2*ms
duration = .1*second
C = 1000
sparseness = float(C)/N
J = .1*mV
muext = 25*mV
sigmaext = 1*mV

eqs = """
dV/dt = (-V+muext + sigmaext * sqrt(tau) * xi)/tau : volt
"""

group = NeuronGroup(N, eqs, threshold='V>theta',
                    reset='V=Vr', refractory=taurefr)
group.V = Vr
conn = Synapses(group, group, pre='V += -J',
                connect='rand()<sparseness', delay=delta)
M = SpikeMonitor(group)
LFP = PopulationRateMonitor(group)

run(duration)
```

```
subplot(211)
plot(M.t/ms, M.i, '.')
xlim(0, duration/ms)

subplot(212)
# Bin the rates (currently not implemented in PopulationRateMonitor directly)
window = 0.4*ms
window_length = int(window/defaultclock.dt)
cumsum = numpy.cumsum(numpy.insert(LFP.rate, 0, 0))
binned_rate = (cumsum[window_length:] - cumsum[:-window_length]) / window_length
plot(LFP.t[window_length-1:]/ms, binned_rate)
xlim(0, duration/ms)

show()
```

5.11.5 Example: Diesmann_et_al_1999

Synfire chains

M. Diesmann et al. (1999). Stable propagation of synchronous spiking in cortical neural networks. Nature 402, 529-533.

```
from brian2 import *

duration = 100*ms

# Neuron model parameters
Vr = -70*mV
Vt = -55*mV
taum = 10*ms
taupsp = 0.325*ms
weight = 4.86*mV
# Neuron model
eqs = Equations('''
dV/dt = (-(V-Vr)+x)*(1./taum) : volt
dx/dt = (-x+y)*(1./taupsp) : volt
dy/dt = -y*(1./taupsp)+25.27*mV/ms+
        (39.24*mV/ms**0.5)*xi : volt
''')

# Neuron groups
n_groups = 10
group_size = 100
P = NeuronGroup(N=n_groups*group_size, model=eqs,
                threshold='V>Vt', reset='V=Vr', refractory=1*ms)

Pinput = SpikeGeneratorGroup(85, np.arange(85),
                             np.random.randn(85)*1*ms + 50*ms)
# The network structure
S = Synapses(P, P, pre='y+=weight')
S.connect('(i/group_size) == (j-group_size)/group_size')
Sinput = Synapses(Pinput, P[:group_size], pre='y+=weight', connect=True)

# Record the spikes
Mgp = SpikeMonitor(P)
Minput = SpikeMonitor(Pinput)
```

```
# Setup the network, and run it
P.V = 'Vr + rand() * (Vt - Vr)'
run(duration)

plot(Mgp.t/ms, 1.0*Mgp.i/group_size, '.')
plot([0, duration/ms], np.arange(n_groups).repeat(2).reshape(-1, 2).T, 'k-')
ylabel('group number')
yticks(np.arange(n_groups))
xlabel('time (ms)')
show()
```

5.11.6 Example: Rossant_et_al_2011bis

5.11.7 Distributed synchrony example

Fig. 14 from:

> Rossant C, Leijon S, Magnusson AK, Brette R (2011). "Sensitivity of noisy neurons to coincident inputs". Journal of Neuroscience, 31(47).

5000 independent E/I Poisson inputs are injected into a leaky integrate-and-fire neuron. Synchronous events, following an independent Poisson process at 40 Hz, are considered, where 15 E Poisson spikes are randomly shifted to be synchronous at those events. The output firing rate is then significantly higher, showing that the spike timing of less than 1% of the excitatory synapses have an important impact on the postsynaptic firing.

```
from brian2 import *

# neuron parameters
theta = -55*mV
El = -65*mV
vmean = -65*mV
taum = 5*ms
taue = 3*ms
taui = 10*ms
eqs = Equations("""
                dv/dt  = (ge+gi-(v-El))/taum : volt
                dge/dt = -ge/taue : volt
                dgi/dt = -gi/taui : volt
                """)

# input parameters
p = 15
ne = 4000
ni = 1000
lambdac = 40*Hz
lambdae = lambdai = 1*Hz

# synapse parameters
we = .5*mV/(taum/taue)**(taum/(taue-taum))
wi = (vmean-El-lambdae*ne*we*taue)/(lambdae*ni*taui)

# NeuronGroup definition
group = NeuronGroup(N=2, model=eqs, reset='v = El',
                    threshold='v>theta',
                    refractory=5*ms)
group.v = El
group.ge = group.gi = 0
```

```
# independent E/I Poisson inputs
p1 = PoissonInput(group[0:1], 'ge', N=ne, rate=lambdae, weight=we)
p2 = PoissonInput(group[0:1], 'gi', N=ni, rate=lambdai, weight=wi)

# independent E/I Poisson inputs + synchronous E events
p3 = PoissonInput(group[1:], 'ge', N=ne, rate=lambdae-(p*1.0/ne)*lambdac, weight=we)
p4 = PoissonInput(group[1:], 'gi', N=ni, rate=lambdai, weight=wi)
p5 = PoissonInput(group[1:], 'ge', N=1, rate=lambdac, weight=p*we)

# run the simulation
M = SpikeMonitor(group)
SM = StateMonitor(group, 'v', record=True)
BrianLogger.log_level_info()
run(1*second)

# plot trace and spikes
for i in [0, 1]:
    spikes = (M.t[M.i == i] - defaultclock.dt)/ms
    val = SM[i].v
    subplot(2,1,i+1)
    plot(SM.t/ms, val)
    plot(tile(spikes, (2,1)),
         vstack((val[array(spikes, dtype=int)],
                 zeros(len(spikes)))), 'b')
    title("%s: %d spikes/second" % (["uncorrelated inputs", "correlated inputs"][i],
                                    M.count[i]))
show()
```

5.11.8 Example: Rothman_Manis_2003

Cochlear neuron model of Rothman & Manis

Rothman JS, Manis PB (2003) The roles potassium currents play in regulating the electrical activity of ventral cochlear nucleus neurons. J Neurophysiol 89:3097-113.

All model types differ only by the maximal conductances.

Adapted from their Neuron implementation by Romain Brette

```
from brian2 import *

#defaultclock.dt=0.025*ms # for better precision

'''
Simulation parameters: choose current amplitude and neuron type
(from type1c, type1t, type12, type 21, type2, type2o)
'''
neuron_type = 'type1c'
Ipulse = 250*pA

C = 12*pF
Eh = -43*mV
EK = -70*mV  # -77*mV in mod file
El = -65*mV
ENa = 50*mV
nf = 0.85  # proportion of n vs p kinetics
zss = 0.5  # steady state inactivation of glt
```

```
temp = 22.  # temperature in degree celcius
q10 = 3. ** ((temp - 22) / 10.)
# hcno current (octopus cell)
frac = 0.0
qt = 4.5 ** ((temp - 33.) / 10.)

# Maximal conductances of different cell types in nS
maximal_conductances = dict(
type1c=(1000, 150, 0, 0, 0.5, 0, 2),
type1t=(1000, 80, 0, 65, 0.5, 0, 2),
type12=(1000, 150, 20, 0, 2, 0, 2),
type21=(1000, 150, 35, 0, 3.5, 0, 2),
type2=(1000, 150, 200, 0, 20, 0, 2),
type2o=(1000, 150, 600, 0, 0, 40, 2) # octopus cell
)
gnabar, gkhtbar, gkltbar, gkabar, ghbar, gbarno, gl = [x * nS for x in maximal_conductances[neuron_ty

# Classical Na channel
eqs_na = """
ina = gnabar*m**3*h*(ENa-v) : amp
dm/dt=q10*(minf-m)/mtau : 1
dh/dt=q10*(hinf-h)/htau : 1
minf = 1./(1+exp(-(vu + 38.) / 7.)) : 1
hinf = 1./(1+exp((vu + 65.) / 6.)) : 1
mtau =  ((10. / (5*exp((vu+60.) / 18.) + 36.*exp(-(vu+60.) / 25.))) + 0.04)*ms : second
htau =  ((100. / (7*exp((vu+60.) / 11.) + 10.*exp(-(vu+60.) / 25.))) + 0.6)*ms : second
"""

# KHT channel (delayed-rectifier K+)
eqs_kht = """
ikht = gkhtbar*(nf*n**2 + (1-nf)*p)*(EK-v) : amp
dn/dt=q10*(ninf-n)/ntau : 1
dp/dt=q10*(pinf-p)/ptau : 1
ninf =   (1 + exp(-(vu + 15) / 5.))**-0.5 : 1
pinf =  1. / (1 + exp(-(vu + 23) / 6.)) : 1
ntau =  ((100. / (11*exp((vu+60) / 24.) + 21*exp(-(vu+60) / 23.))) + 0.7)*ms : second
ptau = ((100. / (4*exp((vu+60) / 32.) + 5*exp(-(vu+60) / 22.))) + 5)*ms : second
"""

# Ih channel (subthreshold adaptive, non-inactivating)
eqs_ih = """
ih = ghbar*r*(Eh-v) : amp
dr/dt=q10*(rinf-r)/rtau : 1
rinf = 1. / (1+exp((vu + 76.) / 7.)) : 1
rtau = ((100000. / (237.*exp((vu+60.) / 12.) + 17.*exp(-(vu+60.) / 14.))) + 25.)*ms : second
"""

# KLT channel (low threshold K+)
eqs_klt = """
iklt = gkltbar*w**4*z*(EK-v) : amp
dw/dt=q10*(winf-w)/wtau : 1
dz/dt=q10*(zinf-z)/wtau : 1
winf = (1. / (1 + exp(-(vu + 48.) / 6.)))**0.25 : 1
zinf = zss + ((1.-zss) / (1 + exp((vu + 71.) / 10.))) : 1
wtau = ((100. / (6.*exp((vu+60.) / 6.) + 16.*exp(-(vu+60.) / 45.))) + 1.5)*ms : second
ztau = ((1000. / (exp((vu+60.) / 20.) + exp(-(vu+60.) / 8.))) + 50)*ms : second
"""
```

```
# Ka channel (transient K+)
eqs_ka = """
ika = gkabar*a**4*b*c*(EK-v): amp
da/dt=q10*(ainf-a)/atau : 1
db/dt=q10*(binf-b)/btau : 1
dc/dt=q10*(cinf-c)/ctau : 1
ainf = (1. / (1 + exp(-(vu + 31) / 6.)))**0.25 : 1
binf = 1. / (1 + exp((vu + 66) / 7.))**0.5 : 1
cinf = 1. / (1 + exp((vu + 66) / 7.))**0.5 : 1
atau =  ((100. / (7*exp((vu+60) / 14.) + 29*exp(-(vu+60) / 24.))) + 0.1)*ms : second
btau =  ((1000. / (14*exp((vu+60) / 27.) + 29*exp(-(vu+60) / 24.))) + 1)*ms : second
ctau = ((90. / (1 + exp((-66-vu) / 17.))) + 10)*ms : second
"""

# Leak
eqs_leak = """
ileak = gl*(El-v) : amp
"""

# h current for octopus cells
eqs_hcno = """
ihcno = gbarno*(h1*frac + h2*(1-frac))*(Eh-v) : amp
dh1/dt=(hinfno-h1)/tau1 : 1
dh2/dt=(hinfno-h2)/tau2 : 1
hinfno = 1./(1+exp((vu+66.)/7.)) : 1
tau1 = bet1/(qt*0.008*(1+alp1))*ms : second
tau2 = bet2/(qt*0.0029*(1+alp2))*ms : second
alp1 = exp(1e-3*3*(vu+50)*9.648e4/(8.315*(273.16+temp))) : 1
bet1 = exp(1e-3*3*0.3*(vu+50)*9.648e4/(8.315*(273.16+temp))) : 1
alp2 = exp(1e-3*3*(vu+84)*9.648e4/(8.315*(273.16+temp))) : 1
bet2 = exp(1e-3*3*0.6*(vu+84)*9.648e4/(8.315*(273.16+temp))) : 1
"""

eqs = """
dv/dt = (ileak + ina + ikht + iklt + ika + ih + ihcno + I)/C : volt
vu = v/mV : 1  # unitless v
I : amp
"""
eqs += eqs_leak + eqs_ka + eqs_na + eqs_ih + eqs_klt + eqs_kht + eqs_hcno

neuron = NeuronGroup(1, eqs, method='exponential_euler')
neuron.v = El

run(50*ms, report='text')  # Go to rest

M = StateMonitor(neuron, 'v', record=0)
neuron.I = Ipulse

run(100*ms, report='text')

plot(M.t / ms, M[0].v / mV)
xlabel('t (ms)')
ylabel('v (mV)')
show()
```

5.11.9 Example: Sturzl_et_al_2000

Adapted from Theory of Arachnid Prey Localization W. Sturzl, R. Kempter, and J. L. van Hemmen PRL 2000

Poisson inputs are replaced by integrate-and-fire neurons

Romain Brette

```
from brian2 import *

# Parameters
degree = 2 * pi / 360.
duration = 500*ms
R = 2.5*cm  # radius of scorpion
vr = 50*meter/second  # Rayleigh wave speed
phi = 144*degree  # angle of prey
A = 250*Hz
deltaI = .7*ms  # inhibitory delay
gamma = (22.5 + 45 * arange(8)) * degree  # leg angle
delay = R / vr * (1 - cos(phi - gamma))   # wave delay

# Wave (vector w)
t = arange(int(duration / defaultclock.dt) + 1) * defaultclock.dt
Dtot = 0.
w = 0.
for f in arange(150, 451)*Hz:
    D = exp(-(f/Hz - 300) ** 2 / (2 * (50 ** 2)))
    rand_angle = 2 * pi * rand()
    w += 100 * D * cos(2 * pi * f * t + rand_angle)
    Dtot += D
w = .01 * w / Dtot

# Rates from the wave
rates = TimedArray(w, dt=defaultclock.dt)

# Leg mechanical receptors
tau_legs = 1 * ms
sigma = .01
eqs_legs = """
dv/dt = (1 + rates(t - d) - v)/tau_legs + sigma*(2./tau_legs)**.5*xi:1
d : second
"""
legs = NeuronGroup(8, model=eqs_legs, threshold='v > 1', reset='v = 0',
                   refractory=1*ms, method='euler')
legs.d = delay
spikes_legs = SpikeMonitor(legs)

# Command neurons
tau = 1 * ms
taus = 1.001 * ms
wex = 7
winh = -2
eqs_neuron = '''
dv/dt = (x - v)/tau : 1
dx/dt = (y - x)/taus : 1 # alpha currents
dy/dt = -y/taus : 1
'''
neurons = NeuronGroup(8, model=eqs_neuron, threshold='v>1', reset='v=0')
synapses_ex = Synapses(legs, neurons, pre='y+=wex', connect='i==j')
```

```
synapses_inh = Synapses(legs, neurons, pre='y+=winh', delay=deltaI)
synapses_inh.connect('abs(((j - i) % N_post) - N_post/2) <= 1')
spikes = SpikeMonitor(neurons)

run(duration, report='text')

nspikes = spikes.count
x = sum(nspikes * exp(gamma * 1j))
print "Angle (deg):", arctan(imag(x) / real(x)) / degree
polar(concatenate((gamma, [gamma[0] + 2 * pi])),
      concatenate((nspikes, [nspikes[0]])) / duration / Hz)
show()
```

5.11.10 Example: Touboul_Brette_2008

Chaos in the AdEx model

Fig. 8B from: Touboul, J. and Brette, R. (2008). Dynamics and bifurcations of the adaptive exponential integrate-and-fire model. Biological Cybernetics 99(4-5):319-34.

This shows the bifurcation structure when the reset value is varied (vertical axis shows the values of w at spike times for a given a reset value Vr).

```
from brian2 import *

defaultclock.dt = 0.01*ms

C = 281*pF
gL = 30*nS
EL = -70.6*mV
VT = -50.4*mV
DeltaT = 2*mV
tauw = 40*ms
a = 4*nS
b = 0.08*nA
I = .8*nA
Vcut = VT + 5 * DeltaT  # practical threshold condition
N = 200

eqs = """
dvm/dt=(gL*(EL-vm)+gL*DeltaT*exp((vm-VT)/DeltaT)+I-w)/C : volt
dw/dt=(a*(vm-EL)-w)/tauw : amp
Vr:volt
"""

neuron = NeuronGroup(N, model=eqs, threshold='vm > Vcut',
                     reset="vm = Vr; w += b")
neuron.vm = EL
neuron.w = a * (neuron.vm - EL)
neuron.Vr = linspace(-48.3 * mV, -47.7 * mV, N)  # bifurcation parameter

init_time = 3*second
run(init_time, report='text')  # we discard the first spikes

states = StateMonitor(neuron, "w", record=True, when='start')
spikes = SpikeMonitor(neuron)
run(1 * second, report='text')
```

```
# Get the values of Vr and w for each spike
Vr = neuron.Vr[spikes.i]
w = states.w[spikes.i, int_((spikes.t-init_time)/defaultclock.dt)]

figure()
plot(Vr / mV, w / nA, '.k')
xlabel('Vr (mV)')
ylabel('w (nA)')
show()
```

5.11.11 Example: Vogels_et_al_2011

Inhibitory synaptic plasticity in a recurrent network model

(F. Zenke, 2011) (from the 2012 Brian twister)

Adapted from: Vogels, T. P., H. Sprekeler, F. Zenke, C. Clopath, and W. Gerstner. Inhibitory Plasticity Balances Excitation and Inhibition in Sensory Pathways and Memory Networks. Science (November 10, 2011).

```
from brian2 import *

# ##########################################
# Defining network model parameters
# ##########################################

NE = 8000           # Number of excitatory cells
NI = NE/4           # Number of inhibitory cells

tau_ampa = 5.0*ms   # Glutamatergic synaptic time constant
tau_gaba = 10.0*ms  # GABAergic synaptic time constant
epsilon = 0.02      # Sparseness of synaptic connections

tau_stdp = 20*ms    # STDP time constant

simtime = 10*second # Simulation time

# ##########################################
# Neuron model
# ##########################################

gl = 10.0*nsiemens   # Leak conductance
el = -60*mV          # Resting potential
er = -80*mV          # Inhibitory reversal potential
vt = -50.*mV         # Spiking threshold
memc = 200.0*pfarad  # Membrane capacitance
bgcurrent = 200*pA   # External current

eqs_neurons='''
dv/dt=(-gl*(v-el)-(g_ampa*v+g_gaba*(v-er))+bgcurrent)/memc : volt (unless refractory)
dg_ampa/dt = -g_ampa/tau_ampa : siemens
dg_gaba/dt = -g_gaba/tau_gaba : siemens
'''

# ##########################################
# Initialize neuron group
# ##########################################
```

```
neurons = NeuronGroup(NE+NI, model=eqs_neurons, threshold='v > vt',
                      reset='v=el', refractory=5*ms)
Pe = neurons[:NE]
Pi = neurons[NE:]

# ########################################
# Connecting the network
# ########################################

con_e = Synapses(Pe, neurons, pre='g_ampa += 0.3*nS', connect='rand()<epsilon')
con_ii = Synapses(Pi, Pi, pre='g_gaba += 3*nS', connect='rand()<epsilon')

# ########################################
# Inhibitory Plasticity
# ########################################

eqs_stdp_inhib = '''
w : 1
dA_pre/dt=-A_pre/tau_stdp : 1 (event-driven)
dA_post/dt=-A_post/tau_stdp : 1 (event-driven)
'''
alpha = 3*Hz*tau_stdp*2  # Target rate parameter
gmax = 100               # Maximum inhibitory weight

con_ie = Synapses(Pi, Pe, model=eqs_stdp_inhib,
                  pre='''A_pre += 1.
                         w = clip(w+(A_post-alpha)*eta, 0, gmax)
                         g_gaba += w*nS''',
                  post='''A_post += 1.
                          w = clip(w+A_pre*eta, 0, gmax)
                       ''',
                  connect='rand()<epsilon')
con_ie.w = 1e-10

# ########################################
# Setting up monitors
# ########################################

sm = SpikeMonitor(Pe)

# ########################################
# Run without plasticity
# ########################################
eta = 0          # Learning rate
run(1*second)

# ########################################
# Run with plasticity
# ########################################
eta = 1e-2           # Learning rate
run(simtime-1*second, report='text')

# ########################################
# Make plots
# ########################################

i, t = sm.it
subplot(211)
```

```
plot(t/ms, i, 'k.', ms=0.25)
title("Before")
xlabel("")
yticks([])
xlim(0.8*1e3, 1*1e3)
subplot(212)
plot(t/ms, i, 'k.', ms=0.25)
xlabel("time (ms)")
yticks([])
title("After")
xlim((simtime-0.2*second)/ms, simtime/ms)
show()
```

5.11.12 Example: Wang_Buszaki_1996

Wang-Buszaki model

J Neurosci. 1996 Oct 15;16(20):6402-13. Gamma oscillation by synaptic inhibition in a hippocampal interneuronal network model. Wang XJ, Buzsaki G.

Note that implicit integration (exponential Euler) cannot be used, and therefore simulation is rather slow.

```
from brian2 import *

defaultclock.dt = 0.01*ms

Cm = 1*uF # /cm**2
Iapp = 2*uA
gL = 0.1*msiemens
EL = -65*mV
ENa = 55*mV
EK = -90*mV
gNa = 35*msiemens
gK = 9*msiemens

eqs = '''
dv/dt = (-gNa*m**3*h*(v-ENa)-gK*n**4*(v-EK)-gL*(v-EL)+Iapp)/Cm : volt
m = alpha_m/(alpha_m+beta_m) : 1
alpha_m = -0.1/mV*(v+35*mV)/(exp(-0.1/mV*(v+35*mV))-1)/ms : Hz
beta_m = 4*exp(-(v+60*mV)/(18*mV))/ms : Hz
dh/dt = 5*(alpha_h*(1-h)-beta_h*h) : 1
alpha_h = 0.07*exp(-(v+58*mV)/(20*mV))/ms : Hz
beta_h = 1./(exp(-0.1/mV*(v+28*mV))+1)/ms : Hz
dn/dt = 5*(alpha_n*(1-n)-beta_n*n) : 1
alpha_n = -0.01/mV*(v+34*mV)/(exp(-0.1/mV*(v+34*mV))-1)/ms : Hz
beta_n = 0.125*exp(-(v+44*mV)/(80*mV))/ms : Hz
'''

neuron = NeuronGroup(1, eqs)
neuron.v = -70*mV
neuron.h = 1
M = StateMonitor(neuron, 'v', record=0)

run(100*ms, report='text')

plot(M.t/ms, M[0].v/mV)
show()
```

5.11.13 Example: kremer_et_al_2011_barrel_cortex

Late Emergence of the Whisker Direction Selectivity Map in the Rat Barrel Cortex. Kremer Y, Leger JF, Goodman DF, Brette R, Bourdieu L (2011). J Neurosci 31(29):10689-700.

Development of direction maps with pinwheels in the barrel cortex. Whiskers are deflected with random moving bars. N.B.: network construction can be long.

```
from brian2 import *
import time

# Uncomment if you have a C compiler and you are running on Python 2.x
#prefs.codegen.target = 'weave'
# Uncomment if you have a C compiler and you are running on Python 3.x
#prefs.codegen.target = 'cython'
# Set this to True to run in standalone mode (much faster, but requires a
# C++ compiler to be installed).
standalone = False
# Change this to use multiple OpenMP threads
#prefs.codegen.cpp_standalone.openmp_threads = 1

if standalone:
    set_device('cpp_standalone')

t1 = time.time()

# PARAMETERS
# Neuron numbers
M4, M23exc, M23inh = 22, 25, 12  # size of each barrel (in neurons)
N4, N23exc, N23inh = M4**2, M23exc**2, M23inh**2  # neurons per barrel
barrelarraysize = 5  # Choose 3 or 4 if memory error
Nbarrels = barrelarraysize**2
# Stimulation
stim_change_time = 5*ms
Fmax = .5/stim_change_time # maximum firing rate in layer 4 (.5 spike / stimulation)
# Neuron parameters
taum, taue, taui = 10*ms, 2*ms, 25*ms
El = -70*mV
Vt, vt_inc, tauvt = -55*mV, 2*mV, 50*ms  # adaptive threshold
# STDP
taup, taud = 5*ms, 25*ms
Ap, Ad= .05, -.04
# EPSPs/IPSPs
EPSP, IPSP = 1*mV, -1*mV
EPSC = EPSP * (taue/taum)**(taum/(taue-taum))
IPSC = IPSP * (taui/taum)**(taum/(taui-taum))
Ap, Ad = Ap*EPSC, Ad*EPSC

# Layer 4, models the input stimulus
eqs_layer4 = '''
rate = int(is_active)*clip(cos(direction - selectivity), 0, inf)*Fmax: Hz
is_active = abs((barrel_x + 0.5 - bar_x) * cos(direction) + (barrel_y + 0.5 - bar_y) * sin(direction)
barrel_x : integer # The x index of the barrel
barrel_y : integer # The y index of the barrel
selectivity : 1
# Stimulus parameters (same for all neurons)
bar_x = cos(direction)*(t - stim_start_time)/(5*ms) + stim_start_x : 1 (shared)
bar_y = sin(direction)*(t - stim_start_time)/(5*ms) + stim_start_y : 1 (shared)
direction : 1 (shared) # direction of the current stimulus
```

```
stim_start_time : second (shared) # start time of the current stimulus
stim_start_x : 1 (shared) # start position of the stimulus
stim_start_y : 1 (shared) # start position of the stimulus
'''
layer4 = NeuronGroup(N4*Nbarrels, eqs_layer4, threshold='rand() < rate*dt',
                     method='euler', name='layer4')
layer4.barrel_x = '(i / N4) % barrelarraysize + 0.5'
layer4.barrel_y = 'i / (barrelarraysize*N4) + 0.5'
layer4.selectivity = '(i%N4)/(1.0*N4)*2*pi'  # for each barrel, selectivity between 0 and 2*pi

stimradius = (11+1)*.5

# Chose a new randomly oriented bar every 60ms
runner_code = '''
direction = rand()*2*pi
stim_start_x = barrelarraysize / 2.0 - cos(direction)*stimradius
stim_start_y = barrelarraysize / 2.0 - sin(direction)*stimradius
stim_start_time = t
'''
layer4.run_regularly(runner_code, dt=60*ms, when='start')

# Layer 2/3
# Model: IF with adaptive threshold
eqs_layer23 = '''
dv/dt=(ge+gi+El-v)/taum : volt
dge/dt=-ge/taue : volt
dgi/dt=-gi/taui : volt
dvt/dt=(Vt-vt)/tauvt : volt # adaptation
barrel_idx : integer
x : 1  # in "barrel width" units
y : 1  # in "barrel width" units
'''
layer23 = NeuronGroup(Nbarrels*(N23exc+N23inh), eqs_layer23,
                      threshold='v>vt', reset='v = El; vt += vt_inc',
                      refractory=2*ms, method='euler', name='layer23')
layer23.v = El
layer23.vt = Vt

# Subgroups for excitatory and inhibitory neurons in layer 2/3
layer23exc = layer23[:Nbarrels*N23exc]
layer23inh = layer23[Nbarrels*N23exc:]

# Layer 2/3 excitatory
# The units for x and y are the width/height of a single barrel
layer23exc.x = '(i % (barrelarraysize*M23exc)) * (1.0/M23exc)'
layer23exc.y = '(i / (barrelarraysize*M23exc)) * (1.0/M23exc)'
layer23exc.barrel_idx = 'floor(x) + floor(y)*barrelarraysize'

# Layer 2/3 inhibitory
layer23inh.x = 'i % (barrelarraysize*M23inh) * (1.0/M23inh)'
layer23inh.y = 'i / (barrelarraysize*M23inh) * (1.0/M23inh)'
layer23inh.barrel_idx = 'floor(x) + floor(y)*barrelarraysize'

print "Building synapses, please wait..."
# Feedforward connections (plastic)
feedforward = Synapses(layer4, layer23exc,
                       model='''w:volt
                                dA_source/dt = -A_source/taup : volt (event-driven)
```

```
                                dA_target/dt = -A_target/taud : volt (event-driven)''',
                      pre='''ge+=w
                             A_source += Ap
                             w = clip(w+A_target, 0, EPSC)''',
                      post='''
                             A_target += Ad
                             w = clip(w+A_source, 0, EPSC)''',
                      name='feedforward')
# Connect neurons in the same barrel with 50% probability
feedforward.connect('(barrel_x_pre + barrelarraysize*barrel_y_pre) == barrel_idx_post',
                    p=0.5)
feedforward.w = EPSC*.5

print 'excitatory lateral'
# Excitatory lateral connections
recurrent_exc = Synapses(layer23exc, layer23, model='w:volt', pre='ge+=w',
                         name='recurrent_exc')
recurrent_exc.connect('j<Nbarrels*N23exc',  # excitatory->excitatory
                      p='.15*exp(-.5*(((x_pre-x_post)/.4)**2+((y_pre-y_post)/.4)**2))')
recurrent_exc.w['j<Nbarrels*N23exc'] = EPSC*.3
recurrent_exc.connect('j>=Nbarrels*N23exc',  # excitatory->inhibitory
                      p='.15*exp(-.5*(((x_pre-x_post)/.4)**2+((y_pre-y_post)/.4)**2))')
recurrent_exc.w['j>=Nbarrels*N23exc'] = EPSC


# Inhibitory lateral connections
print 'inhibitory lateral'
recurrent_inh = Synapses(layer23inh, layer23exc, pre='gi+=IPSC',
                         name='recurrent_inh')
recurrent_inh.connect('True',  # excitatory->inhibitory
                      p='exp(-.5*(((x_pre-x_post)/.2)**2+((y_pre-y_post)/.2)**2))')

if not standalone:
    print 'Total number of connections'
    print 'feedforward:', len(feedforward)
    print 'recurrent exc:', len(recurrent_exc)
    print 'recurrent inh:', len(recurrent_inh)

    t2 = time.time()
    print "Construction time: %.1fs" % (t2 - t1)

run(5*second, report='text')
device.build(directory='barrelcortex', compile=True, run=True)

# Calculate the preferred direction of each cell in layer23 by doing a
# vector average of the selectivity of the projecting layer4 cells, weighted
# by the synaptic weight.
_r = bincount(feedforward.j,
              weights=feedforward.w * cos(feedforward.selectivity_pre)/feedforward.N_incoming,
              minlength=len(layer23exc))
_i = bincount(feedforward.j,
              weights=feedforward.w * sin(feedforward.selectivity_pre)/feedforward.N_incoming,
              minlength=len(layer23exc))
selectivity_exc = (arctan2(_r, _i) % (2*pi))*180./pi


scatter(layer23.x[:Nbarrels*N23exc], layer23.y[:Nbarrels*N23exc],
        c=selectivity_exc[:Nbarrels*N23exc],
```

```
        edgecolors='none', marker='s', cmap='hsv')
vlines(np.arange(barrelarraysize), 0, barrelarraysize, 'k')
hlines(np.arange(barrelarraysize), 0, barrelarraysize, 'k')
clim(0, 360)
colorbar()
show()
```

5.12 frompapers/Brette_2012

5.12.1 Example: Fig1

Brette R (2013). Sharpness of spike initiation in neurons explained by compartmentalization. PLoS Comp Biol, doi: 10.1371/journal.pcbi.1003338.

Fig 1C-E. Somatic voltage-clamp in a ball-and-stick model with Na channels at a particular location.

```
from brian2 import *
from params import *

defaultclock.dt = 0.025*ms

# Morphology
morpho = Soma(50*um)  # chosen for a target Rm
morpho.axon = Cylinder(diameter=1*um, length=300*um, n=300)

location = 40*um # where Na channels are placed
duration = 500*ms

# Channels
eqs='''
Im = gL*(EL - v) + gclamp*(vc - v) + gNa*m*(ENa - v) : amp/meter**2
dm/dt = (minf - m) / taum: 1  # simplified Na channel
minf = 1 / (1 + exp((va - v) / ka)) : 1
gclamp : siemens/meter**2
gNa : siemens/meter**2
vc = EL + 50*mV * t/duration : volt (shared)  # Voltage clamp with a ramping voltage command
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri)
compartment = morpho.axon[location]
neuron.v = EL
neuron.gclamp[0] = gL*500
neuron.gNa[compartment] = gNa/neuron.area[compartment]

# Monitors
mon = StateMonitor(neuron, ['v', 'vc', 'm'], record=True)

run(duration, report='text')

subplot(221)
plot(mon[0].vc/mV,
     -((mon[0].vc - mon[0].v)*(neuron.gclamp[0]))*neuron.area[0]/nA, 'k')
xlabel('V (mV)')
ylabel('I (nA)')
xlim(-75, -45)
title('I-V curve')
```

```
subplot(222)
plot(mon[0].vc/mV, mon[compartment].m, 'k')
xlabel('V (mV)')
ylabel('m')
title('Activation curve (m(V))')

subplot(223)
# Number of simulation time steps for each volt increment in the voltage-clamp
dt_per_volt = len(mon.t)/(50*mV)
for v in [-64*mV, -61*mV, -58*mV, -55*mV]:
    plot(mon.v[:100 ,int(dt_per_volt*(v - EL))]/mV, 'k')
xlabel('Distance from soma (um)')
ylabel('V (mV)')
title('Voltage across axon')

subplot(224)
plot(mon[compartment].v/mV, mon[compartment].v/mV, 'k--')  # Diagonal
plot(mon[0].v/mV, mon[compartment].v/mV, 'k')
xlabel('Vs (mV)')
ylabel('Va (mV)')
show()
```

5.12.2 Example: Fig3AB

Brette R (2013). Sharpness of spike initiation in neurons explained by compartmentalization. PLoS Comp Biol, doi: 10.1371/journal.pcbi.1003338.

Fig. 3. A, B. Kink with only Nav1.6 channels

```
from brian2 import *
from params import *

codegen.target='numpy'

defaultclock.dt = 0.025*ms

# Morphology
morpho = Soma(50*um)  # chosen for a target Rm
morpho.axon = Cylinder(diameter=1*um, length=300*um, n=300)

location = 40*um  # where Na channels are placed

# Channels
eqs='''
Im = gL*(EL - v) + gNa*m*(ENa - v) : amp/meter**2
dm/dt = (minf - m) / taum : 1 # simplified Na channel
minf = 1 / (1 + exp((va - v) / ka)) : 1
gNa : siemens/meter**2
Iin : amp (point current)
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri,
                       method="exponential_euler")

compartment = morpho.axon[location]
neuron.v = EL
neuron.gNa[compartment] = gNa/neuron.area[compartment]
```

```
M = StateMonitor(neuron, ['v', 'm'], record=True)

run(20*ms, report='text')
neuron.Iin[0] = gL * 20*mV * neuron.area[0]
run(80*ms, report='text')

subplot(121)
plot(M.t/ms, M[0].v/mV, 'r')
plot(M.t/ms, M[compartment].v/mV, 'k')
plot(M.t/ms, M[compartment].m*(80+60)-80, 'k--')  # open channels
ylim(-80, 60)
xlabel('Time (ms)')
ylabel('V (mV)')
title('Voltage traces')

subplot(122)
dm = diff(M[0].v) / defaultclock.dt
dm40 = diff(M[compartment].v) / defaultclock.dt
plot((M[0].v/mV)[1:], dm/(volt/second), 'r')
plot((M[compartment].v/mV)[1:], dm40/(volt/second), 'k')
xlim(-80, 40)
xlabel('V (mV)')
ylabel('dV/dt (V/s)')
title('Phase plot')

show()
```

5.12.3 Example: Fig3CF

Brette R (2013). Sharpness of spike initiation in neurons explained by compartmentalization. PLoS Comp Biol, doi: 10.1371/journal.pcbi.1003338.

Fig. 3C-F. Kink with Nav1.6 and Nav1.2

```
from brian2 import *
from params import *

defaultclock.dt = 0.01*ms

# Morphology
morpho = Soma(50*um) # chosen for a target Rm
morpho.axon = Cylinder(diameter=1*um, length=300*um, n=300)

location16 = 40*um  # where Nav1.6 channels are placed
location12 = 15*um  # where Nav1.2 channels are placed

va2 = va + 15*mV  # depolarized Nav1.2

# Channels
duration = 100*ms
eqs='''
Im = gL * (EL - v) + gNa*m*(ENa - v) + gNa2*m2*(ENa - v) : amp/meter**2
dm/dt = (minf - m) / taum : 1  # simplified Na channel
minf = 1 / (1 + exp((va - v) / ka)) : 1
dm2/dt = (minf2 - m2) / taum : 1 # simplified Na channel, Nav1.2
minf2 = 1/(1 + exp((va2 - v) / ka)) : 1
gNa : siemens/meter**2
```

```
gNa2 : siemens/meter**2  # Nav1.2
Iin : amp (point current)
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri,
                       method="exponential_euler")
compartment16 = morpho.axon[location16]
compartment12 = morpho.axon[location12]
neuron.v = EL
neuron.gNa[compartment16] = gNa/neuron.area[compartment16]
neuron.gNa2[compartment12] = 20*gNa/neuron.area[compartment12]
# Monitors
M = StateMonitor(neuron, ['v', 'm', 'm2'], record=True)

run(20*ms, report='text')
neuron.Iin[0] = gL * 20*mV * neuron.area[0]
run(80*ms, report='text')

subplot(221)
plot(M.t/ms, M[0].v/mV, 'r')
plot(M.t/ms, M[compartment16].v/mV, 'k')
plot(M.t/ms, M[compartment16].m*(80+60)-80, 'k--')  # open channels
ylim(-80, 60)
xlabel('Time (ms)')
ylabel('V (mV)')
title('Voltage traces')

subplot(222)
plot(M[0].v/mV, M[compartment16].m,'k')
plot(M[0].v/mV, 1 / (1 + exp((va - M[0].v) / ka)), 'k--')
plot(M[0].v/mV, M[compartment12].m2, 'r')
plot(M[0].v/mV, 1 / (1 + exp((va2 - M[0].v) / ka)), 'r--')
xlim(-70, 0)
xlabel('V (mV)')
ylabel('m')
title('Activation curves')

subplot(223)
dm = diff(M[0].v) / defaultclock.dt
dm40 = diff(M[compartment16].v) / defaultclock.dt
plot((M[0].v/mV)[1:], dm/(volt/second), 'r')
plot((M[compartment16].v/mV)[1:], dm40/(volt/second), 'k')
xlim(-80, 40)
xlabel('V (mV)')
ylabel('dV/dt (V/s)')
title('Phase plot')

subplot(224)
plot((M[0].v/mV)[1:], dm/(volt/second), 'r')
plot((M[compartment16].v/mV)[1:], dm40/(volt/second), 'k')
plot((M[0].v/mV)[1:], 10 + 0*dm/(volt/second), 'k--')
xlim(-70, -40)
ylim(0, 20)
xlabel('V (mV)')
ylabel('dV/dt (V/s)')
title('Phase plot(zoom)')

show()
```

5.12.4 Example: Fig4

Brette R (2013). Sharpness of spike initiation in neurons explained by compartmentalization. PLoS Comp Biol, doi: 10.1371/journal.pcbi.1003338.

Fig. 4E-F. Spatial distribution of Na channels. Tapering axon near soma.

```
from brian2 import *
from params import *

defaultclock.dt = 0.025*ms

# Morphology
morpho = Soma(50*um) # chosen for a target Rm
# Tapering (change this for the other figure panels)
axon = Cylinder(diameter=1*um, length=300*um, n=300)
axon.diameter[0:10] = linspace(4*um, 1*um, 10)
axon.set_area()
morpho.axon = axon

# Na channels
Na_start = (25 + 10)*um
Na_end = (40 + 10)*um
linear_distribution = True  # True is F, False is E

duration = 500*ms

# Channels
eqs='''
Im = gL*(EL - v) + gclamp*(vc - v) + gNa*m*(ENa - v) : amp/meter**2
dm/dt = (minf - m) / taum: 1  # simplified Na channel
minf = 1 / (1 + exp((va - v) / ka)) : 1
gclamp : siemens/meter**2
gNa : siemens/meter**2
vc = EL + 50*mV * t / duration : volt (shared)  # Voltage clamp with a ramping voltage command
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri,
                       method="exponential_euler")
compartments = morpho.axon[Na_start:Na_end]
neuron.v = EL
neuron.gclamp[0] = gL*500

if linear_distribution:
    profile = linspace(1, 0, len(compartments))
else:
    profile = ones(len(compartments))
profile = profile / sum(profile)  # normalization

neuron.gNa[compartments] = gNa * profile / neuron.area[compartments]

# Monitors
mon = StateMonitor(neuron, 'v', record=True)

run(duration, report='text')

dt_per_volt = len(mon.t) / (50*mV)
for v in [-64*mV, -61*mV, -58*mV, -55*mV, -52*mV]:
    plot(mon.v[:100, int(dt_per_volt * (v - EL))]/mV, 'k')
```

```
xlim(0, 50+10)
ylim(-65, -25)
ylabel('V (mV)')
xlabel('Location (um)')
title('Voltage across axon')
show()
```

5.12.5 Example: Fig5A

Brette R (2013). Sharpness of spike initiation in neurons explained by compartmentalization. PLoS Comp Biol, doi: 10.1371/journal.pcbi.1003338.

Fig. 5A. Voltage trace for current injection, with an additional reset when a spike is produced.

Trick: to reset the entire neuron, we use a set of synapses from the spike initiation compartment where the threshold condition applies to all compartments, and the reset operation (v = EL) is applied there every time a spike is produced.

```
from brian2 import *
from params import *

defaultclock.dt = 0.025*ms
duration = 500*ms

# Morphology
morpho = Soma(50*um)  # chosen for a target Rm
morpho.axon = Cylinder(diameter=1*um, length=300*um, n=300)

# Input
taux = 5*ms
sigmax = 12*mV
xx0 = 7*mV

compartment = 40

# Channels
eqs = '''
Im = gL * (EL - v) + gNa * m * (ENa - v) + gLx * (xx0 + xx) : amp/meter**2
dm/dt = (minf - m) / taum : 1  # simplified Na channel
minf = 1 / (1 + exp((va - v) / ka)) : 1
gNa : siemens/meter**2
gLx : siemens/meter**2
dxx/dt = -xx / taux + sigmax * (2 / taux)**.5 *xi : volt
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri,
                       threshold='m>0.5', threshold_location=compartment,
                       refractory=5*ms)
neuron.v = EL
neuron.gLx[0] = gL
neuron.gNa[compartment] = gNa / neuron.area[compartment]

# Reset the entire neuron when there is a spike
reset = Synapses(neuron, neuron, pre='v = EL')
reset.connect('i == compartment')  # Connects the spike initiation compartment to all compartments

# Monitors
S = SpikeMonitor(neuron)
```

```
M = StateMonitor(neuron, 'v', record=0)
run(duration, report='text')

# Add spikes for display
v = M[0].v
for t in S.t:
    v[int(t / defaultclock.dt)] = 50*mV

plot(M.t/ms, v/mV, 'k')
show()
```

5.12.6 Example: params

Parameters for spike initiation simulations.

```
from brian2.units import *

# Passive parameters
EL = -75*mV
S = 7.85e-9*meter**2  # area (sphere of 50 um diameter)
Cm = 0.75*uF/cm**2
gL = 1. / (30000*ohm*cm**2)
Ri = 150*ohm*cm

# Na channels
ENa = 60*mV
ka = 6*mV
va = -40*mV
gNa = gL * 2*S
taum = 0.1*ms
```

5.13 standalone

5.13.1 Example: STDP_standalone

Spike-timing dependent plasticity. Adapted from Song, Miller and Abbott (2000) and Song and Abbott (2001).

This example is modified from `synapses_STDP.py` and writes a standalone C++ project in the directory `STDP_standalone`.

```
from brian2 import *

set_device('cpp_standalone')

N = 1000
taum = 10*ms
taupre = 20*ms
taupost = taupre
Ee = 0*mV
vt = -54*mV
vr = -60*mV
El = -74*mV
taue = 5*ms
F = 15*Hz
```

```
gmax = .01
dApre = .01
dApost = -dApre * taupre / taupost * 1.05
dApost *= gmax
dApre *= gmax

eqs_neurons = '''
dv/dt = (ge * (Ee-vr) + El - v) / taum : volt
dge/dt = -ge / taue : 1
'''

input = PoissonGroup(N, rates=F)
neurons = NeuronGroup(1, eqs_neurons, threshold='v>vt', reset='v = vr')
S = Synapses(input, neurons,
             '''w : 1
                dApre/dt = -Apre / taupre : 1 (event-driven)
                dApost/dt = -Apost / taupost : 1 (event-driven)''',
             pre='''ge += w
                    Apre += dApre
                    w = clip(w + Apost, 0, gmax)''',
             post='''Apost += dApost
                     w = clip(w + Apre, 0, gmax)''',
             connect=True,
             )
S.w = 'rand() * gmax'
mon = StateMonitor(S, 'w', record=[0, 1])
s_mon = SpikeMonitor(input)
r_mon = PopulationRateMonitor(input)

run(100*second, report='text')
device.build(directory='STDP_standalone', compile=True,
             run=True, debug=True)

subplot(311)
plot(S.w / gmax, '.k')
ylabel('Weight / gmax')
xlabel('Synapse index')
subplot(312)
hist(S.w / gmax, 20)
xlabel('Weight / gmax')
subplot(313)
plot(mon.t/second, mon.w.T/gmax)
xlabel('Time (s)')
ylabel('Weight / gmax')
tight_layout()
show()
```

5.13.2 Example: cuba_openmp

Run the `cuba.py` example with OpenMP threads.

```
from brian2 import *

set_device('cpp_standalone')
prefs.devices.cpp_standalone.openmp_threads = 4

taum = 20*ms
```

```
taue = 5*ms
taui = 10*ms
Vt = -50*mV
Vr = -60*mV
El = -49*mV

eqs = '''
dv/dt  = (ge+gi-(v-El))/taum : volt (unless refractory)
dge/dt = -ge/taue : volt (unless refractory)
dgi/dt = -gi/taui : volt (unless refractory)
'''

P = NeuronGroup(4000, eqs, threshold='v>Vt', reset='v = Vr', refractory=5*ms)
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, P, pre='ge += we')
Ci = Synapses(P, P, pre='gi += wi')
Ce.connect('i<3200', p=0.02)
Ci.connect('i>=3200', p=0.02)

s_mon = SpikeMonitor(P)

run(1 * second)
device.build(directory='CUBA', compile=True, run=True, debug=True)

plot(s_mon.t/ms, s_mon.i, '.k')
xlabel('Time (ms)')
ylabel('Neuron index')
show()
```

5.14 synapses

5.14.1 Example: STDP

Spike-timing dependent plasticity Adapted from Song, Miller and Abbott (2000) and Song and Abbott (2001)

```
from brian2 import *

N = 1000
taum = 10*ms
taupre = 20*ms
taupost = taupre
Ee = 0*mV
vt = -54*mV
vr = -60*mV
El = -74*mV
taue = 5*ms
F = 15*Hz
gmax = .01
dApre = .01
dApost = -dApre * taupre / taupost * 1.05
```

```
dApost *= gmax
dApre *= gmax

eqs_neurons = '''
dv/dt = (ge * (Ee-vr) + El - v) / taum : volt
dge/dt = -ge / taue : 1
'''

input = PoissonGroup(N, rates=F)
neurons = NeuronGroup(1, eqs_neurons, threshold='v>vt', reset='v = vr')
S = Synapses(input, neurons,
             '''w : 1
                dApre/dt = -Apre / taupre : 1 (event-driven)
                dApost/dt = -Apost / taupost : 1 (event-driven)''',
             pre='''ge += w
                    Apre += dApre
                    w = clip(w + Apost, 0, gmax)''',
             post='''Apost += dApost
                     w = clip(w + Apre, 0, gmax)''',
             connect=True,
             )
S.w = 'rand() * gmax'
mon = StateMonitor(S, 'w', record=[0, 1])
s_mon = SpikeMonitor(input)
r_mon = PopulationRateMonitor(input)

run(100*second, report='text')

subplot(311)
plot(S.w / gmax, '.k')
ylabel('Weight / gmax')
xlabel('Synapse index')
subplot(312)
hist(S.w / gmax, 20)
xlabel('Weight / gmax')
subplot(313)
plot(mon.t/second, mon.w.T/gmax)
xlabel('Time (s)')
ylabel('Weight / gmax')
tight_layout()
show()
```

5.14.2 Example: gapjunctions

Neurons with gap junctions.

```
from brian2 import *

n = 10
v0 = 1.05
tau = 10*ms

eqs = '''
dv/dt = (v0 - v + Igap) / tau : 1
Igap : 1 # gap junction current
'''
```

```
neurons = NeuronGroup(n, eqs, threshold='v > 1', reset='v = 0')
neurons.v = 'i * 1.0 / (n-1)'
trace = StateMonitor(neurons, 'v', record=[0, 5])

S = Synapses(neurons, neurons, '''
             w : 1 # gap junction conductance
             Igap_post = w * (v_pre - v_post) : 1 (summed)
             ''')
S.connect(True)
S.w = .02

run(500*ms)

plot(trace.t/ms, trace[0].v)
plot(trace.t/ms, trace[5].v)
xlabel('Time (ms)')
ylabel('v')
show()
```

5.14.3 Example: jeffress

Jeffress model, adapted with spiking neuron models. A sound source (white noise) is moving around the head. Delay differences between the two ears are used to determine the azimuth of the source. Delays are mapped to a neural place code using delay lines (each neuron receives input from both ears, with different delays).

```
from brian2 import *

defaultclock.dt = .02*ms

# Sound
sound = TimedArray(10 * randn(50000), dt=defaultclock.dt) # white noise

# Ears and sound motion around the head (constant angular speed)
sound_speed = 300*metre/second
interaural_distance = 20*cm # big head!
max_delay = interaural_distance / sound_speed
print "Maximum interaural delay:", max_delay
angular_speed = 2 * pi / second # 1 turn/second
tau_ear = 1*ms
sigma_ear = .1
eqs_ears = '''
dx/dt = (sound(t-delay)-x)/tau_ear+sigma_ear*(2./tau_ear)**.5*xi : 1 (unless refractory)
delay = distance*sin(theta) : second
distance : second # distance to the centre of the head in time units
dtheta/dt = angular_speed : radian
'''
ears = NeuronGroup(2, eqs_ears, threshold='x>1', reset='x = 0',
                   refractory=2.5*ms, name='ears')
ears.distance = [-.5 * max_delay, .5 * max_delay]
traces = StateMonitor(ears, 'delay', record=True)
# Coincidence detectors
num_neurons = 30
tau = 1*ms
sigma = .1
eqs_neurons = '''
dv/dt = -v / tau + sigma * (2 / tau)**.5 * xi : 1
'''
```

```
neurons = NeuronGroup(num_neurons, eqs_neurons, threshold='v>1',
                      reset='v = 0', name='neurons')

synapses = Synapses(ears, neurons, pre='v += .5')
synapses.connect(True)

synapses.delay['i==0'] = '(1.0*j)/(num_neurons-1)*1.1*max_delay'
synapses.delay['i==1'] = '(1.0*(num_neurons-j-1))/(num_neurons-1)*1.1*max_delay'

spikes = SpikeMonitor(neurons)

run(1000*ms)

# Plot the results
i, t = spikes.it
subplot(2, 1, 1)
plot(t/ms, i, '.')
xlabel('Time (ms)')
ylabel('Neuron index')
xlim(0, 1000)
subplot(2, 1, 2)
plot(traces.t/ms, traces.delay.T/ms)
xlabel('Time (ms)')
ylabel('Input delay (ms)')
xlim(0, 1000)
show()
```

5.14.4 Example: licklider

Spike-based adaptation of Licklider's model of pitch processing (autocorrelation with delay lines) with phase locking.

```
from brian2 import *

defaultclock.dt = .02 * ms

# Ear and sound
max_delay = 20*ms # 50 Hz
tau_ear = 1*ms
sigma_ear = 0.0
eqs_ear = '''
dx/dt = (sound-x)/tau_ear+0.1*(2./tau_ear)**.5*xi : 1 (unless refractory)
sound = 5*sin(2*pi*frequency*t)**3 : 1 # nonlinear distortion
#sound = 5*(sin(4*pi*frequency*t)+.5*sin(6*pi*frequency*t)) : 1 # missing fundamental
frequency = (200+200*t*Hz)*Hz : Hz # increasing pitch
'''
receptors = NeuronGroup(2, eqs_ear, threshold='x>1', reset='x=0',
                        refractory=2*ms)
# Coincidence detectors
min_freq = 50*Hz
max_freq = 1000*Hz
num_neurons = 300
tau = 1*ms
sigma = .1
eqs_neurons = '''
dv/dt = -v/tau+sigma*(2./tau)**.5*xi : 1
'''
```

```
neurons = NeuronGroup(num_neurons, eqs_neurons, threshold='v>1', reset='v=0')

synapses = Synapses(receptors, neurons, pre='v += 0.5', connect=True)
synapses.delay = 'i*1.0/exp(log(min_freq/Hz)+(j*1.0/(num_neurons-1))*log(max_freq/min_freq))*second'

spikes = SpikeMonitor(neurons)

run(500*ms)
plot(spikes.t/ms, spikes.i, '.k')
xlabel('Time (ms)')
ylabel('Frequency')
yticks([0, 99, 199, 299],
       array(1. / synapses.delay[1, [0, 99, 199, 299]], dtype=int))
show()
```

5.14.5 Example: nonlinear

NMDA synapses.

```
from brian2 import *

a = 1 / (10*ms)
b = 1 / (10*ms)
c = 1 / (10*ms)

input = NeuronGroup(2, 'dv/dt = 1/(10*ms) : 1', threshold='v>1', reset='v = 0')
neurons = NeuronGroup(1, """dv/dt = (g-v)/(10*ms) : 1
                            g : 1""")
S = Synapses(input,neurons,
             '''# This variable could also be called g_syn to avoid confusion
                dg_syn/dt = -a*g_syn+b*x*(1-g_syn) : 1
                g_post = g_syn : 1 (summed)
                dx/dt=-c*x : 1
                w : 1 # synaptic weight
             ''', pre='x += w') # NMDA synapses

S.connect(True)
S.w = [1., 10.]
input.v = [0., 0.5]

M = StateMonitor(S, 'g', record=True)
Mn = StateMonitor(neurons, 'g', record=0)

run(1000*ms)

subplot(2, 1, 1)
plot(M.t/ms, M.g.T)
xlabel('Time (ms)')
ylabel('g_syn')
subplot(2, 1, 2)
plot(Mn.t/ms, Mn[0].g)
ylabel('Time (ms)')
ylabel('g')
show()
```

5.14.6 Example: spatial_connections

A simple example showing how string expressions can be used to implement spatial (deterministic or stochastic) connection patterns.

```
from brian2 import *

rows, cols = 20, 20
G = NeuronGroup(rows * cols, '''x : meter
                                y : meter''')
# initialize the grid positions
grid_dist = 25*umeter
G.x = '(i / rows) * grid_dist - rows/2.0 * grid_dist'
G.y = '(i % rows) * grid_dist - cols/2.0 * grid_dist'

# Deterministic connections
distance = 120*umeter
S_deterministic = Synapses(G, G)
S_deterministic.connect('sqrt((x_pre - x_post)**2 + (y_pre - y_post)**2) < distance')

# Random connections (no self-connections)
S_stochastic = Synapses(G, G)
S_stochastic.connect('i != j',
                     p='1.5 * exp(-((x_pre-x_post)**2 + (y_pre-y_post)**2)/(2*(60*umeter)**2))')

figure(figsize=(12, 6))

# Show the connections for some neurons in different colors
for color in ['g', 'b', 'm']:
    subplot(1, 2, 1)
    neuron_idx = np.random.randint(0, rows*cols)
    plot(G.x[neuron_idx] / umeter, G.y[neuron_idx] / umeter, 'o', mec=color,
             mfc='none')
    plot(G.x[S_deterministic.j[neuron_idx, :]] / umeter,
             G.y[S_deterministic.j[neuron_idx, :]] / umeter, color + '.')
    subplot(1, 2, 2)
    plot(G.x[neuron_idx] / umeter, G.y[neuron_idx] / umeter, 'o', mec=color,
             mfc='none')
    plot(G.x[S_stochastic.j[neuron_idx, :]] / umeter,
             G.y[S_stochastic.j[neuron_idx, :]] / umeter, color + '.')

for idx, t in enumerate(['determininstic connections',
                         'random connections']):
    subplot(1, 2, idx + 1)
    xlim((-rows/2.0 * grid_dist) / umeter, (rows/2.0 * grid_dist) / umeter)
    ylim((-cols/2.0 * grid_dist) / umeter, (cols/2.0 * grid_dist) / umeter)
    title(t)
    xlabel('x')
    ylabel('y', rotation='horizontal')
    axis('equal')

tight_layout()
show()
```

5.14.7 Example: state_variables

Set state variable values with a string (using code generation).

```
from brian2 import *

G = NeuronGroup(100, 'v:volt', threshold='v>-50*mV')
G.v = '(sin(2*pi*i/N) - 70 + 0.25*randn()) * mV'
S = Synapses(G, G, 'w : volt', pre='v += w')
S.connect('True')

space_constant = 200.0
S.w['i > j'] = 'exp(-(i - j)**2/space_constant) * mV'

# Generate a matrix for display
w_matrix = np.zeros((len(G), len(G)))
w_matrix[S.i[:], S.j[:]] = S.w[:]

subplot(1, 2, 1)
plot(G.v[:] / mV)
xlabel('Neuron index')
ylabel('v')
subplot(1, 2, 2)
imshow(w_matrix)
xlabel('i')
ylabel('j')
title('Synaptic weight')
show()
```

5.14.8 Example: synapses

A simple example of using *Synapses*.

```
from brian2 import *

G1 = NeuronGroup(10, 'dv/dt = -v / (10*ms) : 1',
                 threshold='v > 1', reset='v=0.')
G1.v = 1.2
G2 = NeuronGroup(10, 'dv/dt = -v / (10*ms) : 1',
                 threshold='v > 1', reset='v=0')

syn = Synapses(G1, G2, 'dw/dt = -w / (50*ms): 1', pre='v += w')

syn.connect('i == j', p=0.75)

# Set the delays
syn.delay = '1*ms + i*ms + 0.25*ms * randn()'
# Set the initial values of the synaptic variable
syn.w = 1

mon = StateMonitor(G2, 'v', record=True)
run(20*ms)
plot(mon.t/ms, mon.v.T)
xlabel('Time (ms)')
ylabel('v')
show()
```

CHAPTER 6

brian2 package

Brian 2.0

6.1 `hears` module

This is only a temporary bridge for using Brian 1 hears with Brian 2.

This will be removed as soon as brian2hears is working.

NOTES:

- Slicing sounds with Brian 2 units doesn't work, you need to either use Brian 1 units or replace calls to `sound[:20*ms]` with `sound.slice(None, 20*ms)`, etc.

TODO: handle properties (e.g. sound.duration)

Not working examples:

- time_varying_filter1 (care with units)

Exported members: *convert_unit_b1_to_b2*, *convert_unit_b2_to_b1*

Classes

BridgeSound	We add a new method slice because slicing with units can't work with Brian 2 units.

6.1.1 BridgeSound class

(*Shortest import*: `from brian2.hears import BridgeSound`)

class `brian2.hears.`**`BridgeSound`**

Bases: `brian2.hears.new_class`

We add a new method slice because slicing with units can't work with Brian 2 units.

Methods

slice(*args)	

Details

slice(**args*)

FilterbankGroup(filterbank, targetvar, ...)	**Methods**

6.1.2 FilterbankGroup class

(*Shortest import*: `from brian2.hears import FilterbankGroup`)

class `brian2.hears.`**FilterbankGroup**(*filterbank*, *targetvar*, **args*, ***kwds*)
Bases: *brian2.groups.neurongroup.NeuronGroup*

Methods

reinit()

Details

reinit()

Sound	alias of *BridgeSound*

6.1.3 Sound class

(*Shortest import*: `from brian2.hears import Sound`)

`brian2.hears.`**Sound**
alias of *BridgeSound*

WrappedSound	alias of `new_class`

6.1.4 WrappedSound class

(*Shortest import*: `from brian2.hears import WrappedSound`)

`brian2.hears.`**WrappedSound**
alias of `new_class`

Functions

convert_unit_b1_to_b2(val)

6.1.5 convert_unit_b1_to_b2 function

(*Shortest import*: `from brian2.hears import convert_unit_b1_to_b2`)

`brian2.hears.`**convert_unit_b1_to_b2**(*val*)

convert_unit_b2_to_b1(val)	

6.1.6 convert_unit_b2_to_b1 function

(*Shortest import*: `from brian2.hears import convert_unit_b2_to_b1`)

`brian2.hears.`**`convert_unit_b2_to_b1`**`(`*val*`)`

modify_arg(arg)	Modify arguments to make them compatible with Brian 1.

6.1.7 modify_arg function

(*Shortest import*: `from brian2.hears import modify_arg`)

`brian2.hears.`**`modify_arg`**`(`*arg*`)`

Modify arguments to make them compatible with Brian 1.

- Arrays of units are replaced with straight arrays
- Single values are replaced with Brian 1 equivalents
- Slices are handled so we can use e.g. sound[:20*ms]

The second part was necessary because some functions/classes test if an object is an array or not to see if it is a sequence, but because brian2.Quantity derives from ndarray this was causing problems.

wrap_units(f)	Wrap a function to convert units into a form that Brian 1 can handle.

6.1.8 wrap_units function

(*Shortest import*: `from brian2.hears import wrap_units`)

`brian2.hears.`**`wrap_units`**`(`*f*`)`

Wrap a function to convert units into a form that Brian 1 can handle. Also, check the output argument, if it is a `b1h.Sound` wrap it.

wrap_units_class(_C)	Wrap a class to convert units into a form that Brian 1 can handle in all methods

6.1.9 wrap_units_class function

(*Shortest import*: `from brian2.hears import wrap_units_class`)

`brian2.hears.`**`wrap_units_class`**`(`*_C*`)`

Wrap a class to convert units into a form that Brian 1 can handle in all methods

wrap_units_property(p)	

6.1.10 wrap_units_property function

(*Shortest import*: `from brian2.hears import wrap_units_property`)

`brian2.hears.`**`wrap_units_property`**`(`*p*`)`

6.2 numpy_ module

A dummy package to allow importing numpy and the unit-aware replacements of numpy functions without having to know which functions are overwritten.

This can be used for example as `import brian2.numpy_ as np`

Exported members: `add_newdocs`, `ModuleDeprecationWarning`, `__version__`, `pkgload()`, `PackageLoader`, `show_config()`, `char`, `rec`, `memmap`, `newaxis`, `ndarray`, `flatiter`, `nditer`, `nested_iters`, `ufunc`, `arange`, `array`, `zeros`, `count_nonzero`, `empty`, `broadcast`, `dtype`, `fromstring`, `fromfile`, `frombuffer` ... (590 more members)

6.3 only module

A dummy package to allow wildcard import from brian2 without also importing the pylab (numpy + matplotlib) namespace.

Usage: `from brian2.only import *`

Functions

restore_initial_state()	Restores internal Brian variables to the state they are in when Brian is imported

6.3.1 restore_initial_state function

(*Shortest import*: `from brian2 import restore_initial_state`)

brian2.only.**restore_initial_state**()

Restores internal Brian variables to the state they are in when Brian is imported

Resets `defaultclock.dt = 0.1*ms`, `BrianGlobalPreferences._restore` preferences, and set *BrianObject._scope_current_key* back to 0.

6.4 Subpackages

6.4.1 codegen package

Package providing the code generation framework.

_prefs module

Module declaring general code generation preferences.

Preferences

Code generation preferences `codegen.loop_invariant_optimisations` = `True`

Whether to pull out scalar expressions out of the statements, so that they are only evaluated once instead of once for every neuron/synapse/... Can be switched off, e.g. because it complicates the code (and the same optimisation is already performed by the compiler) or because the code generation target does not deal well with it. Defaults to `True`.

`codegen.string_expression_target = 'numpy'`

Default target for the evaluation of string expressions (e.g. when indexing state variables). Should normally not be changed from the default numpy target, because the overhead of compiling code is not worth the speed gain for simple expressions.

Accepts the same arguments as *codegen.target*, except for `'auto'`

`codegen.target = 'auto'`

Default target for code generation.

Can be a string, in which case it should be one of:

- `'auto'` the default, automatically chose the best code generation target available.
- `'weave'` uses `scipy.weave` to generate and compile C++ code, should work anywhere where `gcc` is installed and available at the command line.
- `'cython'`, uses the Cython package to generate C++ code. Needs a working installation of Cython and a C++ compiler.
- `'numpy'` works on all platforms and doesn't need a C compiler but is often less efficient.

Or it can be a `CodeObject` class.

`codeobject` module

Module providing the base *CodeObject* and related functions.

Exported members: *CodeObject*, `CodeObjectUpdater`

Classes

CodeObject(owner, code, variables, ...[, name])	Executable code object.

CodeObject class

(*Shortest import*: `from brian2 import CodeObject`)

class `brian2.codegen.codeobject.`**`CodeObject`**(*owner*, *code*, *variables*, *variable_indices*, *template_name*, *template_source*, *name='codeobject*'*)

Bases: *brian2.core.names.Nameable*

Executable code object.

The `code` can either be a string or a *brian2.codegen.templates.MultiTemplate*.

After initialisation, the code is compiled with the given namespace using `code.compile(namespace)`.

Calling `code(key1=val1, key2=val2)` executes the code with the given variables inserted into the namespace.

Attributes

class_name	A short name for this type of *CodeObject*
generator_class	The *CodeGenerator* class used by this *CodeObject*

Methods

__call__(**kwds)	
compile()	
is_available()	Whether this target for code generation is available.
run()	Runs the code in the namespace.
update_namespace()	Update the namespace for this timestep.

Details

class_name
A short name for this type of *CodeObject*

generator_class
The *CodeGenerator* class used by this *CodeObject*

__call__(***kwds*)

compile()

classmethod is_available()
Whether this target for code generation is available. Should use a minimal example to check whether code generation works in general.

run()
Runs the code in the namespace.

Returns **return_value** : dict

A dictionary with the keys corresponding to the `output_variables` defined during the call of `CodeGenerator.code_object`.

update_namespace()
Update the namespace for this timestep. Should only deal with variables where *the reference* changes every timestep, i.e. where the current reference in `namespace` is not correct.

Functions

check_code_units(code, group[, user_code, ...])	Check statements for correct units.

check_code_units function

(*Shortest import*: `from brian2.codegen.codeobject import check_code_units`)

brian2.codegen.codeobject.**check_code_units**(*code*, *group*, *user_code=None*, *additional_variables=None*, *level=0*, *run_namespace=None*)

Check statements for correct units.

Parameters **code** : str

The series of statements to check

group : *Group*

The context for the code execution

user_code : str, optional

The code that was provided by the user. Used to determine whether to emit warnings and for better error messages. If not specified, assumed to be equal to `code`.

additional_variables : dict-like, optional

A mapping of names to `Variable` objects, used in addition to the variables saved in `self.group`.

level : int, optional

How far to go up in the stack to find the calling frame.

run_namespace : dict-like, optional

An additional namespace, as provided to `Group.before_run()`

Raises

`DimensionMismatchError` If `code` has unit mismatches

create_runner_codeobj(group, code, template_name)	Create a *CodeObject* for the execution of code in the context of a

create_runner_codeobj function

(*Shortest import*: `from brian2.codegen.codeobject import create_runner_codeobj`)

`brian2.codegen.codeobject.`**`create_runner_codeobj`**(*group*, *code*, *template_name*, *user_code=None*, *variable_indices=None*, *name=None*, *check_units=True*, *needed_variables=None*, *additional_variables=None*, *level=0*, *run_namespace=None*, *template_kwds=None*, *override_conditional_write=None*, *codeobj_class=None*)

Create a *CodeObject* for the execution of code in the context of a *Group*.

Parameters **group** : *Group*

The group where the code is to be run

code : str or dict of str

The code to be executed.

template_name : str

The name of the template to use for the code.

user_code : str, optional

The code that had been specified by the user before other code was added automatically. If not specified, will be assumed to be identical to `code`.

variable_indices : dict-like, optional

A mapping from `Variable` objects to index names (strings). If none is given, uses the corresponding attribute of `group`.

name : str, optional

A name for this code object, will use `group + '_codeobject*'` if none is given.

check_units : bool, optional

Whether to check units in the statement. Defaults to `True`.

needed_variables: list of str, optional :

A list of variables that are neither present in the abstract code, nor in the `USES_VARIABLES` statement in the template. This is only rarely necessary, an example being a *`StateMonitor`* where the names of the variables are neither known to the template nor included in the abstract code statements.

additional_variables : dict-like, optional

A mapping of names to `Variable` objects, used in addition to the variables saved in `group`.

level : int, optional

How far to go up in the stack to find the call frame.

run_namespace : dict-like, optional

An additional namespace that is used for variable lookup (if not defined, the implicit namespace of local variables is used).

template_kwds : dict, optional

A dictionary of additional information that is passed to the template.

override_conditional_write: list of str, optional :

A list of variable names which are used as conditions (e.g. for refractoriness) which should be ignored.

codeobj_class : class, optional

The *`CodeObject`* class to run code with. If not specified, defaults to the `group`'s `codeobj_class` attribute.

`cpp_prefs` module

Preferences related to C++ compilation

Preferences

C++ compilation preferences `codegen.cpp.compiler = ''`

Compiler to use (uses default if empty)

Should be gcc or msvc.

`codegen.cpp.define_macros = []`

List of macros to define; each macro is defined using a 2-tuple, where 'value' is either the string to define it to or None to define it without a particular value (equivalent of "#define FOO" in source or -DFOO on Unix C compiler command line).

`codegen.cpp.extra_compile_args = None`

Extra arguments to pass to compiler (if None, use either `extra_compile_args_gcc` or `extra_compile_args_msvs`).

`codegen.cpp.extra_compile_args_gcc = ['-w', '-O3']`

Extra compile arguments to pass to GCC compiler

`codegen.cpp.extra_compile_args_msvc = ['/Ox', '/EHsc', '/w']`

Extra compile arguments to pass to MSVC compiler

`codegen.cpp.extra_link_args = []`

Any extra platform- and compiler-specific information to use when linking object files together.

`codegen.cpp.headers = []`

A list of strings specifying header files to use when compiling the code. The list might look like ["<vector>","'my_header"']. Note that the header strings need to be in a form than can be pasted at the end of a #include statement in the C++ code.

`codegen.cpp.include_dirs = []`

Include directories to use. Note that `$prefix/include` will be appended to the end automatically, where `$prefix` is Python's site-specific directory prefix as returned by `sys.prefix`.

`codegen.cpp.libraries = []`

List of library names (not filenames or paths) to link against.

`codegen.cpp.library_dirs = []`

List of directories to search for C/C++ libraries at link time.

`codegen.cpp.msvc_architecture = ''`

MSVC architecture name (or use system architectue by default).

Could take values such as x86, amd64, etc.

`codegen.cpp.msvc_vars_location = ''`

Location of the MSVC command line tool (or search for best by default).

`codegen.cpp.runtime_library_dirs = []`

List of directories to search for C/C++ libraries at run time.

Exported members: *get_compiler_and_args*

Functions

get_compiler_and_args()	Returns the computed compiler and compilation flags

get_compiler_and_args function

(*Shortest import*: `from brian2.codegen.cpp_prefs import get_compiler_and_args`)

`brian2.codegen.cpp_prefs.`**`get_compiler_and_args`**`()`
Returns the computed compiler and compilation flags

`permutation_analysis` module

Module for analysing synaptic pre and post code for synapse order independence.

Exported members: *OrderDependenceError*, *check_for_order_independence*

Classes

OrderDependenceError

OrderDependenceError class

(*Shortest import*: `from brian2.codegen.permutation_analysis import OrderDependenceError`)

class `brian2.codegen.permutation_analysis.`**`OrderDependenceError`**
Bases: `exceptions.Exception`

Functions

check_for_order_independence(statements, ...)

check_for_order_independence function

(*Shortest import*: `from brian2.codegen.permutation_analysis import check_for_order_independence`)

`brian2.codegen.permutation_analysis.`**`check_for_order_independence`**(*statements*, *variables*, *indices*)

`statements` module

Module providing the *Statement* class.

Classes

Statement(var, op, expr, comment, dtype[, ...])	A single line mathematical statement.

Statement class

(*Shortest import*: `from brian2 import Statement`)

class `brian2.codegen.statements.`**`Statement`**(*var*, *op*, *expr*, *comment*, *dtype*, *constant=False*, *subexpression=False*, *scalar=False*)
Bases: `object`

A single line mathematical statement.

The structure is `var op expr`.

Parameters **var** : str

The left hand side of the statement, the value being written to.

op : str

The operation, can be any of the standard Python operators (including += etc.) or a special operator `:=` which means you are defining a new symbol (whereas = means you are setting the value of an existing symbol).

expr : str, *Expression*

The right hand side of the statement.

dtype : dtype

The numpy dtype of the value or array `var()`.

constant : bool, optional

Set this flag to `True` if the value will not change (only applies for `op==':='`.

subexpression : bool, optional

Set this flag to `True` if the variable is a subexpression. In some languages (e.g. Python) you can use this to save a memory copy, because you don't need to do `lhs[:] = rhs` but a redefinition `lhs = rhs`.

scalar : bool, optional

Set this flag to `True` if `var()` and `expr` are scalar.

Notes

Will compute the following attributes:

inplace True or False depending if the operation is in-place or not.

`targets` module

Module that stores all known code generation targets as `codegen_targets`.

Exported members: `codegen_targets`

`templates` module

Handles loading templates from a directory.

Exported members: *Templater*

Classes

CodeObjectTemplate(template, template_source)	**Attributes**

CodeObjectTemplate class

(*Shortest import*: `from brian2.codegen.templates import CodeObjectTemplate`)

class `brian2.codegen.templates.`**CodeObjectTemplate**(*template*, *template_source*)
Bases: `object`

Attributes

allows_scalar_write	Does this template allow writing to scalar variables?

Continued on next page

Table 6.24 – continued from previous page

iterate_all	The indices over which the template iterates completely
variables	The set of variables in this template

Methods

__call__(scalar_code, vector_code, **kwds)	

Details

allows_scalar_write
Does this template allow writing to scalar variables?

iterate_all
The indices over which the template iterates completely

variables
The set of variables in this template

__call__(*scalar_code*, *vector_code*, ***kwds*)

MultiTemplate(module)	

MultiTemplate class

(*Shortest import*: `from brian2.codegen.templates import MultiTemplate`)

class `brian2.codegen.templates.`**MultiTemplate**(*module*)
Bases: `object`

Templater(package_name[, env_globals])	Class to load and return all the templates a *CodeObject* defines.

Templater class

(*Shortest import*: `from brian2.codegen.templates import Templater`)

class `brian2.codegen.templates.`**Templater**(*package_name*, *env_globals=None*)
Bases: `object`

Class to load and return all the templates a *CodeObject* defines.

Functions

autoindent(code)	

autoindent function

(*Shortest import*: `from brian2.codegen.templates import autoindent`)

`brian2.codegen.templates.`**autoindent**(*code*)

autoindent_postfilter(code)	

autoindent_postfilter function

(*Shortest import*: `from brian2.codegen.templates import autoindent_postfilter`)

`brian2.codegen.templates.`**`autoindent_postfilter`**`(`*code*`)`

`translation` module

This module translates a series of statements into a language-specific syntactically correct code block that can be inserted into a template.

It infers whether or not a variable can be declared as constant, etc. It should handle common subexpressions, and so forth.

The input information needed:

- The sequence of statements (a multiline string) in standard mathematical form
- The list of known variables, common subexpressions and functions, and for each variable whether or not it is a value or an array, and if an array what the dtype is.
- The dtype to use for newly created variables
- The language to translate to

Exported members: *make_statements()*, *analyse_identifiers()*, *get_identifiers_recursively()*

Classes

LIONodeRenderer(variables)	Renders expressions, pulling out scalar expressions and remembering them for later use.

LIONodeRenderer class

(*Shortest import*: `from brian2.codegen.translation import LIONodeRenderer`)

class `brian2.codegen.translation.`**`LIONodeRenderer`**`(`*variables*`)`

Bases: *brian2.parsing.rendering.NodeRenderer*

Renders expressions, pulling out scalar expressions and remembering them for later use.

Methods

render_node(node)	

Details

`render_node``(`*node*`)`

LineInfo(**kwds)	A helper class, just used to store attributes.

LineInfo class

(*Shortest import*: `from brian2.codegen.translation import LineInfo`)

class `brian2.codegen.translation.`**`LineInfo`**`(**kwds)`
Bases: `object`

A helper class, just used to store attributes.

Functions

analyse_identifiers(code, variables[, recursive])	Analyses a code string (sequence of statements) to find all identifiers by ty

analyse_identifiers function

(*Shortest import*: `from brian2 import analyse_identifiers`)

`brian2.codegen.translation.`**`analyse_identifiers`**`(`*code*, *variables*, *recursive=False*`)`
Analyses a code string (sequence of statements) to find all identifiers by type.

In a given code block, some variable names (identifiers) must be given as inputs to the code block, and some are created by the code block. For example, the line:

```
a = b+c
```

This could mean to create a new variable a from b and c, or it could mean modify the existing value of a from b or c, depending on whether a was previously known.

Parameters **code** : str

The code string, a sequence of statements one per line.

variables : dict of `Variable`, set of names

Specifiers for the model variables or a set of known names

recursive : bool, optional

Whether to recurse down into subexpressions (defaults to `False`).

Returns **newly_defined** : set

A set of variables that are created by the code block.

used_known : set

A set of variables that are used and already known, a subset of the `known` parameter.

unknown : set

A set of variables which are used by the code block but not defined by it and not previously known. Should correspond to variables in the external namespace.

apply_loop_invariant_optimisations(...)	Analyzes statements to pull out expressions that need to be evaluated only on

apply_loop_invariant_optimisations function

(*Shortest import*: `from brian2.codegen.translation import apply_loop_invariant_optimisations`)

brian2.codegen.translation.**apply_loop_invariant_optimisations**(*statements*, *variables*, *dtype*)

Analyzes statements to pull out expressions that need to be evaluated only once.

Parameters **statements** : list of *Statement*

The statements to analyze.

variables : dict-like

A mapping of identifier names used in `statements` to `Variable` or *Function* objects.

dtype : `dtype`

The data type to use for the newly introduced scalar constants

Returns **scalar_stmts, vector_stmts** : pair of list of *Statement* objects

A list of new scalar statements to define constant for expressions that need to be evaluated only once and the rewritten statements using those constants

get_identifiers_recursively(expressions, ...)	Gets all the identifiers in a list of expressions, recursing down into subexp

get_identifiers_recursively function

(*Shortest import*: `from brian2 import get_identifiers_recursively`)

brian2.codegen.translation.**get_identifiers_recursively**(*expressions*, *variables*, *include_numbers=False*)

Gets all the identifiers in a list of expressions, recursing down into subexpressions.

Parameters **expressions** : list of str

List of expressions to check.

variables : dict-like

Dictionary of `Variable` objects

include_numbers : bool, optional

Whether to include number literals in the output. Defaults to `False`.

has_non_float(expr, variables)	Whether the given expression has an integer or boolean variable in it.

has_non_float function

(*Shortest import*: `from brian2.codegen.translation import has_non_float`)

brian2.codegen.translation.**has_non_float**(*expr*, *variables*)

Whether the given expression has an integer or boolean variable in it.

Parameters **expr** : str

The expression to check

variables : dict-like

`Variable` and *Function* object for all the identifiers used in `expr`

Returns **has_non_float** : bool

Whether `expr` has an integer or boolean in it

is_scalar_expression(expr, variables)	Whether the given expression is scalar.

is_scalar_expression function

(*Shortest import*: `from brian2.codegen.translation import is_scalar_expression`)

`brian2.codegen.translation.`**`is_scalar_expression`**(*expr*, *variables*)

Whether the given expression is scalar.

Parameters **expr** : str

The expression to check

variables : dict-like

`Variable` and *`Function`* object for all the identifiers used in `expr`

Returns **scalar** : bool

Whether `expr` is a scalar expression

make_statements(code, variables, dtype)	Turn a series of abstract code statements into Statement objects, inferring whether ea

make_statements function

(*Shortest import*: `from brian2 import make_statements`)

`brian2.codegen.translation.`**`make_statements`**(*code*, *variables*, *dtype*)

Turn a series of abstract code statements into Statement objects, inferring whether each line is a set/declare operation, whether the variables are constant or not, and handling the cacheing of subexpressions.

Parameters **code** : str

A (multi-line) string of statements.

variables : dict-like

A dictionary of with `Variable` and *`Function`* objects for every identifier used in the `code`.

dtype : `dtype`

The data type to use for temporary variables

Returns **scalar_statements, vector_statements** : (list of *`Statement`*, list of *`Statement`*)

Lists with statements that are to be executed once and statements that are to be executed once for every neuron/synapse/... (or in a vectorised way)

Notes

The `scalar_statements` may include newly introduced scalar constants that have been identified as loop-invariant and have therefore been pulled out of the vector statements. The resulting statements will also use augmented assignments where possible, i.e. a statement such as `w = w + 1` will be replaced by `w += 1`.

Subpackages

generators package

base module Base class for generating code in different programming languages, gives the methods which should be overridden to implement a new language.

Exported members: `CodeGenerator`

Classes

`CodeGenerator`(variables, variable_indices, ...)	Base class for all languages.

CodeGenerator class (*Shortest import*: `from brian2 import CodeGenerator`)

class `brian2.codegen.generators.base.`**CodeGenerator**(*variables*, *variable_indices*, *owner*, *iterate_all*, *codeobj_class*, *name*, *template_name*, *override_conditional_write=None*, *allows_scalar_write=False*)

Bases: `object`

Base class for all languages.

See definition of methods below.

TODO: more details here

Methods

`array_read_write`(statements)	Helper function, gives the set of ArrayVariables that are read from and
`arrays_helper`(statements)	Combines the two helper functions `array_read_write` and `get_`
`determine_keywords`()	A dictionary of values that is made available to the templated.
`get_array_name`(var[, access_data])	Get a globally unique name for a `ArrayVariable`.
`get_conditional_write_vars`()	Helper function, returns a dict of mappings `(varname, conditio`
`translate`(code, dtype)	Translates an abstract code block into the target language.
`translate_expression`(expr)	Translate the given expression string into a string in the target language
`translate_one_statement_sequence`(statements)	
`translate_statement`(statement)	Translate a single line `Statement` into the target language, returns a
`translate_statement_sequence`(...)	Translate a sequence of `Statement` into the target language, taking c

Details

array_read_write(*statements*)
Helper function, gives the set of ArrayVariables that are read from and written to in the series of statements. Returns the pair read, write of sets of variable names.

arrays_helper(*statements*)
Combines the two helper functions `array_read_write` and `get_conditional_write_vars`, and updates the `read` set.

determine_keywords()
A dictionary of values that is made available to the templated. This is used for example by the `CPPCodeGenerator` to set up all the supporting code

static get_array_name(*var*, *access_data=True*)
Get a globally unique name for a `ArrayVariable`.

Parameters **var** : `ArrayVariable`

The variable for which a name should be found.

access_data : bool, optional

For `DynamicArrayVariable` objects, specifying `True` here means the name for the underlying data is returned. If specifying `False`, the name of object itself is returned (e.g. to allow resizing).

Returns :

——- :

name : str

A uniqe name for `var()`.

get_conditional_write_vars()
Helper function, returns a dict of mappings `(varname, condition_var_name)` indicating that when `varname` is written to, it should only be when `condition_var_name` is `True`.

translate(*code*, *dtype*)
Translates an abstract code block into the target language.

translate_expression(*expr*)
Translate the given expression string into a string in the target language, returns a string.

translate_one_statement_sequence(*statements*, *scalar=False*)

translate_statement(*statement*)
Translate a single line *Statement* into the target language, returns a string.

translate_statement_sequence(*scalar_statements*, *vector_statements*)
Translate a sequence of *Statement* into the target language, taking care to declare variables, etc. if necessary.

Returns a tuple `(scalar_code, vector_code, kwds)` where `scalar_code` is a list of the lines of code executed before the inner loop, `vector_code` is a list of the lines of code in the inner loop, and `kwds` is a dictionary of values that is made available to the template.

cpp_generator module *Exported members: CPPCodeGenerator, c_data_type()*

Classes

CPPCodeGenerator(*args, **kwds)	C++ language

CPPCodeGenerator class (*Shortest import*: `from brian2 import CPPCodeGenerator`)

class `brian2.codegen.generators.cpp_generator.`**CPPCodeGenerator**(**args*, ***kwds*)
Bases: *brian2.codegen.generators.base.CodeGenerator*

C++ language

C++ code templates should provide Jinja2 macros with the following names:

main The main loop.

support_code The support code (function definitions, etc.), compiled in a separate file.

For user-defined functions, there are two keys to provide:

support_code The function definition which will be added to the support code.

hashdefine_code The `#define` code added to the main loop.

See *TimedArray* for an example of these keys.

Attributes

flush_denormals
restrict

Methods

denormals_to_zero_code()
determine_keywords()
get_array_name(var[, access_data])
translate_expression(expr)
translate_one_statement_sequence(statements)
translate_statement(statement)
translate_to_declarations(statements)
translate_to_read_arrays(statements)
translate_to_statements(statements)
translate_to_write_arrays(statements)

Details

flush_denormals

restrict

denormals_to_zero_code()

determine_keywords()

static **get_array_name**(*var*, *access_data=True*)

translate_expression(*expr*)

translate_one_statement_sequence(*statements*, *scalar=False*)

translate_statement(*statement*)

translate_to_declarations(*statements*)

translate_to_read_arrays(*statements*)

translate_to_statements(*statements*)

translate_to_write_arrays(*statements*)

Functions

c_data_type(dtype)	Gives the C language specifier for numpy data types.

c_data_type function (*Shortest import*: `from brian2 import c_data_type`)

`brian2.codegen.generators.cpp_generator.`**`c_data_type`**(*dtype*)
Gives the C language specifier for numpy data types. For example, `numpy.int32` maps to `int32_t` in C.

`cython_generator` module *Exported members:* *CythonCodeGenerator*

Classes

CythonCodeGenerator(variables, ...[, ...])	Cython code generator

CythonCodeGenerator class (*Shortest import*: `from brian2 import CythonCodeGenerator`)

class `brian2.codegen.generators.cython_generator.`**`CythonCodeGenerator`**(*variables*, *variable_indices*, *owner*, *iterate_all*, *codeobj_class*, *name*, *template_name*, *override_conditional_write=None*, *allows_scalar_write=False*)

Bases: *brian2.codegen.generators.base.CodeGenerator*

Cython code generator

Methods

determine_keywords()	
translate_expression(expr)	
translate_one_statement_sequence(statements)	
translate_statement(statement)	

Details

`determine_keywords`()

`translate_expression`(*expr*)

`translate_one_statement_sequence`(*statements*, *scalar=False*)

`translate_statement`(*statement*)

CythonNodeRenderer([use_vectorisation_idx])	**Methods**

CythonNodeRenderer class (*Shortest import*: `from brian2.codegen.generators.cython_generator import CythonNodeRenderer`)

class `brian2.codegen.generators.cython_generator.`**`CythonNodeRenderer`**(*use_vectorisation_idx=True*)
Bases: *brian2.parsing.rendering.NodeRenderer*

Methods

render_Name(node)	
render_NameConstant(node)	

Details

render_Name(*node*)

render_NameConstant(*node*)

Functions

get_cpp_dtype(obj)	

get_cpp_dtype function (*Shortest import*: `from brian2.codegen.generators.cython_generator import get_cpp_dtype`)

brian2.codegen.generators.cython_generator.**get_cpp_dtype**(*obj*)

get_numpy_dtype(obj)	

get_numpy_dtype function (*Shortest import*: `from brian2.codegen.generators.cython_generator import get_numpy_dtype`)

brian2.codegen.generators.cython_generator.**get_numpy_dtype**(*obj*)

numpy_generator module *Exported members:* *NumpyCodeGenerator*

Classes

NumpyCodeGenerator(variables, ...[, ...])	Numpy language

NumpyCodeGenerator class (*Shortest import*: `from brian2 import NumpyCodeGenerator`)

class brian2.codegen.generators.numpy_generator.**NumpyCodeGenerator**(*variables*, *variable_indices*, *owner*, *iterate_all*, *codeobj_class*, *name*, *template_name*, *override_conditional_write=None*, *allows_scalar_write=False*)

Bases: *brian2.codegen.generators.base.CodeGenerator*

Numpy language

Essentially Python but vectorised.

Methods

conditional_write(line, stmt, variables, ...)
determine_keywords()
read_arrays(read, write, indices, variables, ...)
translate_expression(expr)
translate_one_statement_sequence(statements)
translate_statement(statement)
ufunc_at_vectorisation(statement, variables, ...)
vectorise_code(statements, variables, ...[, ...])
write_arrays(statements, read, write, ...)

Details

conditional_write(*line*, *stmt*, *variables*, *conditional_write_vars*, *created_vars*)

determine_keywords()

read_arrays(*read*, *write*, *indices*, *variables*, *variable_indices*)

translate_expression(*expr*)

translate_one_statement_sequence(*statements*, *scalar=False*)

translate_statement(*statement*)

ufunc_at_vectorisation(*statement*, *variables*, *indices*, *conditional_write_vars*, *created_vars*, *index*)

vectorise_code(*statements*, *variables*, *variable_indices*, *index='_idx'*)

write_arrays(*statements*, *read*, *write*, *variables*, *variable_indices*)

VectorisationError

VectorisationError class (*Shortest import*: `from brian2.codegen.generators.numpy_generator import VectorisationError`)

class brian2.codegen.generators.numpy_generator.**VectorisationError**
Bases: exceptions.Exception

Functions

ceil_func(value)

ceil_func function (*Shortest import*: `from brian2.codegen.generators.numpy_generator import ceil_func`)

brian2.codegen.generators.numpy_generator.**ceil_func**(*value*)

clip_func(array, a_min, a_max)

clip_func function (*Shortest import*: `from brian2.codegen.generators.numpy_generator import clip_func`)

brian2.codegen.generators.numpy_generator.**clip_func**(*array*, *a_min*, *a_max*)

floor_func(value)

floor_func function (*Shortest import*: `from brian2.codegen.generators.numpy_generator import floor_func`)

brian2.codegen.generators.numpy_generator.**floor_func**(*value*)

int_func(value)

int_func function (*Shortest import*: `from brian2.codegen.generators.numpy_generator import int_func`)

brian2.codegen.generators.numpy_generator.**int_func**(*value*)

rand_func(vectorisation_idx)

rand_func function (*Shortest import*: `from brian2.codegen.generators.numpy_generator import rand_func`)

brian2.codegen.generators.numpy_generator.**rand_func**(*vectorisation_idx*)

randn_func(vectorisation_idx)

randn_func function (*Shortest import*: `from brian2.codegen.generators.numpy_generator import randn_func`)

brian2.codegen.generators.numpy_generator.**randn_func**(*vectorisation_idx*)

runtime package

Runtime targets for code generation.

Subpackages

cython_rt package

cython_rt module *Exported members:* *CythonCodeObject*

Classes

CythonCodeObject(owner, code, variables, ...) Execute code using Cython.

CythonCodeObject class (*Shortest import*: `from brian2 import CythonCodeObject`)

class brian2.codegen.runtime.cython_rt.cython_rt.**CythonCodeObject**(*owner*, *code*, *variables*, *variable_indices*, *template_name*, *template_source*, *name='cython_code_object*')

Bases: *brian2.codegen.runtime.numpy_rt.numpy_rt.NumpyCodeObject*

Execute code using Cython.

Methods

compile()
is_available()
run()
update_namespace()
variables_to_namespace()

Details

compile()

static is_available()

run()

update_namespace()

variables_to_namespace()

extension_manager module Cython automatic extension builder/manager

Inspired by IPython's Cython cell magics, see: https://github.com/ipython/ipython/blob/master/IPython/extensions/cythonmagic.py

Exported members: *cython_extension_manager*

Classes

CythonExtensionManager()	**Attributes**

CythonExtensionManager class (*Shortest import*: `from brian2.codegen.runtime.cython_rt.extension_manager import CythonExtensionManager`)

class brian2.codegen.runtime.cython_rt.extension_manager.**CythonExtensionManager**

Bases: `object`

Attributes

so_ext	The extension suffix for compiled modules.

Methods

create_extension(code[, force, name, ...])	

Details

so_ext
The extension suffix for compiled modules.

create_extension(*code*, *force=False*, *name=None*, *include_dirs=None*, *library_dirs=None*, *runtime_library_dirs=None*, *extra_compile_args=None*, *extra_link_args=None*, *libraries=None*, *compiler=None*)

Objects

cython_extension_manager	

cython_extension_manager object (*Shortest import*: `from brian2.codegen.runtime.cython_rt.extension_manag import cython_extension_manager`)

`brian2.codegen.runtime.cython_rt.extension_manager.`**`cython_extension_manager = <brian2.codegen.ru`**

numpy_rt package Numpy runtime implementation.

Preferences Numpy runtime codegen preferences `codegen.runtime.numpy.discard_units = False`

Whether to change the namespace of user-specifed functions to remove units.

numpy_rt module Module providing *NumpyCodeObject*.

Exported members: *NumpyCodeObject*

Classes

NumpyCodeObject(owner, code, variables, ...)	Execute code using Numpy

NumpyCodeObject class (*Shortest import*: `from brian2 import NumpyCodeObject`)

class `brian2.codegen.runtime.numpy_rt.numpy_rt.`**`NumpyCodeObject`**(*owner*, *code*, *variables*, *variable_indices*, *template_name*, *template_source*, *name='numpy_code_object*'*)

Bases: *brian2.codegen.codeobject.CodeObject*

Execute code using Numpy

Default for Brian because it works on all platforms.

Methods

compile()
is_available()
run()
update_namespace()
variables_to_namespace()

Details

compile()

static is_available()

run()

update_namespace()

variables_to_namespace()

synapse_vectorisation module Module for efficient vectorisation of synapses code

Exported members: *vectorise_synapses_code*, *SynapseVectorisationError*

Classes

SynapseVectorisationError

SynapseVectorisationError class (*Shortest import*: `from brian2.codegen.runtime.numpy_rt.synapse_vectorisa import SynapseVectorisationError`)

class brian2.codegen.runtime.numpy_rt.synapse_vectorisation.**SynapseVectorisationError**
Bases: exceptions.Exception

Functions

ufunc_at_vectorisation(statements, ...)

ufunc_at_vectorisation function (*Shortest import*: `from brian2.codegen.runtime.numpy_rt.synapse_vectorisat import ufunc_at_vectorisation`)

brian2.codegen.runtime.numpy_rt.synapse_vectorisation.**ufunc_at_vectorisation**(*statements*, *variables*, *indices*, *index*)

vectorise_synapses_code(statements, ...[, index])

vectorise_synapses_code function (*Shortest import*: `from brian2.codegen.runtime.numpy_rt.synapse_vectorisa import vectorise_synapses_code`)

brian2.codegen.runtime.numpy_rt.synapse_vectorisation.**vectorise_synapses_code**(*statements*, *variables*, *indices*, *index='_idx'*)

weave_rt package Runtime C++ code generation via weave.

weave_rt module Module providing *WeaveCodeObject*.

Exported members: *WeaveCodeObject*, *WeaveCodeGenerator*

Classes

WeaveCodeGenerator(*args, **kwds)

WeaveCodeGenerator class (*Shortest import*: `from brian2 import WeaveCodeGenerator`)

class brian2.codegen.runtime.weave_rt.weave_rt.**WeaveCodeGenerator**(**args*, ***kwds*)

Bases: *brian2.codegen.generators.cpp_generator.CPPCodeGenerator*

WeaveCodeObject(owner, code, variables, ...)	Weave code object

WeaveCodeObject class (*Shortest import*: `from brian2 import WeaveCodeObject`)

class brian2.codegen.runtime.weave_rt.weave_rt.**WeaveCodeObject**(*owner*, *code*, *variables*, *variable_indices*, *template_name*, *template_source*, *name='weave_code_object*'*)

Bases: *brian2.codegen.codeobject.CodeObject*

Weave code object

The `code` should be a *MultiTemplate* object with two macros defined, `main` (for the main loop code) and `support_code` for any support code (e.g. function definitions).

Methods

compile()
is_available()
run()
update_namespace()
variables_to_namespace()

Details

compile()

static is_available()

run()

update_namespace()

variables_to_namespace()

Functions

weave_data_type(dtype)	Gives the C language specifier for numpy data types using weave.

weave_data_type function (*Shortest import*: `from brian2.codegen.runtime.weave_rt.weave_rt import weave_data_type`)

brian2.codegen.runtime.weave_rt.weave_rt.**weave_data_type**(*dtype*)

Gives the C language specifier for numpy data types using weave. For example, `numpy.int32` maps to `long` in C.

6.4.2 core package

Essential Brian modules, in particular base classes for all kinds of brian objects.

Built-in preferences

Core Brian preferences `core.default_float_dtype` = `float64`

Default dtype for all arrays of scalars (state variables, weights, etc.).

`core.default_integer_dtype` = `int32`

Default dtype for all arrays of integer scalars.

`core.outdated_dependency_error` = `True`

Whether to raise an error for outdated dependencies (`True`) or just a warning (`False`).

`base` module

All Brian objects should derive from *BrianObject*.

Exported members: *BrianObject*, *weakproxy_with_fallback()*

Classes

BrianObject(*args, **kwds)	All Brian objects derive from this class, defines magic tracking and update.

BrianObject class

(*Shortest import*: `from brian2 import BrianObject`)

class brian2.core.base.**BrianObject**(**args*, ***kwds*)

Bases: *brian2.core.names.Nameable*

All Brian objects derive from this class, defines magic tracking and update.

See the documentation for *Network* for an explanation of which objects get updated in which order.

Parameters **dt** : *Quantity*, optional

The time step to be used for the simulation. Cannot be combined with the *clock* argument.

clock : *Clock*, optional

The update clock to be used. If neither a clock, nor the `dt` argument is specified, the *defaultclock* will be used.

when : str, optional

In which scheduling slot to simulate the object during a time step. Defaults to `'start'`.

order : int, optional

The priority of this object for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

name : str, optional

A unique name for the object - one will be assigned automatically if not provided (of the form `brianobject_1`, etc.).

Notes

The set of all *BrianObject* objects is stored in `BrianObject.__instances__()`.

Attributes

_clock	The clock used for simulating this object
_network	Used to remember the *Network* in which this object has been included
_scope_current_key	Global key value for ipython cell restrict magic
_scope_key	The scope key is used to determine which objects are collected by magic
active	Whether or not the object should be run.
add_to_magic_network	Whether or not the object should be added to a *MagicNetwork*.
clock	The *Clock* determining when the object should be updated.
code_objects	The list of *CodeObject* contained within the *BrianObject*.
contained_objects	The list of objects contained within the *BrianObject*.
invalidates_magic_network	Whether or not *MagicNetwork* is invalidated when a new *BrianObject* of this type is a
name	The unique name for this object.
order	The order in which objects with the same clock and `when` should be updated
updaters	The list of `Updater` that define the runtime behaviour of this object.
when	The ID string determining when the object should be updated in *Network.run()*.

Methods

add_dependency(obj)	Add an object to the list of dependencies.
after_run()	Optional method to do work after a run is finished.
before_run([run_namespace, level])	Optional method to prepare the object before a run.
run()	

Details

_clock
The clock used for simulating this object

_network
Used to remember the *Network* in which this object has been included before, to raise an error if it is included in a new *Network*

_scope_current_key
Global key value for ipython cell restrict magic

_scope_key
The scope key is used to determine which objects are collected by magic

active
Whether or not the object should be run.

Inactive objects will not have their `update` method called in *Network.run()*. Note that setting or unsetting the *active* attribute will set or unset it for all *contained_objects*.

add_to_magic_network
Whether or not the object should be added to a *MagicNetwork*. Note that all objects in *BrianObject.contained_objects* are automatically added when the parent object is added, therefore e.g. *NeuronGroup* should set *add_to_magic_network* to `True`, but it should not be set for all the dependent objects such as `StateUpdater`

clock
The *Clock* determining when the object should be updated.

Note that this cannot be changed after the object is created.

code_objects
The list of *CodeObject* contained within the *BrianObject*.

TODO: more details.

Note that this attribute cannot be set directly, you need to modify the underlying list, e.g. `obj.code_objects.extend([A, B])`.

contained_objects
The list of objects contained within the *BrianObject*.

When a *BrianObject* is added to a *Network*, its contained objects will be added as well. This allows for compound objects which contain a mini-network structure.

Note that this attribute cannot be set directly, you need to modify the underlying list, e.g. `obj.contained_objects.extend([A, B])`.

invalidates_magic_network
Whether or not *MagicNetwork* is invalidated when a new *BrianObject* of this type is added

name
The unique name for this object.

Used when generating code. Should be an acceptable variable name, i.e. starting with a letter character and followed by alphanumeric characters and _.

order
The order in which objects with the same clock and `when` should be updated

updaters
The list of `Updater` that define the runtime behaviour of this object.

TODO: more details.

Note that this attribute cannot be set directly, you need to modify the underlying list, e.g. `obj.updaters.extend([A, B])`.

when
The ID string determining when the object should be updated in *Network.run()*.

add_dependency(*obj*)
Add an object to the list of dependencies. Takes care of handling subgroups correctly (i.e., adds its parent object).

Parameters **obj** : *BrianObject*

The object that this object depends on.

after_run()
Optional method to do work after a run is finished.

Called by *Network.after_run()* after the main simulation loop terminated.

before_run(*run_namespace=None*, *level=0*)
Optional method to prepare the object before a run.

TODO

run()

Functions

device_override(name)	Decorates a function/method to allow it to be overridden by the current `Device`.

device_override function

(*Shortest import*: `from brian2.core.base import device_override`)

brian2.core.base.**device_override**(*name*)
Decorates a function/method to allow it to be overridden by the current `Device`.

The `name` is the function name in the `Device` to use as an override if it exists.

The returned function has an additional attribute `original_function` which is a reference to the original, undecorated function.

weakproxy_with_fallback(obj)	Attempts to create a `weakproxy` to the object, but falls back to the object if not possible.

weakproxy_with_fallback function

(*Shortest import*: `from brian2 import weakproxy_with_fallback`)

brian2.core.base.**weakproxy_with_fallback**(*obj*)
Attempts to create a `weakproxy` to the object, but falls back to the object if not possible.

clocks module

Clocks for the simulator.

Exported members: *Clock*, *defaultclock*

Classes

Clock(dt[, name])	An object that holds the simulation time and the time step.

Clock class

(*Shortest import*: `from brian2 import Clock`)

class `brian2.core.clocks.`**`Clock`**(*dt*, *name='clock**'*)
Bases: *brian2.core.names.Nameable*

An object that holds the simulation time and the time step.

Parameters **dt** : float

The time step of the simulation as a float

name : str, optional

An explicit name, if not specified gives an automatically generated name

Notes

Clocks are run in the same *Network.run()* iteration if *t* is the same. The condition for two clocks to be considered as having the same time is `abs(t1-t2)<epsilon*abs(t1)`, a standard test for equality of floating point values. The value of `epsilon` is `1e-14`.

Attributes

_dt	The internally used dt.
dt	The time step of the simulation in seconds.
*dt_*	The time step of the simulation as a float (in seconds)
running	A `bool` to indicate whether the current simulation is running.
t	The simulation time in seconds
*t_*	The simulation time as a float (in seconds)
t_end	The time the simulation will end (in seconds)

Methods

tick()	Advances the clock by one time step.

Details

_dt
The internally used dt. Note that right after a change of dt, this will not equal the new dt (which is stored in `Clock._new_dt`). Call `Clock._set_t_update_t` to update the internal clock representation.

dt
The time step of the simulation in seconds.

dt_
The time step of the simulation as a float (in seconds)

running
A `bool` to indicate whether the current simulation is running.

t
The simulation time in seconds

t_
The simulation time as a float (in seconds)

t_end
The time the simulation will end (in seconds)

set_interval (*self*, *start*, *end*)
Set the start and end time of the simulation.

Sets the start and end value of the clock precisely if possible (using epsilon) or rounding up if not. This assures that multiple calls to *Network.run()* will not re-run the same time step.

tick()
Advances the clock by one time step.

Tutorials and examples using this

- Example CUBA
- Example COBAHH

Objects

defaultclock	An object that holds the simulation time and the time step.

defaultclock object

(*Shortest import*: `from brian2 import defaultclock`)

brian2.core.clocks.**defaultclock**
An object that holds the simulation time and the time step.

Parameters **dt** : float

The time step of the simulation as a float

name : str, optional

An explicit name, if not specified gives an automatically generated name

Notes

Clocks are run in the same *Network.run()* iteration if *t* is the same. The condition for two clocks to be considered as having the same time is `abs(t1-t2)<epsilon*abs(t1)`, a standard test for equality of floating point values. The value of `epsilon` is `1e-14`.

core_preferences module

Definitions, documentation, default values and validation functions for core Brian preferences.

Functions

dtype_repr(dtype)	

dtype_repr function

(*Shortest import*: `from brian2 import dtype_repr`)

`brian2.core.core_preferences.`**`dtype_repr`**(*dtype*)

`functions` module

Exported members: `DEFAULT_FUNCTIONS`, *Function*, *implementation()*

Classes

Function(pyfunc[, sympy_func, arg_units, ...])	An abstract specification of a function that can be used as part of model equations

Function class

(*Shortest import*: `from brian2 import Function`)

class `brian2.core.functions.`**`Function`**(*pyfunc*, *sympy_func=None*, *arg_units=None*, *return_unit=None*, *stateless=True*)

Bases: `object`

An abstract specification of a function that can be used as part of model equations, etc.

Parameters **pyfunc** : function

A Python function that is represented by this *Function* object.

sympy_func : `sympy.Function`, optional

A corresponding sympy function (if any). Allows functions to be interpreted by sympy and potentially make simplifications. For example, `sqrt(x**2)` could be replaced by `abs(x)`.

arg_units : list of *Unit*, optional

If `pyfunc` does not provide unit information (which typically means that it was not annotated with a *check_units()* decorator), the units of the arguments have to specified explicitly using this parameter.

return_unit : *Unit* or callable, optional

Same as for `arg_units`: if `pyfunc` does not provide unit information, this information has to be provided explictly here. `return_unit` can either be a specific *Unit*, if the function always returns the same unit, or a function of the input units, e.g. a "square" function would return the square of its input units, i.e. `return_unit` could be specified as `lambda u: u**2`.

stateless : bool, optional

Whether this function does not have an internal state, i.e. if it always returns the same output when called with the same arguments. This is true for mathematical functions but not true for `rand()`, for example. Defaults to `True`.

Notes

If a function should be usable for code generation targets other than Python/numpy, implementations for these target languages have to be added using the `implementation` decorator or using the `add_implementations` function.

Attributes

implementations	Stores implementations for this function in a

Methods

__call__(*args)	
is_locally_constant(dt)	Return whether this function (if interpreted as a function of time) should be considered constant ov

Details

implementations
Stores implementations for this function in a *FunctionImplementationContainer*

__call__ (**args*)

is_locally_constant (*dt*)
Return whether this function (if interpreted as a function of time) should be considered constant over a timestep. This is most importantly used by *TimedArray* so that linear integration can be used. In its standard implementation, always returns `False`.

Parameters **dt** : float

The length of a timestep (without units).

Returns **constant** : bool

Whether the results of this function can be considered constant over one timestep of length `dt`.

FunctionImplementation([name, code, ...])	A simple container object for function implementations.

FunctionImplementation class

(*Shortest import*: `from brian2.core.functions import FunctionImplementation`)

class `brian2.core.functions.`**FunctionImplementation**(*name=None*, *code=None*, *namespace=None*, *dependencies=None*, *dynamic=False*)

Bases: `object`

A simple container object for function implementations.

Parameters **name** : str, optional

The name of the function in the target language. Should only be specified if the function has to be renamed for the target language.

code : language-dependent, optional

A language dependent argument specifying the implementation in the target language, e.g. a code string or a dictionary of code strings.

namespace : dict-like, optional

A dictionary of mappings from names to values that should be added to the namespace of a *CodeObject* using the function.

dependencies : dict-like, optional

A mapping of names to *Function* objects, for additional functions needed by this function.

dynamic : bool, optional

Whether this `code`/`namespace` is dynamic, i.e. generated for each new context it is used in. If set to `True`, `code` and `namespace` have to be callable with a *Group* as an argument and are expected to return the final `code` and `namespace`. Defaults to `False`.

Methods

get_code(owner)	
get_namespace(owner)	

Details

get_code(*owner*)

get_namespace(*owner*)

FunctionImplementationContainer(function)	Helper object to store implementations and give access in a dictionary-like

FunctionImplementationContainer class

(*Shortest import*: `from brian2.core.functions import FunctionImplementationContainer`)

class `brian2.core.functions.`**FunctionImplementationContainer**(*function*)

Bases: `_abcoll.Mapping`

Helper object to store implementations and give access in a dictionary-like fashion, using *CodeGenerator* implementations as a fallback for *CodeObject* implementations.

Methods

add_dynamic_implementation(target, code[, ...])	Adds an "dynamic implementation" for this function.
add_implementation(target, code[, ...])	
add_numpy_implementation(wrapped_func[, ...])	Add a numpy implementation to a *Function*.

Details

add_dynamic_implementation(*target*, *code*, *namespace=None*, *dependencies=None*, *name=None*)
Adds an "dynamic implementation" for this function. `code` and `namespace` arguments are expected to be callables that will be called in *Network.before_run()* with the owner of the *CodeObject* as an argument. This allows to generate code that depends on details of the context it is run in, e.g. the `dt` of a clock.

add_implementation(*target*, *code*, *namespace=None*, *dependencies=None*, *name=None*)

add_numpy_implementation(*wrapped_func*, *dependencies=None*, *discard_units=None*)
Add a numpy implementation to a *Function*.

Parameters **function** : *Function*

The function description for which an implementation should be added.

wrapped_func : callable

The original function (that will be used for the numpy implementation)

dependencies : list of *Function*, optional

A list of functions this function needs.

discard_units : bool, optional

See *implementation()*.

SymbolicConstant(name, sympy_obj, value)	Class for representing constants (e.g.

SymbolicConstant class

(*Shortest import*: `from brian2.core.functions import SymbolicConstant`)

class `brian2.core.functions.`**SymbolicConstant**(*name*, *sympy_obj*, *value*)
Bases: *brian2.core.variables.Constant*

Class for representing constants (e.g. pi) that are understood by sympy.

log10	**Methods**

log10 class

(*Shortest import*: `from brian2.core.functions import log10`)

class `brian2.core.functions.`**log10**
Bases: `sympy.core.function.Function`

Methods

eval(args)

Details

classmethod eval (*args*)

Functions

implementation(target[, code, namespace, ...])	A simple decorator to extend user-written Python functions to work with code g

implementation function

(*Shortest import*: `from brian2 import implementation`)

brian2.core.functions.**implementation**(*target*, *code=None*, *namespace=None*, *dependencies=None*, *discard_units=None*)

A simple decorator to extend user-written Python functions to work with code generation in other languages.

Parameters **target** : str

Name of the code generation target (e.g. `'weave'`) for which to add an implementation.

code : str or dict-like, optional

What kind of code the target language expects is language-specific, e.g. C++ code allows for a dictionary of code blocks instead of a single string.

namespaces : dict-like, optional

A namespace dictionary (i.e. a mapping of names to values) that should be added to a *CodeObject* namespace when using this function.

dependencies : dict-like, optional

A mapping of names to *Function* objects, for additional functions needed by this function.

discard_units: bool, optional :

Numpy functions can internally make use of the unit system. However, during a simulation run, state variables are passed around as unitless values for efficiency. If `discard_units` is set to `False`, input arguments will have units added to them so that the function can still use units internally (the units will be stripped away from the return value as well). Alternatively, if `discard_units` is set to `True`, the function will receive unitless values as its input. The namespace of the function will be altered to make references to units (e.g. `ms`) refer to the corresponding floating point values so that no unit mismatch errors are raised. Note that this system cannot work in all cases, e.g. it does not work with functions that internally imports values (e.g. does `from brian2 import ms`) or access values with units indirectly (e.g. uses `brian2.ms` instead of `ms`). If no value is given, defaults to the preference setting *codegen.runtime.numpy.discard_units*.

Notes

While it is in principle possible to provide a numpy implementation as an argument for this decorator, this is normally not necessary – the numpy implementation should be provided in the decorated function.

Examples

Sample usage:

```
@implementation('cpp',"""
            #include<math.h>
            inline double usersin(double x)
            {
                return sin(x);
            }
            """)
def usersin(x):
    return sin(x)
```

`magic` module

Exported members: *MagicNetwork*, *magic_network*, *MagicError*, *run()*, *reinit()*, *stop()*, *collect()*, *store()*, *restore()*, *start_scope()*

Classes

MagicError	Error that is raised when something goes wrong in *MagicNetwork*

MagicError class

(*Shortest import*: `from brian2 import MagicError`)

class `brian2.core.magic.`**MagicError**
Bases: `exceptions.Exception`

Error that is raised when something goes wrong in *MagicNetwork*

See notes to *MagicNetwork* for more details.

MagicNetwork()	*Network* that automatically adds all Brian objects

MagicNetwork class

(*Shortest import*: `from brian2 import MagicNetwork`)

class `brian2.core.magic.`**MagicNetwork**
Bases: *brian2.core.network.Network*

Network that automatically adds all Brian objects

In order to avoid bugs, this class will occasionally raise *MagicError* when the intent of the user is not clear. See the notes below for more details on this point. If you persistently see this error, then Brian is not able to safely guess what you intend to do, and you should use a *Network* object and call *Network.run()* explicitly.

Note that this class cannot be instantiated by the user, there can be only one instance *magic_network* of *MagicNetwork*.

See also:

Network, *collect()*, *run()*, *stop()*, *store()*, *restore()*

Notes

All Brian objects that are visible at the point of the *run()* call will be included in the network. This class is designed to work in the following two major use cases:

1. You create a collection of Brian objects, and call *run()* to run the simulation. Subsequently, you may call *run()* again to run it again for a further duration. In this case, the *Network.t* time will start at 0 and for the second call to *run()* will continue from the end of the previous run.
2. You have a loop in which at each iteration, you create some Brian objects and run a simulation using them. In this case, time is reset to 0 for each call to *run()*.

In any other case, you will have to explicitly create a *Network* object yourself and call *Network.run()* on this object. Brian has a built in system to guess which of the cases above applies and behave correctly. When it is not possible to safely guess which case you are in, it raises *MagicError*. The rules for this guessing system are explained below.

If a simulation consists only of objects that have not been run, it will assume that you want to start a new simulation. If a simulation only consists of objects that have been simulated in the previous *run()* call, it will continue that simulation at the previous time.

If neither of these two situations apply, i.e., the network consists of a mix of previously run objects and new objects, an error will be raised.

In these checks, "non-invalidating" objects (i.e. objects that have *BrianObject.invalidates_magic_network* set to `False`) are ignored, e.g. creating new monitors is always possible.

Methods

add(*objs)	You cannot add objects directly to *MagicNetwork*
after_run()	
check_dependencies()	
remove(*objs)	You cannot remove objects directly from *MagicNetwork*
restore([name, level])	See *Network.store()*.
run(duration[, report, report_period, ...])	
store([name, level])	See *Network.store()*.

Details

add(**objs*)
You cannot add objects directly to *MagicNetwork*

after_run()

check_dependencies()

remove(**objs*)
You cannot remove objects directly from *MagicNetwork*

restore(*name='default'*, *level=0*)
See *Network.store()*.

run(*duration*, *report=None*, *report_period=10. * second*, *namespace=None*, *profile=True*, *level=0*)

store(*name='default'*, *level=0*)
See *Network.store()*.

Functions

collect([level])	Return the list of *BrianObject*s that will be simulated if *run()* is called.

collect function

(*Shortest import*: `from brian2 import collect`)

brian2.core.magic.**collect**(*level=0*)

Return the list of *BrianObject*s that will be simulated if *run()* is called.

Parameters **level** : int, optional

How much further up to go in the stack to find the objects. Needs only to be specified if *collect()* is called as part of a function and should be increased by 1 for every level of nesting. Defaults to 0.

Returns **objects** : set of *BrianObject*

The objects that will be simulated.

get_objects_in_namespace(level)	Get all the objects in the current namespace that derive from *BrianObject*.

get_objects_in_namespace function

(*Shortest import*: `from brian2.core.magic import get_objects_in_namespace`)

brian2.core.magic.**get_objects_in_namespace**(*level*)

Get all the objects in the current namespace that derive from *BrianObject*. Used to determine the objects for the *MagicNetwork*.

Parameters **level** : int, optional

How far to go back to get the locals/globals. Each function/method call should add `1` to this argument, functions/method with a decorator have to add `2`.

Returns **objects** : set

A set with weak references to the *BrianObject*s in the namespace.

reinit()	Reinitialises all Brian objects.

reinit function

(*Shortest import*: `from brian2 import reinit`)

brian2.core.magic.**reinit**()

Reinitialises all Brian objects.

This function works similarly to *run()*, see the documentation for that function for more details.

Raises

MagicError Error raised when it was not possible for Brian to safely guess the intended use. See *MagicNetwork* for more details.

See also:

`Network.reinit`, *`run()`*, *`stop()`*, `clear`

`restore`([name])	Restore the state of the network and all included objects.

restore function

(*Shortest import*: `from brian2 import restore`)

brian2.core.magic.**restore**(*name='default'*)

Restore the state of the network and all included objects.

Parameters **name** : str, optional

The name of the snapshot to restore, if not specified uses `'default'`.

`run`(duration[, report, report_period, ...])	Runs a simulation with all "visible" Brian objects for the given duration.

run function

(*Shortest import*: `from brian2 import run`)

brian2.core.magic.**run**(*duration*, *report=None*, *report_period=10*second*, *namespace=None*, *level=0*)

Runs a simulation with all "visible" Brian objects for the given duration. Calls *`collect()`* to gather all the objects, the simulation can be stopped by calling the global *`stop()`* function.

In order to avoid bugs, this function will occasionally raise *`MagicError`* when the intent of the user is not clear. See the notes to *`MagicNetwork`* for more details on this point. If you persistently see this error, then Brian is not able to safely guess what you intend to do, and you should use a *`Network`* object and call *`Network.run()`* explicitly.

Parameters **duration** : *`Quantity`*

The amount of simulation time to run for. If the network consists of new objects since the last time *`run()`* was called, the start time will be reset to 0. If *`run()`* is called twice or more without changing the set of objects, the second and subsequent runs will start from the end time of the previous run. To explicitly reset the time to 0, do `magic_network.t = 0*second`.

report : {None, 'stdout', 'stderr', 'graphical', function}, optional

How to report the progress of the simulation. If None, do not report progress. If stdout or stderr is specified, print the progress to stdout or stderr. If graphical, Tkinter is used to show a graphical progress bar. Alternatively, you can specify a callback `function(elapsed, complete)` which will be passed the amount of time elapsed (in seconds) and the fraction complete from 0 to 1.

report_period : *`Quantity`*

How frequently (in real time) to report progress.

profile : bool, optional

Whether to record profiling information (see *`Network.profiling_info`*). Defaults to `False`.

namespace : dict-like, optional

A namespace in which objects which do not define their own namespace will be run. If not namespace is given, the locals and globals around the run function will be used.

level : int, optional

How deep to go down the stack frame to look for the locals/global (see `namespace` argument). Only necessary under particular circumstances, e.g. when calling the run function as part of a function call or lambda expression. This is used in tests, e.g.: `assert_raises(MagicError, lambda: run(1*ms, level=3))`.

Raises

MagicError Error raised when it was not possible for Brian to safely guess the intended use. See *MagicNetwork* for more details.

See also:

Network.run(), *MagicNetwork*, *collect()*, *reinit()*, *stop()*, `clear`

start_scope()	Starts a new scope for magic functions

start_scope function

(*Shortest import*: `from brian2 import start_scope`)

`brian2.core.magic.`**`start_scope`**`()`

Starts a new scope for magic functions

All objects created before this call will no longer be automatically included by the magic functions such as *run()*.

stop()	Stops all running simulations.

stop function

(*Shortest import*: `from brian2 import stop`)

`brian2.core.magic.`**`stop`**`()`

Stops all running simulations.

See also:

Network.stop(), *run()*, *reinit()*

store([name])	Store the state of the network and all included objects.

store function

(*Shortest import*: `from brian2 import store`)

`brian2.core.magic.`**`store`**`(`*name='default'*`)`

Store the state of the network and all included objects.

Parameters **name** : str, optional

A name for the snapshot, if not specified uses `'default'`.

Objects

magic_network	Automatically constructed *MagicNetwork* of all Brian objects

magic_network object

(*Shortest import*: `from brian2 import magic_network`)

brian2.core.magic.**magic_network = MagicNetwork()**
Automatically constructed *MagicNetwork* of all Brian objects

names module

Exported members: *Nameable*

Classes

Nameable(name)	Base class to find a unique name for an object

Nameable class

(*Shortest import*: `from brian2 import Nameable`)

class brian2.core.names.**Nameable**(*name*)
Bases: *brian2.core.tracking.Trackable*

Base class to find a unique name for an object

If you specify a name explicitly, and it has already been taken, a `ValueError` is raised. You can also specify a name with a wildcard asterisk in the end, i.e. in the form `'name*'`. It will then try `name` first but if this is already specified, it will try `name_1`, `name__2`, etc. This is the default mechanism used by most core objects in Brian, e.g. *NeuronGroup* uses a default name of `'neurongroup*'`.

Parameters **name** : str

An name for the object, possibly ending in `*` to specify that variants of this name should be tried if the name (without the asterisk) is already taken. If (and only if) the name for this object has already been set, it is also possible to call the initialiser with `None` for the *name* argument. This situation can arise when a class derives from multiple classes that derive themselves from *Nameable* (e.g. *Group* and *CodeRunner*) and their initialisers are called explicitely.

Raises

ValueError If the name is already taken.

Attributes

id	A unique id for this object.
name	The unique name for this object.

Methods

assign_id()	Assign a new id to this object.

Details

id
A unique id for this object.

In contrast to names, which may be reused, the id stays unique. This is used in the dependency checking to not have to deal with the chore of comparing weak references, weak proxies and strong references.

name
The unique name for this object.

Used when generating code. Should be an acceptable variable name, i.e. starting with a letter character and followed by alphanumeric characters and _.

assign_id()
Assign a new id to this object. Under most circumstances, this method should only be called once at the creation of the object to generate a unique id. In the case of the *MagicNetwork*, however, the id should change when a new, independent set of objects is simulated.

Functions

find_name(name)	

find_name function

(*Shortest import*: `from brian2.core.names import find_name`)

brian2.core.names.**find_name**(*name*)

namespace module

Implementation of the namespace system, used to resolve the identifiers in model equations of *NeuronGroup* and *Synapses*

Exported members: *get_local_namespace()*, DEFAULT_FUNCTIONS, DEFAULT_UNITS, DEFAULT_CONSTANTS

Functions

get_local_namespace(level)	Get the surrounding namespace.

get_local_namespace function

(*Shortest import*: `from brian2 import get_local_namespace`)

brian2.core.namespace.**get_local_namespace**(*level*)
Get the surrounding namespace.

Parameters **level** : int, optional

How far to go back to get the locals/globals. Each function/method call should add `1` to

this argument, functions/method with a decorator have to add `2`.

Returns **namespace** : dict

The locals and globals at the given depth of the stack frame.

`network` module

Module defining the *Network* object, the basis of all simulation runs.

Preferences

Network preferences `core.network.default_schedule = ['start', 'groups', 'thresholds', 'synapses', 'resets', 'end']`

Default schedule used for networks that don't specify a schedule.

Exported members: *Network*, *profiling_summary()*

Classes

Network(*objs[, name])	The main simulation controller in Brian

Network class

(*Shortest import*: `from brian2 import Network`)

class `brian2.core.network.`**`Network`**(**objs*, *name='network'**)

Bases: *brian2.core.names.Nameable*

The main simulation controller in Brian

Network handles the running of a simulation. It contains a set of Brian objects that are added with *add*. The *run* method actually runs the simulation. The main run loop, determining which objects get called in what order is described in detail in the notes below. The objects in the *Network* are accesible via their names, e.g. `net['neurongroup']` would return the *NeuronGroup* with this name.

Parameters **objs** : (*BrianObject*, container), optional

A list of objects to be added to the *Network* immediately, see *add*.

name : str, optional

An explicit name, if not specified gives an automatically generated name

See also:

MagicNetwork, *run()*, *stop()*

Notes

The main run loop performs the following steps:

1. Prepare the objects if necessary, see `prepare`.
2. Determine the end time of the simulation as `t``+``duration`.

3. Determine which set of clocks to update. This will be the clock with the smallest value of `t`. If there are several with the same value, then all objects with these clocks will be updated simultaneously. Set *t* to the clock time.
4. If the `t` value of these clocks is past the end time of the simulation, stop running. If the *Network.stop()* method or the *stop()* function have been called, stop running. Set *t* to the end time of the simulation.
5. For each object whose `clock` is set to one of the clocks from the previous steps, call the `update` method. This method will not be called if the `active` flag is set to `False`. The order in which the objects are called is described below.
6. Increase *Clock.t* by *Clock.dt* for each of the clocks and return to step 2.

The order in which the objects are updated in step 4 is determined by the *Network.schedule* and the objects `when` and `order` attributes. The *schedule* is a list of string names. Each `when` attribute should be one of these strings, and the objects will be updated in the order determined by the schedule. The default schedule is `['start', 'groups', 'thresholds', 'synapses', 'resets', 'end']`. In addition to the names provided in the schedule, automatic names starting with `before_` and `after_` can be used. That means that all objects with `when=='before_start'` will be updated first, then those with `when=='start'`, `when=='after_start'`, `when=='before_groups'`, `when=='groups'` and so forth. If several objects have the same `when` attribute, then the order is determined by the `order` attribute (lower first).

Attributes

_stored_t	Stored time for the store/restore mechanism
objects	The list of objects in the Network, should not normally be modified directly.
profiling_info	The time spent in executing the various *CodeObject*s.
schedule	List of `when` slots in the order they will be updated, can be modified.
t	Current simulation time in seconds (*Quantity*)
*t_*	Current time as a float

Methods

add(*objs)	Add objects to the *Network*
check_dependencies()	
remove(*objs)	Remove an object or sequence of objects from a *Network*.

Details

_stored_t
Stored time for the store/restore mechanism

objects
The list of objects in the Network, should not normally be modified directly. Note that in a *MagicNetwork*, this attribute only contains the objects during a run: it is filled in *before_run* and emptied in *after_run*

profiling_info
The time spent in executing the various *CodeObject*s.

A list of `(name, time)` tuples, containing the name of the *CodeObject* and the total execution time

for simulations of this object (as a *Quantity* with unit `second`). The list is sorted descending with execution time.

Profiling has to be activated using the `profile` keyword in *run()* or *Network.run()*.

schedule
List of `when` slots in the order they will be updated, can be modified.

See notes on scheduling in *Network*. Note that additional `when` slots can be added, but the schedule should contain at least all of the names in the default schedule: `['start', 'groups', 'thresholds', 'synapses', 'resets', 'end']`.

The schedule can also be set to `None`, resetting it to the default schedule set by the *core.network.default_schedule* preference.

t
Current simulation time in seconds (*Quantity*)

t_
Current time as a float

add(**objs*)
Add objects to the *Network*

Parameters **objs** : (*BrianObject*, container)

The *BrianObject* or container of Brian objects to be added. Specify multiple objects, or lists (or other containers) of objects. Containers will be added recursively.

after_run()

before_run(*namespace*)
Prepares the *Network* for a run.

Objects in the *Network* are sorted into the correct running order, and their *BrianObject.before_run()* methods are called.

Parameters **namespace** : dict-like, optional

A namespace in which objects which do not define their own namespace will be run.

check_dependencies()

get_profiling_info(**args*, ***kwds*)
The only reason this is not directly implemented in *profiling_info* is to allow devices (e.g. `CPPStandaloneDevice`) to overwrite this.

remove(**objs*)
Remove an object or sequence of objects from a *Network*.

Parameters **objs** : (*BrianObject*, container)

The *BrianObject* or container of Brian objects to be removed. Specify multiple objects, or lists (or other containers) of objects. Containers will be removed recursively.

restore(*name='default'*)
Retore the state of the network and all included objects.

Parameters **name** : str, optional

The name of the snapshot to restore, if not specified uses `'default'`.

run(*duration*, *report=None*, *report_period=60*second*, *namespace=None*, *level=0*)
Runs the simulation for the given duration.

Parameters **duration** : *Quantity*

The amount of simulation time to run for.

report : {None, 'text', 'stdout', 'stderr', function}, optional

How to report the progress of the simulation. If `None`, do not report progress. If `'text'` or `'stdout'` is specified, print the progress to stdout. If `'stderr'` is specified, print the progress to stderr. Alternatively, you can specify a callback `callable(elapsed, complete, duration)` which will be passed the amount of time elapsed as a *Quantity*, the fraction complete from 0.0 to 1.0 and the total duration of the simulation (in biological time). The function will always be called at the beginning and the end (i.e. for fractions 0.0 and 1.0), regardless of the `report_period`.

report_period : *Quantity*

How frequently (in real time) to report progress.

namespace : dict-like, optional

A namespace that will be used in addition to the group-specific namespaces (if defined). If not specified, the locals and globals around the run function will be used.

profile : bool, optional

Whether to record profiling information (see *Network.profiling_info*). Defaults to `True`.

level : int, optional

How deep to go up the stack frame to look for the locals/global (see `namespace` argument). Only used by run functions that call this run function, e.g. *MagicNetwork.run()* to adjust for the additional nesting.

Notes

The simulation can be stopped by calling *Network.stop()* or the global *stop()* function.

stop()
Stops the network from running, this is reset the next time *Network.run()* is called.

store(*name='default'*)
Store the state of the network and all included objects.

Parameters **name** : str, optional

A name for the snapshot, if not specified uses `'default'`.

Tutorials and examples using this

- Example IF_curve_LIF
- Example IF_curve_Hodgkin_Huxley
- Example advanced/stochastic_odes

ProfilingSummary(net[, show])	Class to nicely display the results of profiling.

ProfilingSummary class

(*Shortest import*: `from brian2.core.network import ProfilingSummary`)

class brian2.core.network.**ProfilingSummary**(*net*, *show=None*)
Bases: object

Class to nicely display the results of profiling. Objects of this class are returned by *profiling_summary()*.

Parameters **net** : *Network*

The *Network* object to profile.

show : int, optional

The number of results to show (the longest results will be shown). If not specified, all results will be shown.

See also:

Network.profiling_info

TextReport(stream)	Helper object to report simulation progress in *Network.run()*.

TextReport class

(*Shortest import*: `from brian2.core.network import TextReport`)

class brian2.core.network.**TextReport**(*stream*)
Bases: object

Helper object to report simulation progress in *Network.run()*.

Parameters **stream** : file

The stream to write to, commonly `sys.stdout` or `sys.stderr`.

Methods

__call__(elapsed, completed, duration)	

Details

__call__(*elapsed*, *completed*, *duration*)

Functions

profiling_summary([net, show])	Returns a *ProfilingSummary* of the profiling info for a run.

profiling_summary function

(*Shortest import*: `from brian2 import profiling_summary`)

brian2.core.network.**profiling_summary**(*net=None*, *show=None*)
Returns a *ProfilingSummary* of the profiling info for a run. This object can be transformed to a string explicitly but on an interactive console simply calling *profiling_summary()* is enough since it will automatically convert the *ProfilingSummary* object.

Parameters **net** : {*Network*, None} optional

The *Network* object to profile, or *magic_network* if not specified.

show : int

The number of results to show (the longest results will be shown). If not specified, all results will be shown.

`operations` module

Exported members: *NetworkOperation*, *network_operation()*

Classes

NetworkOperation(function[, dt, clock, ...])	Object with function that is called every time step.

NetworkOperation class

(*Shortest import*: `from brian2 import NetworkOperation`)

class `brian2.core.operations.`**`NetworkOperation`**(*function*, *dt=None*, *clock=None*, *when='start'*, *order=0*)

Bases: *brian2.core.base.BrianObject*

Object with function that is called every time step.

Parameters **function** : function

The function to call every time step, should take either no arguments in which case it is called as `function()` or one argument, in which case it is called with the current *Clock* time (*Quantity*).

dt : *Quantity*, optional

The time step to be used for the simulation. Cannot be combined with the `clock` argument.

clock : *Clock*, optional

The update clock to be used. If neither a clock, nor the `dt` argument is specified, the *defaultclock* will be used.

when : str, optional

In which scheduling slot to execute the operation during a time step. Defaults to `'start'`.

order : int, optional

The priority of this operation for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

See also:

network_operation(), *Network*, *BrianObject*

Attributes

function	The function to be called each time step

Methods

run()	

Details

function
The function to be called each time step

run()

Functions

network_operation([when])	Decorator to make a function get called every time step of a simulation.

network_operation function

(*Shortest import*: `from brian2 import network_operation`)

brian2.core.operations.**network_operation**(*when=None*)
Decorator to make a function get called every time step of a simulation.

The function being decorated should either have no arguments, or a single argument which will be called with the current time `t`.

Parameters **dt** : *Quantity*, optional

The time step to be used for the simulation. Cannot be combined with the `clock` argument.

clock : *Clock*, optional

The update clock to be used. If neither a clock, nor the `dt` argument is specified, the *defaultclock* will be used.

when : str, optional

In which scheduling slot to execute the operation during a time step. Defaults to `'start'`.

order : int, optional

The priority of this operation for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

See also:

NetworkOperation, *Network*, *BrianObject*

Notes

Converts the function into a *NetworkOperation*.

If using the form:

```
@network_operations(when='end')
def f():
    ...
```

Then the arguments to network_operation must be keyword arguments.

Examples

Print something each time step: >>> from brian2 import * >>> @network_operation ... def f(): ... print('something') ... >>> net = Network(f)

Print the time each time step:

```
>>> @network_operation
... def f(t):
...   print('The time is', t)
...
>>> net = Network(f)
```

Specify a dt, etc.:

```
>>> @network_operation(dt=0.5*ms, when='end')
... def f():
...   print('This will happen at the end of each timestep.')
...
>>> net = Network(f)
```

`preferences` module

Brian global preferences are stored as attributes of a *BrianGlobalPreferences* object `prefs`.

Exported members: *PreferenceError*, *BrianPreference*, *prefs*, *brian_prefs*

Classes

BrianGlobalPreferences()	Class of the `prefs` object.

BrianGlobalPreferences class

(*Shortest import*: `from brian2.core.preferences import BrianGlobalPreferences`)

class `brian2.core.preferences.`**BrianGlobalPreferences**
Bases: `_abcoll.MutableMapping`

Class of the `prefs` object.

Used for getting/setting/validating/registering preference values. All preferences must be registered via *register_preferences*. To get or set a preference, you can either use a dictionary-based or an attribute-based interface:

```
prefs['core.default_float_dtype'] = float32
prefs.core.default_float_dtype = float32
```

Preferences can be read from files, see *load_preferences* and *read_preference_file*. Note that *load_preferences* is called automatically when Brian has finished importing.

Attributes

as_file	Get a Brian preference doc file format string for the current preferences
defaults_as_file	Get a Brian preference doc file format string for the default preferences
toplevel_categories	The toplevel preference categories

Methods

check_all_validated()	Checks that all preferences that have been set have been validated.
do_validation()	Validates preferences that have not yet been validated.
eval_pref(value)	Evaluate a string preference in the units namespace
get_documentation([basename, link_targets])	Generates a string documenting all preferences with the given `basename`.
load_preferences()	Load all the preference files, but do not validate them.
read_preference_file(file)	Reads a Brian preferences file
register_preferences(prefbasename, ...)	Registers a set of preference names, docs and validation functions.
reset_to_defaults()	Resets the parameters to their default values.

Details

as_file
Get a Brian preference doc file format string for the current preferences

defaults_as_file
Get a Brian preference doc file format string for the default preferences

toplevel_categories
The toplevel preference categories

check_all_validated()
Checks that all preferences that have been set have been validated.

Logs a warning if not. Should be called by *Network.run()* or other key Brian functions.

do_validation()
Validates preferences that have not yet been validated.

eval_pref(*value*)
Evaluate a string preference in the units namespace

get_documentation(*basename=None*, *link_targets=True*)
Generates a string documenting all preferences with the given `basename`. If no `basename` is given, all preferences are documented.

load_preferences()
Load all the preference files, but do not validate them.

Preference files are read in the following order:

1. `brian2/default_preferences` from the Brian installation directory.
2. `~/.brian/user_preferences` from the user's home directory
3. `./brian_preferences` from the current directory

Files that are missing are ignored. Preferences read at each step override preferences from previous steps.

See also:

read_preference_file

read_preference_file(*file*)
Reads a Brian preferences file

The file format for Brian preferences is a plain text file of the form:

```
a.b.c = 1
# Comment line
[a]
b.d = 2
[a.b]
e = 3
```

Blank and comment lines are ignored, all others should be of one of the following two forms:

```
key = value
[section]
```

`eval` is called on the values, so strings should be written as, e.g. `'3'` rather than `3`. The eval is called with all unit names available. Within a section, the section name is prepended to the key. So in the above example, it would give the following unvalidated dictionary:

```
{'a.b.c': 1,
 'a.b.d': 2,
 'a.b.e': 3,
 }
```

Parameters **file** : file, str

The file object or filename of the preference file.

register_preferences(*prefbasename*, *prefbasedoc*, ***prefs*)
Registers a set of preference names, docs and validation functions.

Parameters **prefbasename** : str

The base name of the preference.

prefbasedoc : str

Documentation for this base name

****prefs** : dict of (name, *BrianPreference*) pairs

The preference names to be defined. The full preference name will be `prefbasename.name`, and the *BrianPreference* value is used to define the default value, docs, and validation function.

Raises

PreferenceError If the base name is already registered.

See also:

BrianPreference

reset_to_defaults()
Resets the parameters to their default values.

BrianGlobalPreferencesView(basename, all_prefs)	A class allowing for accessing preferences in a subcategory.

BrianGlobalPreferencesView class

(*Shortest import*: `from brian2.core.preferences import BrianGlobalPreferencesView`)

class `brian2.core.preferences.`**`BrianGlobalPreferencesView`**(*basename*, *all_prefs*)
Bases: `_abcoll.MutableMapping`

A class allowing for accessing preferences in a subcategory. It forwards requests to *BrianGlobalPreferences* and provides documentation and autocompletion support for all preferences in the given category. This object is used to allow accessing preferences via attributes of the *prefs* object.

Parameters **basename** : str

The name of the preference category. Has to correspond to a key in `BrianGlobalPreferences.pref_register`.

all_prefs : *BrianGlobalPreferences*

A reference to the main object storing the preferences.

BrianPreference(default, docs[, validator, ...])	Used for defining a Brian preference.

BrianPreference class

(*Shortest import*: `from brian2 import BrianPreference`)

class `brian2.core.preferences.`**`BrianPreference`**(*default*, *docs*, *validator=None*, *representor=<built-in function repr>*)
Bases: `object`

Used for defining a Brian preference.

Parameters **default** : object

The default value.

docs : str

Documentation for the preference value.

validator : func

A function that True or False depending on whether the preference value is valid or not. If not specified, uses the *DefaultValidator* for the default value provided (check if the class is the same, and for *Quantity* objects, whether the units are consistent).

representor : func

A function that returns a string representation of a valid preference value that can be passed to `eval`. By default, uses `repr` which works in almost all cases.

DefaultValidator(value)	Default preference validator

DefaultValidator class

(*Shortest import*: `from brian2.core.preferences import DefaultValidator`)

class `brian2.core.preferences.`**`DefaultValidator`**(*value*)
Bases: `object`

Default preference validator

Used by *BrianPreference* as the default validator if none is given. First checks if the provided value is of the same class as the default value, and then if the default is a *Quantity*, checks that the units match.

Methods

__call__(value)	

Details

__call__ (*value*)

ErrorRaiser	

ErrorRaiser class

(*Shortest import*: `from brian2.core.preferences import ErrorRaiser`)

class `brian2.core.preferences.`**ErrorRaiser**
Bases: `object`

PreferenceError	Exception relating to the Brian preferences system.

PreferenceError class

(*Shortest import*: `from brian2 import PreferenceError`)

class `brian2.core.preferences.`**PreferenceError**
Bases: `exceptions.Exception`

Exception relating to the Brian preferences system.

Functions

check_preference_name(name)	Make sure that a preference name is valid.

check_preference_name function

(*Shortest import*: `from brian2.core.preferences import check_preference_name`)

`brian2.core.preferences.`**check_preference_name**(*name*)
Make sure that a preference name is valid. This currently checks that the name does not contain illegal characters and does not clash with method names such as “keys” or “items”.

Parameters **name** : str

The name to check.

Raises

PreferenceError In case the name is invalid.

parse_preference_name(name)	Split a preference name into a base and end name.

parse_preference_name function

(*Shortest import*: `from brian2.core.preferences import parse_preference_name`)

`brian2.core.preferences.`**`parse_preference_name`**(*name*)

Split a preference name into a base and end name.

Parameters **name** : str

The full name of the preference.

Returns **basename** : str

The first part of the name up to the final `.`.

endname : str

The last part of the name from the final `.` onwards.

Examples

```
>>> parse_preference_name('core.weave_compiler')
('core', 'weave_compiler')
>>> parse_preference_name('codegen.cpp.compiler')
('codegen.cpp', 'compiler')
```

Objects

brian_prefs

brian_prefs object

(*Shortest import*: `from brian2 import brian_prefs`)

`brian2.core.preferences.`**`brian_prefs = <brian2.core.preferences.ErrorRaiser object>`**

prefs	Preference categories:

prefs object

(*Shortest import*: `from brian2 import prefs`)

`brian2.core.preferences.`**`prefs = <BrianGlobalPreferences with top-level categories: “core”, “logging”, “devices”, “c`**

Preference categories:

** **core** ** Core Brian preferences

** **logging** ** Logging system preferences

**** devices **** Device preferences

**** codegen **** Code generation preferences

`spikesource` module

Exported members: *SpikeSource*

Classes

SpikeSource	A source of spikes.

SpikeSource class

(*Shortest import*: `from brian2 import SpikeSource`)

class `brian2.core.spikesource.`**SpikeSource**
Bases: `object`

A source of spikes.

An object that can be used as a source of spikes for objects such as *SpikeMonitor*, *Synapses*, etc.

The defining properties of *SpikeSource* are that it should have:

- A length that can be extracted with `len(obj)`, where the maximum spike index possible is `len(obj)-1`.
- An attribute *spikes*, an array of ints each from 0 to `len(obj)-1` with no repeats (but possibly not in sorted order). This should be updated each time step.
- A *clock* attribute, this will be used as the default clock for objects with this as a source.

spikes
An array of ints, each from 0 to `len(obj)-1` with no repeats (but possibly not in sorted order). Updated each time step.

clock
The clock on which the spikes will be updated.

`tracking` module

Exported members: *Trackable*

Classes

InstanceFollower	Keep track of all instances of classes derived from *Trackable*

InstanceFollower class

(*Shortest import*: `from brian2.core.tracking import InstanceFollower`)

class `brian2.core.tracking.`**InstanceFollower**
Bases: `object`

Keep track of all instances of classes derived from *Trackable*

The variable `__instancesets__` is a dictionary with keys which are class objects, and values which are

InstanceTrackerSet, so `__instanceset__[cls]` is a set tracking all of the instances of class `cls` (or a subclass).

Methods

add(value)	
get(cls)	

Details

add(*value*)

get(*cls*)

InstanceTrackerSet	A `set` of `weakref.ref` to all existing objects of a certain class.

InstanceTrackerSet class

(*Shortest import*: `from brian2.core.tracking import InstanceTrackerSet`)

class `brian2.core.tracking.`**InstanceTrackerSet**
Bases: `set`

A `set` of `weakref.ref` to all existing objects of a certain class.

Should not normally be directly used.

Methods

add(value)	Adds a `weakref.ref` to the `value`
remove(value)	Removes the `value` (which should be a weakref) if it is in the set

Details

add(*value*)
Adds a `weakref.ref` to the `value`

remove(*value*)
Removes the `value` (which should be a weakref) if it is in the set

Sometimes the value will have been removed from the set by `clear`, so we ignore `KeyError` in this case.

Trackable	Classes derived from this will have their instances tracked.

Trackable class

(*Shortest import*: `from brian2 import Trackable`)

class `brian2.core.tracking.`**Trackable**

Bases: object

Classes derived from this will have their instances tracked.

The classmethod __instances__() will return an *InstanceTrackerSet* of the instances of that class, and its subclasses.

variables module

Classes used to specify the type of a function, variable or common sub-expression.

Exported members: *Variable*, *Constant*, *AttributeVariable*, *ArrayVariable*, *DynamicArrayVariable*, *Subexpression*, *AuxiliaryVariable*, *VariableView*, *Variables*, *LinkedVariable*, *linked_var()*

Classes

ArrayVariable(name, unit, owner, size, device)	An object providing information about a model variable stored in an array (for

ArrayVariable class

(*Shortest import*: from brian2.core.variables import ArrayVariable)

class brian2.core.variables.**ArrayVariable**(*name*, *unit*, *owner*, *size*, *device*, *dtype=None*, *constant=False*, *scalar=False*, *read_only=False*, *dynamic=False*, *unique=False*)

Bases: *brian2.core.variables.Variable*

An object providing information about a model variable stored in an array (for example, all state variables). Most of the time *Variables.add_array* should be used instead of instantiating this class directly.

Parameters **name** : 'str'

The name of the variable. Note that this refers to the *original* name in the owning group. The same variable may be known under other names in other groups (e.g. the variable v of a *NeuronGroup* is known as v_post in a Synapse connecting to the group).

unit : *Unit*

The unit of the variable

owner : *Nameable*

The object that "owns" this variable, e.g. the *NeuronGroup* or *Synapses* object that declares the variable in its model equations.

size : int

The size of the array

device : Device

The device responsible for the memory access.

dtype : dtype, optional

The dtype used for storing the variable. If none is given, defaults to *core.default_float_dtype*.

constant : bool, optional

Whether the variable's value is constant during a run. Defaults to False.

scalar : bool, optional

Whether this array is a 1-element array that should be treated like a scalar (e.g. for a single delay value across synapses). Defaults to `False`.

read_only : bool, optional

Whether this is a read-only variable, i.e. a variable that is set internally and cannot be changed by the user. Defaults to `False`.

unique : bool, optional

Whether the values in this array are all unique. This information is only important for variables used as indices and does not have to reflect the actual contents of the array but only the possibility of non-uniqueness (e.g. synaptic indices are always unique but the corresponding pre- and post-synaptic indices are not). Defaults to `False`.

Attributes

conditional_write	Another variable, on which the write is conditioned (e.g.
device	The `Device` responsible for memory access.
size	The size of this variable.
unique	Wether all values in this arrays are necessarily unique (only relevant for index variables).

Methods

get_addressable_value(name, group)
get_addressable_value_with_unit(name, group)
get_len()
get_value()
set_conditional_write(var)
set_value(value)

Details

conditional_write
Another variable, on which the write is conditioned (e.g. a variable denoting the absence of refractoriness)

device
The `Device` responsible for memory access.

size
The size of this variable.

unique
Wether all values in this arrays are necessarily unique (only relevant for index variables).

get_addressable_value(*name*, *group*)

get_addressable_value_with_unit(*name*, *group*)

get_len()

get_value()

set_conditional_write(*var*)

set_value(*value*)

AttributeVariable(name, unit, obj, attribute)	An object providing information about a value saved as an attribute of an object

AttributeVariable class

(*Shortest import*: `from brian2.core.variables import AttributeVariable`)

class `brian2.core.variables.`**AttributeVariable**(*name*, *unit*, *obj*, *attribute*, *dtype=None*, *owner=None*, *constant=False*, *scalar=True*)

Bases: *brian2.core.variables.Variable*

An object providing information about a value saved as an attribute of an object. Instead of saving a reference to the value itself, we save the name of the attribute. This way, we get the correct value if the attribute is overwritten with a new value (e.g. in the case of `clock.t_`). Most of the time *Variables.add_attribute_variable* should be used instead of instantiating this class directly.

The object value has to be accessible by doing `getattr(obj, attribute)`. Variables of this type are considered read-only.

Parameters **name** : str

The name of the variable

unit : *Unit*

The unit of the variable

obj : object

The object storing the attribute.

attribute : str

The name of the attribute storing the variable's value. *attribute* has to be an attribute of *obj*.

dtype : `dtype`, optional

The dtype used for storing the variable. If none is given, defaults to *core.default_float_dtype*.

owner : *Nameable*, optional

The object that "owns" this variable, e.g. the *NeuronGroup* to which a `dt` value belongs (even if it is the attribute of a *Clock* object). Defaults to `None`.

constant : bool, optional

Whether the attribute's value is constant during a run. Defaults to `False`.

scalar : bool, optional

Whether the variable is a scalar value (`True`) or vector-valued, e.g. defined for every neuron (`False`). Defaults to `True`.

Attributes

attribute	The name of the attribute storing the variable's value
obj	The object storing the *attribute*

Methods

get_value()	

Details

attribute
The name of the attribute storing the variable's value

obj
The object storing the *attribute*

get_value()

AuxiliaryVariable(name, unit[, dtype, scalar])	Variable description for an auxiliary variable (most likely one that is added au

AuxiliaryVariable class

(*Shortest import*: `from brian2.core.variables import AuxiliaryVariable`)

class `brian2.core.variables.`**AuxiliaryVariable**(*name*, *unit*, *dtype=None*, *scalar=False*)
Bases: *brian2.core.variables.Variable*

Variable description for an auxiliary variable (most likely one that is added automatically to abstract code, e.g. `_cond` for a threshold condition), specifying its type and unit for code generation. Most of the time *Variables.add_auxiliary_variable* should be used instead of instantiating this class directly.

Parameters **name** : str

The name of the variable

unit : *Unit*

The unit of the variable.

dtype : `dtype`, optional

The dtype used for storing the variable. If none is given, defaults to *core.default_float_dtype*.

scalar : bool, optional

Whether the variable is a scalar value (`True`) or vector-valued, e.g. defined for every neuron (`False`). Defaults to `False`.

Methods

get_value()	

Details

get_value()

Constant(name, unit, value[, owner])	A scalar constant (e.g.

Constant class

(*Shortest import*: `from brian2.core.variables import Constant`)

class `brian2.core.variables.`**`Constant`**(*name*, *unit*, *value*, *owner=None*)
Bases: *brian2.core.variables.Variable*

A scalar constant (e.g. the number of neurons `N`). Information such as the dtype or whether this variable is a boolean are directly derived from the *value*. Most of the time *Variables.add_constant* should be used instead of instantiating this class directly.

Parameters **name** : str

The name of the variable

unit : *Unit*

The unit of the variable. Note that the variable itself (as referenced by value) should never have units attached.

value: reference to the variable value :

The value of the constant.

owner : *Nameable*, optional

The object that "owns" this variable, for constants that belong to a specific group, e.g. the `N` constant for a *NeuronGroup*. External constants will have `None` (the default value).

Attributes

value	The constant's value

Methods

get_value()

Details

`value`
The constant's value

`get_value``()`

DynamicArrayVariable(name, unit, owner, ...)	An object providing information about a model variable stored in a dynamic ar

DynamicArrayVariable class

(*Shortest import*: `from brian2.core.variables import DynamicArrayVariable`)

class brian2.core.variables.**DynamicArrayVariable**(*name*, *unit*, *owner*, *size*, *device*, *dtype=None*, *constant=False*, *constant_size=True*, *resize_along_first=False*, *scalar=False*, *read_only=False*, *unique=False*)

Bases: *brian2.core.variables.ArrayVariable*

An object providing information about a model variable stored in a dynamic array (used in *Synapses*). Most of the time *Variables.add_dynamic_array* should be used instead of instantiating this class directly.

Parameters **name** : 'str'

The name of the variable. Note that this refers to the *original* name in the owning group. The same variable may be known under other names in other groups (e.g. the variable `v` of a *NeuronGroup* is known as `v_post` in a `Synapse` connecting to the group).

unit : *Unit*

The unit of the variable

owner : *Nameable*

The object that "owns" this variable, e.g. the *NeuronGroup* or *Synapses* object that declares the variable in its model equations.

size : int or tuple of int

The (initial) size of the variable.

device : `Device`

The device responsible for the memory access.

dtype : `dtype`, optional

The dtype used for storing the variable. If none is given, defaults to *core.default_float_dtype*.

constant : bool, optional

Whether the variable's value is constant during a run. Defaults to `False`.

constant_size : bool, optional

Whether the size of the variable is constant during a run. Defaults to `True`.

scalar : bool, optional

Whether this array is a 1-element array that should be treated like a scalar (e.g. for a single delay value across synapses). Defaults to `False`.

read_only : bool, optional

Whether this is a read-only variable, i.e. a variable that is set internally and cannot be changed by the user. Defaults to `False`.

unique : bool, optional

Whether the values in this array are all unique. This information is only important for variables used as indices and does not have to reflect the actual contents of the array but only the possibility of non-uniqueness (e.g. synaptic indices are always unique but the corresponding pre- and post-synaptic indices are not). Defaults to `False`.

Attributes

constant_size	Whether the size of the variable is constant during a run.
dimensions	The number of dimensions
resize_along_first	Whether this array will be only resized along the first dimension

Methods

resize(new_size)	Resize the dynamic array.

Details

constant_size
Whether the size of the variable is constant during a run.

dimensions
The number of dimensions

resize_along_first
Whether this array will be only resized along the first dimension

resize(*new_size*)
Resize the dynamic array. Calls `self.device.resize` to do the actual resizing.

Parameters **new_size** : int or tuple of int

The new size.

LinkedVariable(group, name, variable[, index])	A simple helper class to make linking variables explicit.

LinkedVariable class

(*Shortest import*: `from brian2.core.variables import LinkedVariable`)

class `brian2.core.variables.`**LinkedVariable**(*group*, *name*, *variable*, *index=None*)
Bases: `object`

A simple helper class to make linking variables explicit. Users should use *linked_var()* instead.

Parameters **group** : *Group*

The group through which the `variable` is accessed (not necessarily the same as `variable.owner`.

name : str

The name of `variable` in `group` (not necessarily the same as `variable.name`).

variable : *Variable*

The variable that should be linked.

index : str or `ndarray`, optional

An indexing array (or the name of a state variable), providing a mapping from the entries in the link source to the link target.

Subexpression(name, unit, owner, expr, device)	An object providing information about a named subexpression in a model.

Subexpression class

(*Shortest import*: `from brian2.core.variables import Subexpression`)

class `brian2.core.variables.`**`Subexpression`**(*name*, *unit*, *owner*, *expr*, *device*, *dtype=None*, *scalar=False*)

Bases: *brian2.core.variables.Variable*

An object providing information about a named subexpression in a model. Most of the time *Variables.add_subexpression* should be used instead of instantiating this class directly.

Parameters **name** : str

The name of the subexpression.

unit : *Unit*

The unit of the subexpression.

owner : *Group*

The group to which the expression refers.

expr : str

The subexpression itself.

device : `Device`

The device responsible for the memory access.

dtype : `dtype`, optional

The dtype used for the expression. Defaults to *core.default_float_dtype*.

scalar: bool, optional :

Whether this is an expression only referring to scalar variables. Defaults to `False`

Attributes

device	The `Device` responsible for memory access
expr	The expression defining the subexpression
identifiers	The identifiers used in the expression

Methods

get_addressable_value(name, group)	
get_addressable_value_with_unit(name, group)	

Details

`device`

The `Device` responsible for memory access

expr
The expression defining the subexpression

identifiers
The identifiers used in the expression

get_addressable_value(*name*, *group*)

get_addressable_value_with_unit(*name*, *group*)

Variable(name, unit[, owner, dtype, scalar, ...])	An object providing information about model variables (including implicit variab

Variable class

(*Shortest import*: `from brian2.core.variables import Variable`)

class `brian2.core.variables.`**Variable**(*name*, *unit*, *owner=None*, *dtype=None*, *scalar=False*, *constant=False*, *read_only=False*, *dynamic=False*)

Bases: `object`

An object providing information about model variables (including implicit variables such as `t` or `xi`). This class should never be instantiated outside of testing code, use one of its subclasses instead.

Parameters **name** : 'str'

The name of the variable. Note that this refers to the *original* name in the owning group. The same variable may be known under other names in other groups (e.g. the variable `v` of a *NeuronGroup* is known as `v_post` in a `Synapse` connecting to the group).

unit : *Unit*

The unit of the variable.

owner : *Nameable*, optional

The object that "owns" this variable, e.g. the *NeuronGroup* or *Synapses* object that declares the variable in its model equations. Defaults to `None` (the value used for *Variable* objects without an owner, e.g. external *Constant*s).

dtype : `dtype`, optional

The dtype used for storing the variable. Defaults to the preference `core.default_scalar.dtype`.

scalar : bool, optional

Whether the variable is a scalar value (`True`) or vector-valued, e.g. defined for every neuron (`False`). Defaults to `False`.

constant: bool, optional :

Whether the value of this variable can change during a run. Defaults to `False`.

read_only : bool, optional

Whether this is a read-only variable, i.e. a variable that is set internally and cannot be changed by the user (this is used for example for the variable `N`, the number of neurons in a group). Defaults to `False`.

Attributes

constant	Whether the variable is constant during a run
dim	The dimensions of this variable.
dtype	The dtype used for storing the variable.
dtype_str	String representation of the numpy dtype
dynamic	Whether the variable is dynamically sized (only for non-scalars)
is_boolean	
name	The variable's name.
owner	The *Group* to which this variable belongs.
read_only	Whether the variable is read-only
scalar	Whether the variable is a scalar
unit	The variable's unit.

Methods

get_addressable_value(name, group)	Get the value (without units) of this variable in a form that can be inde
get_addressable_value_with_unit(name, group)	Get the value (with units) of this variable in a form that can be indexed
get_len()	Get the length of the value associated with the variable or `0` for a scala
get_value()	Return the value associated with the variable (without units).
get_value_with_unit()	Return the value associated with the variable (with units).
set_value(value)	Set the value associated with the variable.

Details

constant
Whether the variable is constant during a run

dim
The dimensions of this variable.

dtype
The dtype used for storing the variable.

dtype_str
String representation of the numpy dtype

dynamic
Whether the variable is dynamically sized (only for non-scalars)

is_boolean

name
The variable's name.

owner
The *Group* to which this variable belongs.

read_only
Whether the variable is read-only

scalar
Whether the variable is a scalar

unit
The variable's unit.

get_addressable_value(*name*, *group*)
Get the value (without units) of this variable in a form that can be indexed in the context of a group. For example, if a postsynaptic variable `x` is accessed in a synapse `S` as `S.x_post`, the synaptic indexing scheme can be used.

Parameters **name** : str

The name of the variable

group : *Group*

The group providing the context for the indexing. Note that this `group` is not necessarily the same as *Variable.owner*: a variable owned by a *NeuronGroup* can be indexed in a different way if accessed via a *Synapses* object.

Returns **variable** : object

The variable in an indexable form (without units).

get_addressable_value_with_unit(*name*, *group*)
Get the value (with units) of this variable in a form that can be indexed in the context of a group. For example, if a postsynaptic variable `x` is accessed in a synapse `S` as `S.x_post`, the synaptic indexing scheme can be used.

Parameters **name** : str

The name of the variable

group : *Group*

The group providing the context for the indexing. Note that this `group` is not necessarily the same as *Variable.owner*: a variable owned by a *NeuronGroup* can be indexed in a different way if accessed via a *Synapses* object.

Returns **variable** : object

The variable in an indexable form (with units).

get_len()
Get the length of the value associated with the variable or `0` for a scalar variable.

get_value()
Return the value associated with the variable (without units). This is the way variables are accessed in generated code.

get_value_with_unit()
Return the value associated with the variable (with units).

set_value(*value*)
Set the value associated with the variable.

VariableView(name, variable, group[, unit])	A view on a variable that allows to treat it as an numpy array while allowing specia

VariableView class

(*Shortest import*: `from brian2.core.variables import VariableView`)

class `brian2.core.variables.`**VariableView**(*name*, *variable*, *group*, *unit=None*)
Bases: `object`

A view on a variable that allows to treat it as an numpy array while allowing special indexing (e.g. with strings) in the context of a *Group*.

Parameters **name** : str

The name of the variable (not necessarily the same as `variable.name`).

variable : *Variable*

The variable description.

group : *Group*

The group through which the variable is accessed (not necessarily the same as `variable.owner`).

unit : *Unit*, optional

The unit to be used for the variable, should be `None` when a variable is accessed without units (e.g. when accessing `G.var_`).

Attributes

dim	The dimensions of this variable.

Methods

get_item(item[, level, namespace])	Get the value of this variable.
set_item(item, value[, level, namespace])	Set this variable.

Details

dim
The dimensions of this variable.

get_item(*item*, *level=0*, *namespace=None*)
Get the value of this variable. Called by `__getitem__`.

Parameters **item** : slice, `ndarray` or string

The index for the setting operation

level : int, optional

How much farther to go up in the stack to find the implicit namespace (if used, see `run_namespace`).

namespace : dict-like, optional

An additional namespace that is used for variable lookup (if not defined, the implicit namespace of local variables is used).

get_subexpression_with_index_array(**args*, ***kwds*)

get_with_expression(**args*, ***kwds*)
Gets a variable using a string expression. Is called by *VariableView.get_item* for statements such as `print G.v['g_syn > 0']`.

Parameters **code** : str

An expression that states a condition for elements that should be selected. Can contain references to indices, such as `i` or `j` and to state variables. For example: `'i>3 and v>0*mV'`.

level : int, optional

How much farther to go up in the stack to find the implicit namespace (if used, see `run_namespace`).

run_namespace : dict-like, optional

An additional namespace that is used for variable lookup (if not defined, the implicit namespace of local variables is used).

get_with_index_array(**args*, ***kwds*)

set_item(*item*, *value*, *level=0*, *namespace=None*)

Set this variable. This function is called by `__setitem__` but there is also a situation where it should be called directly: if the context for string-based expressions is higher up in the stack, this function allows to set the `level` argument accordingly.

Parameters **item** : slice, `ndarray` or string

The index for the setting operation

value : *`Quantity`*, `ndarray` or number

The value for the setting operation

level : int, optional

How much farther to go up in the stack to find the implicit namespace (if used, see `run_namespace`).

namespace : dict-like, optional

An additional namespace that is used for variable lookup (if not defined, the implicit namespace of local variables is used).

set_with_expression(**args*, ***kwds*)

Sets a variable using a string expression. Is called by *`VariableView.set_item`* for statements such as `S.var[:, :] = 'exp(-abs(i-j)/space_constant)*nS'`

Parameters **item** : `ndarray`

The indices for the variable (in the context of this `group`).

code : str

The code that should be executed to set the variable values. Can contain references to indices, such as `i` or `j`

check_units : bool, optional

Whether to check the units of the expression.

level : int, optional

How much farther to go up in the stack to find the implicit namespace (if used, see `run_namespace`).

run_namespace : dict-like, optional

An additional namespace that is used for variable lookup (if not defined, the implicit namespace of local variables is used).

set_with_expression_conditional(**args*, ***kwds*)
Sets a variable using a string expression and string condition. Is called by *VariableView.set_item* for statements such as `S.var['i!=j'] = 'exp(-abs(i-j)/space_constant)*nS'`

Parameters **cond** : str

The string condition for which the variables should be set.

code : str

The code that should be executed to set the variable values.

check_units : bool, optional

Whether to check the units of the expression.

level : int, optional

How much farther to go up in the stack to find the implicit namespace (if used, see `run_namespace`).

run_namespace : dict-like, optional

An additional namespace that is used for variable lookup (if not defined, the implicit namespace of local variables is used).

set_with_index_array(**args*, ***kwds*)

Variables(owner[, default_index])	A container class for storing *Variable* objects.

Variables class

(*Shortest import*: `from brian2.core.variables import Variables`)

class `brian2.core.variables.`**Variables**(*owner*, *default_index='_idx'*)
Bases: `_abcoll.Mapping`

A container class for storing *Variable* objects. Instances of this class are used as the `Group.variables` attribute and can be accessed as (read-only) dictionaries.

Parameters **owner** : *Nameable*

The object (typically a *Group*) "owning" the variables.

default_index : str, optional

The index to use for the variables (only relevant for *ArrayVariable* and *DynamicArrayVariable*). Defaults to `'_idx'`.

Attributes

indices	A dictionary given the index name for every array name
owner	A reference to the *Group* owning these variables

Methods

add_arange(name, size[, start, dtype, ...])	Add an array, initialized with a range of integers.

Table 6.169 – continued from previous page

add_array(name, unit, size[, values, dtype, ...])	Add an array (initialized with zeros).
add_attribute_variable(name, unit, obj, ...)	Add a variable stored as an attribute of an object.
add_auxiliary_variable(name, unit[, dtype, ...])	Add an auxiliary variable (most likely one that is added automatically to a
add_constant(name, unit, value)	Add a scalar constant (e.g.
add_dynamic_array(name, unit, size[, ...])	Add a dynamic array.
add_reference(name, group[, varname, index])	Add a reference to a variable defined somewhere else (possibly under a dif
add_references(group, varnames[, index])	Add all *Variable* objects from a name to *Variable* mapping with the
add_referred_subexpression(name, group, ...)	
add_subexpression(name, unit, expr[, dtype, ...])	Add a named subexpression.
create_clock_variables(clock[, prefix])	Convenience function to add the `t` and `dt` attributes of a `clock`.

Details

indices
A dictionary given the index name for every array name

owner
A reference to the *Group* owning these variables

add_arange(*name*, *size*, *start=0*, *dtype=<type 'numpy.int32'>*, *constant=True*, *read_only=True*, *unique=True*, *index=None*)
Add an array, initialized with a range of integers.

Parameters **name** : str

The name of the variable.

size : int

The size of the array.

start : int

The start value of the range.

dtype : `dtype`, optional

The dtype used for storing the variable. If none is given, defaults to `np.int32`.

constant : bool, optional

Whether the variable's value is constant during a run. Defaults to `True`.

read_only : bool, optional

Whether this is a read-only variable, i.e. a variable that is set internally and cannot be changed by the user. Defaults to `True`.

index : str, optional

The index to use for this variable. Defaults to `Variables.default_index`.

unique : bool, optional

See *ArrayVariable*. Defaults to `True` here.

add_array(*name*, *unit*, *size*, *values=None*, *dtype=None*, *constant=False*, *read_only=False*, *scalar=False*, *unique=False*, *index=None*)
Add an array (initialized with zeros).

Parameters **name** : str

The name of the variable.

unit : *Unit*

The unit of the variable

size : int

The size of the array.

values : ndarray, optional

The values to initalize the array with. If not specified, the array is initialized to zero.

dtype : dtype, optional

The dtype used for storing the variable. If none is given, defaults to *core.default_float_dtype*.

constant : bool, optional

Whether the variable's value is constant during a run. Defaults to `False`.

scalar : bool, optional

Whether this is a scalar variable. Defaults to `False`, if set to `True`, also implies that `size()` equals 1.

read_only : bool, optional

Whether this is a read-only variable, i.e. a variable that is set internally and cannot be changed by the user. Defaults to `False`.

index : str, optional

The index to use for this variable. Defaults to `Variables.default_index`.

unique : bool, optional

See *ArrayVariable*. Defaults to `False`.

add_attribute_variable(*name*, *unit*, *obj*, *attribute*, *dtype=None*, *constant=False*, *scalar=True*)

Add a variable stored as an attribute of an object.

Parameters **name** : str

The name of the variable

unit : *Unit*

The unit of the variable

obj : object

The object storing the attribute.

attribute : str

The name of the attribute storing the variable's value. `attribute` has to be an attribute of `obj`.

dtype : dtype, optional

The dtype used for storing the variable. If none is given, uses the type of `obj.attribute` (which have to exist).

constant : bool, optional

Whether the attribute's value is constant during a run. Defaults to `False`.

scalar : bool, optional

Whether the variable is a scalar value (`True`) or vector-valued, e.g. defined for every neuron (`False`). Defaults to `True`.

add_auxiliary_variable(*name*, *unit*, *dtype=None*, *scalar=False*)
Add an auxiliary variable (most likely one that is added automatically to abstract code, e.g. `_cond` for a threshold condition), specifying its type and unit for code generation.

Parameters **name** : str

The name of the variable

unit : *Unit*

The unit of the variable.

dtype : dtype, optional

The dtype used for storing the variable. If none is given, defaults to *core.default_float_dtype*.

scalar : bool, optional

Whether the variable is a scalar value (`True`) or vector-valued, e.g. defined for every neuron (`False`). Defaults to `False`.

add_constant(*name*, *unit*, *value*)
Add a scalar constant (e.g. the number of neurons `N`).

Parameters **name** : str

The name of the variable

unit : *Unit*

The unit of the variable. Note that the variable itself (as referenced by value) should never have units attached.

value: reference to the variable value :

The value of the constant.

add_dynamic_array(*name*, *unit*, *size*, *values=None*, *dtype=None*, *constant=False*, *constant_size=True*, *resize_along_first=False*, *read_only=False*, *unique=False*, *scalar=False*, *index=None*)
Add a dynamic array.

Parameters **name** : str

The name of the variable.

unit : *Unit*

The unit of the variable

size : int or tuple of int

The (inititial) size of the array.

values : ndarray, optional

The values to initalize the array with. If not specified, the array is initialized to zero.

dtype : dtype, optional

The dtype used for storing the variable. If none is given, defaults to *core.default_float_dtype*.

constant : bool, optional

Whether the variable's value is constant during a run. Defaults to `False`.

constant_size : bool, optional

Whether the size of the variable is constant during a run. Defaults to `True`.

scalar : bool, optional

Whether this is a scalar variable. Defaults to `False`, if set to `True`, also implies that `size()` equals 1.

read_only : bool, optional

Whether this is a read-only variable, i.e. a variable that is set internally and cannot be changed by the user. Defaults to `False`.

index : str, optional

The index to use for this variable. Defaults to `Variables.default_index`.

unique : bool, optional

See *DynamicArrayVariable*. Defaults to `False`.

add_reference(*name*, *group*, *varname=None*, *index=None*)
Add a reference to a variable defined somewhere else (possibly under a different name). This is for example used in *Subgroup* and *Synapses* to refer to variables in the respective *NeuronGroup*.

Parameters **name** : str

The name of the variable (in this group, possibly a different name from `var.name`).

group : *Group*

The group from which `var()` is referenced

varname : str, optional

The variable to refer to. If not given, defaults to `name`.

index : str, optional

The index that should be used for this variable (defaults to `Variables.default_index`).

add_references(*group*, *varnames*, *index=None*)
Add all *Variable* objects from a name to *Variable* mapping with the same name as in the original mapping.

Parameters **group** : *Group*

The group from which the `variables` are referenced

varnames : iterable of str

The variables that should be referred to in the current group

index : str, optional

The index to use for all the variables (defaults to `Variables.default_index`)

add_referred_subexpression(*name*, *group*, *subexpr*, *index*)

add_subexpression(*name*, *unit*, *expr*, *dtype=None*, *scalar=False*, *index=None*)
Add a named subexpression.

Parameters **name** : str

The name of the subexpression.

unit : *Unit*

The unit of the subexpression.

expr : str

The subexpression itself.

dtype : dtype, optional

The dtype used for the expression. Defaults to *core.default_float_dtype*.

scalar : bool, optional

Whether this is an expression only referring to scalar variables. Defaults to `False`

index : str, optional

The index to use for this variable. Defaults to `Variables.default_index`.

create_clock_variables(*clock*, *prefix=''*)

Convenience function to add the `t` and `dt` attributes of a `clock`.

Parameters **clock** : *Clock*

The clock that should be used for `t` and `dt`. Note that the actual attributes referred to are `t_` and `dt_`, i.e. the unitless values.

prefix : str, optional

A prefix for the variable names. Used for example in monitors to not confuse the dynamic array of recorded times with the current time in the recorded group.

Functions

get_dtype(obj)	Helper function to return the `numpy.dtype` of an arbitrary object.

get_dtype function

(*Shortest import*: `from brian2.core.variables import get_dtype`)

brian2.core.variables.**get_dtype**(*obj*)

Helper function to return the `numpy.dtype` of an arbitrary object.

Parameters **obj** : object

Any object (but typically some kind of number or array).

Returns **dtype** : `numpy.dtype`

The type of the given object.

get_dtype_str(val)	Returns canonical string representation of the dtype of a value or dtype

get_dtype_str function

(*Shortest import*: `from brian2.core.variables import get_dtype_str`)

brian2.core.variables.**get_dtype_str**(*val*)

Returns canonical string representation of the dtype of a value or dtype

Returns **dtype_str** : str

The numpy dtype name

linked_var(group_or_variable[, name, index])	Represents a link target for setting a linked variable.

linked_var function

(*Shortest import*: `from brian2 import linked_var`)

`brian2.core.variables.`**`linked_var`**(*group_or_variable*, *name=None*, *index=None*)

Represents a link target for setting a linked variable.

Parameters **group_or_variable** : *NeuronGroup* or *VariableView*

Either a reference to the target *NeuronGroup* (e.g. `G`) or a direct reference to a *VariableView* object (e.g. `G.v`). In case only the group is specified, `name` has to be specified as well.

name : str, optional

The name of the target variable, necessary if `group_or_variable` is a *NeuronGroup*.

index : str or `ndarray`, optional

An indexing array (or the name of a state variable), providing a mapping from the entries in the link source to the link target.

Examples

```
>>> from brian2 import *
>>> G1 = NeuronGroup(10, 'dv/dt = -v / (10*ms) : volt')
>>> G2 = NeuronGroup(10, 'v : volt (linked)')
>>> G2.v = linked_var(G1, 'v')
>>> G2.v = linked_var(G1.v)  # equivalent
```

variables_by_owner(variables, owner)

variables_by_owner function

(*Shortest import*: `from brian2.core.variables import variables_by_owner`)

`brian2.core.variables.`**`variables_by_owner`**(*variables*, *owner*)

6.4.3 devices package

Package providing the "devices" infrastructure.

`device` module

Module containing the *Device* base class as well as the *RuntimeDevice* implementation and some helper functions to access/set devices.

Exported members: *Device*, *RuntimeDevice*, *get_device()*, *set_device()*, `all_devices`, *restore_device*, *device*

Classes

CurrentDeviceProxy	Method proxy for access to the currently active device

CurrentDeviceProxy class

(*Shortest import*: `from brian2.devices.device import CurrentDeviceProxy`)

class `brian2.devices.device.`**`CurrentDeviceProxy`**

Bases: `object`

Method proxy for access to the currently active device

Device()	Base Device object.

Device class

(*Shortest import*: `from brian2.devices import Device`)

class `brian2.devices.device.`**`Device`**

Bases: `object`

Base Device object.

Attributes

network_schedule	The network schedule that this device supports.

Methods

activate()	Called when this device is set as the current device.
add_array(var)	Add an array to this device.
build(**kwds)	For standalone projects, called when the project is ready to be built.
code_object(owner, name, abstract_code, ...)	
code_object_class([codeobj_class])	
fill_with_array(var, arr)	Fill an array with the values given in another array.
get_array_name(var[, access_data])	Return a globally unique name for `var()`.
get_len(array)	Return the length of the array.
init_with_arange(var, start)	Initialize an array with an integer range.
init_with_zeros(var)	Initialize an array with zeros.
insert_code(slot, code)	Insert code directly into a given slot in the device.
insert_device_code(slot, code)	
reinit()	Reinitialize the device.
resize(var, new_size)	Resize a `DynamicArrayVariable`.
resize_along_first(var, new_size)	
spike_queue(source_start, source_end)	Create and return a new `SpikeQueue` for this *Device*.

Details

network_schedule
The network schedule that this device supports. If the device only supports a specific, fixed schedule, it has to set this attribute to the respective schedule (see *Network.schedule* for details). If it supports arbitrary schedules, it should be set to None (the default).

activate()
Called when this device is set as the current device.

add_array(*var*)
Add an array to this device.

Parameters **var** : ArrayVariable

The array to add.

build(***kwds*)
For standalone projects, called when the project is ready to be built. Does nothing for runtime mode.

code_object(*owner*, *name*, *abstract_code*, *variables*, *template_name*, *variable_indices*, *codeobj_class=None*, *template_kwds=None*, *override_conditional_write=None*)

code_object_class(*codeobj_class=None*)

fill_with_array(*var*, *arr*)
Fill an array with the values given in another array.

Parameters **var** : ArrayVariable

The array to fill.

arr : ndarray

The array values that should be copied to var().

get_array_name(*var*, *access_data=True*)
Return a globally unique name for var().

Parameters **access_data** : bool, optional

For DynamicArrayVariable objects, specifying True here means the name for the underlying data is returned. If specifying False, the name of object itself is returned (e.g. to allow resizing).

Returns **name** : str

The name for var().

get_len(*array*)
Return the length of the array.

Parameters **array** : ArrayVariable

The array for which the length is requested.

Returns **l** : int

The length of the array.

init_with_arange(*var*, *start*)
Initialize an array with an integer range.

Parameters **var** : ArrayVariable

The array to fill with the integer range.

start : int

The start value for the integer range

init_with_zeros(*var*)
Initialize an array with zeros.

Parameters **var** : ArrayVariable

The array to initialize with zeros.

insert_code(*slot*, *code*)
Insert code directly into a given slot in the device. By default does nothing.

insert_device_code(*slot*, *code*)

reinit()
Reinitialize the device. For standalone devices, clears all the internal state of the device.

resize(*var*, *new_size*)
Resize a DynamicArrayVariable.

Parameters **var** : DynamicArrayVariable

The variable that should be resized.

new_size : int

The new size of the variable

resize_along_first(*var*, *new_size*)

spike_queue(*source_start*, *source_end*)
Create and return a new SpikeQueue for this *Device*.

Parameters **source_start** : int

The start index of the source group (necessary for subgroups)

source_end : int

The end index of the source group (necessary for subgroups)

Dummy	Dummy object

Dummy class

(*Shortest import*: from brian2.devices.device import Dummy)

class brian2.devices.device.**Dummy**
Bases: object

Dummy object

Methods

__call__(*args, **kwds)	

Details

__call__(**args*, ***kwds*)

RuntimeDevice()	The default device used in Brian, state variables are stored as numpy arrays in memory.

RuntimeDevice class

(*Shortest import*: `from brian2.devices import RuntimeDevice`)

class `brian2.devices.device.`**RuntimeDevice**
Bases: *brian2.devices.device.Device*

The default device used in Brian, state variables are stored as numpy arrays in memory.

Attributes

arrays	Mapping from `Variable` objects to numpy arrays (or `DynamicArray` objects).

Methods

add_array(var)
fill_with_array(var, arr)
get_array_name(var[, access_data])
get_value(var[, access_data])
init_with_arange(var, start)
init_with_array(var, arr)
init_with_zeros(var)
resize(var, new_size)
resize_along_first(var, new_size)
set_value(var, value)
spike_queue(source_start, source_end)

Details

arrays
Mapping from `Variable` objects to numpy arrays (or `DynamicArray` objects). Arrays in this dictionary will disappear as soon as the last reference to the `Variable` object used as a key is gone

add_array(*var*)

fill_with_array(*var*, *arr*)

get_array_name(*var*, *access_data=True*)

get_value(*var*, *access_data=True*)

init_with_arange(*var*, *start*)

init_with_array(*var*, *arr*)

init_with_zeros(*var*)

resize(*var*, *new_size*)

resize_along_first(*var*, *new_size*)

set_value(*var*, *value*)

spike_queue(*source_start*, *source_end*)

Functions

auto_target()	Automatically chose a code generation target (invoked when the *codegen.target* preference is set to `'auto'`.

auto_target function

(*Shortest import*: `from brian2.devices.device import auto_target`)

brian2.devices.device.**auto_target**()
Automatically chose a code generation target (invoked when the *codegen.target* preference is set to `'auto'`. Caches its result so it only does the check once. Prefers weave > cython > numpy.

Returns **target** : class derived from *CodeObject*

The target to use

get_default_codeobject_class([pref])	Returns the default *CodeObject* class from the preferences.

get_default_codeobject_class function

(*Shortest import*: `from brian2.devices.device import get_default_codeobject_class`)

brian2.devices.device.**get_default_codeobject_class**(*pref='codegen.target'*)
Returns the default *CodeObject* class from the preferences.

get_device()	Gets the actve *Device* object

get_device function

(*Shortest import*: `from brian2 import get_device`)

brian2.devices.device.**get_device**()
Gets the actve *Device* object

restore_device()	

restore_device function

(*Shortest import*: `from brian2.devices import restore_device`)

brian2.devices.device.**restore_device**()

set_device(device)	Sets the active *Device* object

set_device function

(*Shortest import*: `from brian2 import set_device`)

`brian2.devices.device.`**`set_device`**(*device*)
Sets the active *Device* object

Objects

active_device	The default device used in Brian, state variables are stored as numpy arrays in memory.

active_device object

(*Shortest import*: `from brian2.devices.device import active_device`)

`brian2.devices.device.`**`active_device`** **= <brian2.devices.device.RuntimeDevice object>**
The default device used in Brian, state variables are stored as numpy arrays in memory.

device	Proxy object to access methods of the current device

device object

(*Shortest import*: `from brian2 import device`)

`brian2.devices.device.`**`device`** **= <brian2.devices.device.CurrentDeviceProxy object>**
Proxy object to access methods of the current device

runtime_device	The default device used in Brian, state variables are stored as numpy arrays in memory.

runtime_device object

(*Shortest import*: `from brian2.devices.device import runtime_device`)

`brian2.devices.device.`**`runtime_device`** **= <brian2.devices.device.RuntimeDevice object>**
The default device used in Brian, state variables are stored as numpy arrays in memory.

Subpackages

cpp_standalone package

Package implementing the C++ "standalone" `Device` and *CodeObject*.

`codeobject` module Module implementing the C++ "standalone" *CodeObject*

Exported members: *CPPStandaloneCodeObject*

Classes

CPPStandaloneCodeObject(owner, code, ...[, name])	C++ standalone code object

CPPStandaloneCodeObject class (*Shortest import*: `from brian2.devices.cpp_standalone import CPPStandaloneCodeObject`)

class `brian2.devices.cpp_standalone.codeobject.`**`CPPStandaloneCodeObject`**(*owner*, *code*, *variables*, *variable_indices*, *template_name*, *template_source*, *name='codeobject*'*)

Bases: *brian2.codegen.codeobject.CodeObject*

C++ standalone code object

The `code` should be a *MultiTemplate* object with two macros defined, `main` (for the main loop code) and `support_code` for any support code (e.g. function definitions).

Methods

__call__(**kwds)	
run()	

Details

`__call__`(***kwds*)

`run`()

Functions

openmp_pragma(pragma_type)	

openmp_pragma function (*Shortest import*: `from brian2.devices.cpp_standalone.codeobject import openmp_pragma`)

`brian2.devices.cpp_standalone.codeobject.`**`openmp_pragma`**(*pragma_type*)

`device` module Module implementing the C++ "standalone" device.

Classes

CPPStandaloneDevice()	The `Device` used for C++ standalone simulations.

CPPStandaloneDevice class (*Shortest import*: `from brian2.devices.cpp_standalone.device import CPPStandaloneDevice`)

class `brian2.devices.cpp_standalone.device.`**`CPPStandaloneDevice`**

Bases: *brian2.devices.device.Device*

The `Device` used for C++ standalone simulations.

Attributes

arange_arrays	Dictionary of all arrays to be filled with numbers (mapping
arrays	Dictionary mapping `ArrayVariable` objects to their globally
dynamic_arrays	List of all dynamic arrays
dynamic_arrays_2d	Dictionary mapping `DynamicArrayVariable` objects with 2 dimensions
has_been_run	Whether the simulation has been run
static_arrays	Dict of all static saved arrays
zero_arrays	List of all arrays to be filled with zeros

Methods

add_array(var)	
build([directory, compile, run, debug, ...])	Build the project
check_openmp_compatible(nb_threads)	
code_object(owner, name, abstract_code, ...)	
code_object_class([codeobj_class])	
compile_source(directory, compiler, debug, ...)	
copy_source_files(writer, directory)	
fill_with_array(var, arr)	
find_synapses()	
generate_codeobj_source(writer)	
generate_main_source(writer)	
generate_makefile(writer, compiler, native, ...)	
generate_network_source(writer, compiler)	
generate_objects_source(writer, ...)	
generate_run_source(writer)	
generate_synapses_classes_source(writer)	
get_array_filename(var[, basedir])	Return a file name for a variable.
get_array_name(var[, access_data])	Return a globally unique name for `var()`.
get_value(var[, access_data])	
init_with_arange(var, start)	
init_with_array(var, arr)	
init_with_zeros(var)	
insert_code(slot, code)	Insert code directly into main.cpp
network_get_profiling_info(net)	
network_restore(net[, name])	
network_run(net, duration[, report, ...])	
network_store(net[, name])	
reinit()	
resize(var, new_size)	
run(directory, with_output, run_args)	
run_function(name[, include_in_parent])	Context manager to divert code into a function
static_array(name, arr)	
variableview_get_subexpression_with_index_array(...)	
variableview_get_with_expression(...[, ...])	
variableview_set_with_index_array(...)	
write_static_arrays(directory)	

Details

arange_arrays
Dictionary of all arrays to be filled with numbers (mapping `ArrayVariable` objects to start value)

arrays
Dictionary mapping `ArrayVariable` objects to their globally unique name

dynamic_arrays
List of all dynamic arrays Dictionary mapping `DynamicArrayVariable` objects with 1 dimension to their globally unique name

dynamic_arrays_2d
Dictionary mapping `DynamicArrayVariable` objects with 2 dimensions to their globally unique name

has_been_run
Whether the simulation has been run

static_arrays
Dict of all static saved arrays

zero_arrays
List of all arrays to be filled with zeros

add_array(*var*)

build(*directory='output'*, *compile=True*, *run=True*, *debug=False*, *clean=True*, *with_output=True*, *native=True*, *additional_source_files=None*, *run_args=None*, ***kwds*)
Build the project

TODO: more details

Parameters **directory** : str

The output directory to write the project to, any existing files will be overwritten.

compile : bool

Whether or not to attempt to compile the project

run : bool

Whether or not to attempt to run the built project if it successfully builds.

debug : bool

Whether to compile in debug mode.

with_output : bool

Whether or not to show the `stdout` of the built program when run.

native : bool

Whether or not to compile for the current machine's architecture (best for speed, but not portable)

clean : bool

Whether or not to clean the project before building

additional_source_files : list of str

A list of additional `.cpp` files to include in the build.

check_openmp_compatible(*nb_threads*)

code_object(*owner*, *name*, *abstract_code*, *variables*, *template_name*, *variable_indices*, *codeobj_class=None*, *template_kwds=None*, *override_conditional_write=None*)

code_object_class(*codeobj_class=None*)

compile_source(*directory*, *compiler*, *debug*, *clean*, *native*)

copy_source_files(*writer*, *directory*)

fill_with_array(*var*, *arr*)

find_synapses()

generate_codeobj_source(*writer*)

generate_main_source(*writer*)

generate_makefile(*writer*, *compiler*, *native*, *compiler_flags*, *linker_flags*, *nb_threads*)

generate_network_source(*writer*, *compiler*)

generate_objects_source(*writer*, *arange_arrays*, *synapses*, *static_array_specs*, *networks*)

generate_run_source(*writer*)

generate_synapses_classes_source(*writer*)

get_array_filename(*var*, *basedir='results'*)
Return a file name for a variable.

Parameters **var** : `ArrayVariable`

The variable to get a filename for.

basedir : str

The base directory for the filename, defaults to `'results'`.

Returns :

———- :

filename : str

A filename of the form `'results/'+varname+'_'+str(hash(varname))`, where varname is the name returned by *get_array_name*.

Notes

The reason that the filename is not simply `'results/' + varname` is that this could lead to file names that are not unique in file systems that are not case sensitive (e.g. on Windows).

get_array_name(*var*, *access_data=True*)
Return a globally unique name for `var()`.

Parameters **access_data** : bool, optional

For `DynamicArrayVariable` objects, specifying `True` here means the name for the underlying data is returned. If specifying `False`, the name of object itself is returned (e.g. to allow resizing).

get_value(*var*, *access_data=True*)

init_with_arange(*var*, *start*)

init_with_array(*var*, *arr*)

init_with_zeros(*var*)

insert_code(*slot*, *code*)
Insert code directly into main.cpp

network_get_profiling_info(*net*)

network_restore(*net*, *name='default'*)

network_run(*net*, *duration*, *report=None*, *report_period=10. * second*, *namespace=None*, *profile=True*, *level=0*, ***kwds*)

network_store(*net*, *name='default'*)

reinit()

resize(*var*, *new_size*)

run(*directory*, *with_output*, *run_args*)

run_function(*name*, *include_in_parent=True*)
Context manager to divert code into a function

Code that happens within the scope of this context manager will go into the named function.

Parameters **name** : str

The name of the function to divert code into.

include_in_parent : bool

Whether or not to include a call to the newly defined function in the parent context.

static_array(*name*, *arr*)

variableview_get_subexpression_with_index_array(*variableview*, *item*, *level=0*, *run_namespace=None*)

variableview_get_with_expression(*variableview*, *code*, *level=0*, *run_namespace=None*)

variableview_set_with_index_array(*variableview*, *item*, *value*, *check_units*)

write_static_arrays(*directory*)

CPPStandaloneSimpleDevice() **Methods**

CPPStandaloneSimpleDevice class (*Shortest import*: `from brian2.devices.cpp_standalone.device import CPPStandaloneSimpleDevice`)

class `brian2.devices.cpp_standalone.device.`**CPPStandaloneSimpleDevice**
Bases: *brian2.devices.cpp_standalone.device.CPPStandaloneDevice*

Methods

---

network_run(net, duration[, report, ...])

Details

network_run(*net*, *duration*, *report=None*, *report_period=10. * second*, *namespace=None*, *profile=True*, *level=0*, ***kwds*)

CPPWriter(project_dir) **Methods**

CPPWriter class (*Shortest import*: `from brian2.devices.cpp_standalone.device import CPPWriter`)

class `brian2.devices.cpp_standalone.device.`**CPPWriter**(*project_dir*)
Bases: `object`

Methods

write(filename, contents)

Details

write(*filename*, *contents*)

RunFunctionContext(name, include_in_parent)

RunFunctionContext class (*Shortest import*: `from brian2.devices.cpp_standalone.device import RunFunctionContext`)

class `brian2.devices.cpp_standalone.device.`**RunFunctionContext**(*name*, *include_in_parent*)
Bases: `object`

Functions

freeze(code, ns)

freeze function (*Shortest import*: `from brian2.devices.cpp_standalone.device import freeze`)

`brian2.devices.cpp_standalone.device.`**freeze**(*code*, *ns*)

invert_dict(x)

invert_dict function (*Shortest import*: `from brian2.devices.cpp_standalone.device import invert_dict`)

`brian2.devices.cpp_standalone.device.`**invert_dict**(*x*)

Objects

cpp_standalone_device	The `Device` used for C++ standalone simulations.

cpp_standalone_device object (*Shortest import*: `from brian2.devices.cpp_standalone import cpp_standalone_device`)

`brian2.devices.cpp_standalone.device.`**cpp_standalone_device = <brian2.devices.cpp_standalone.device.CPP**

The `Device` used for C++ standalone simulations.

cpp_standalone_simple_device

cpp_standalone_simple_device object (*Shortest import*: `from brian2.devices.cpp_standalone.device import cpp_standalone_simple_device`)

`brian2.devices.cpp_standalone.device.`**cpp_standalone_simple_device = <brian2.devices.cpp_standalone.d**

6.4.4 equations package

Module handling equations and "code strings", expressions or statements, used for example for the reset and threshold definition of a neuron.

Exported members: *Equations*, *Expression*, *Statements*

codestrings module

Module defining *CodeString*, a class for a string of code together with information about its namespace. Only serves as a parent class, its subclasses *Expression* and *Statements* are the ones that are actually used.

Exported members: *Expression*, *Statements*

Classes

CodeString(code)	A class for representing "code strings", i.e.

CodeString class

(*Shortest import*: `from brian2.equations.codestrings import CodeString`)

class `brian2.equations.codestrings.`**CodeString**(*code*)

Bases: `object`

A class for representing "code strings", i.e. a single Python expression or a sequence of Python statements.

Parameters **code** : str

The code string, may be an expression or a statement(s) (possibly multi-line).

Expression(code)	Class for representing an expression.

Expression class

(*Shortest import*: `from brian2 import Expression`)

class `brian2.equations.codestrings.`**Expression**(*code*)

Bases: *brian2.equations.codestrings.CodeString*

Class for representing an expression.

Parameters **code** : str

The expression. Note that the expression has to be written in a form that is parseable by sympy.

Attributes

stochastic_variables	Stochastic variables in this expression

Methods

split_stochastic()	Split the expression into a stochastic and non-stochastic part.

Details

stochastic_variables
Stochastic variables in this expression

split_stochastic()
Split the expression into a stochastic and non-stochastic part.

Splits the expression into a tuple of one *Expression* objects f (the non-stochastic part) and a dictionary mapping stochastic variables to *Expression* objects. For example, an expression of the form `f + g * xi_1 + h * xi_2` would be returned as: `(f, {'xi_1': g, 'xi_2': h})` Note that the *Expression* objects for the stochastic parts do not include the stochastic variable itself.

Returns **(f, d)** : (*Expression*, dict)

A tuple of an *Expression* object and a dictionary, the first expression being the non-stochastic part of the equation and the dictionary mapping stochastic variables (`xi` or starting with `xi_`) to *Expression* objects. If no stochastic variable is present in the code string, a tuple `(self, None)` will be returned with the unchanged *Expression* object.

Statements(code)	Class for representing statements.

Statements class

(*Shortest import*: `from brian2 import Statements`)

class `brian2.equations.codestrings.`**Statements**(*code*)
Bases: *brian2.equations.codestrings.CodeString*

Class for representing statements.

Parameters **code** : str

The statement or statements. Several statements can be given as a multi-line string or separated by semicolons.

Notes

Currently, the implementation of this class does not add anything to *CodeString*, but it should be used instead of that class for clarity and to allow for future functionality that is only relevant to statements and not to expressions.

`equations` module

Differential equations for Brian models.

Exported members: *Equations*

Classes

EquationError	Exception type related to errors in an equation definition.

EquationError class

(*Shortest import*: `from brian2.equations.equations import EquationError`)

class `brian2.equations.equations.`**EquationError**
Bases: `exceptions.Exception`

Exception type related to errors in an equation definition.

Equations(eqns, **kwds)	Container that stores equations from which models can be created.

Equations class

(*Shortest import*: `from brian2 import Equations`)

class `brian2.equations.equations.`**Equations**(*eqns*, ***kwds*)
Bases: `_abcoll.Mapping`

Container that stores equations from which models can be created.

String equations can be of any of the following forms:

1. `dx/dt = f : unit (flags)` (differential equation)
2. `x = f : unit (flags)` (equation)
3. `x : unit (flags)` (parameter)

String equations can span several lines and contain Python-style comments starting with #

Parameters **eqs** : `str` or list of *SingleEquation* objects

A multiline string of equations (see above) – for internal purposes also a list of *SingleEquation* objects can be given. This is done for example when adding new equations to implement the refractory mechanism. Note that in this case the variable names are not checked to allow for "internal names", starting with an underscore.

kwds: keyword arguments :

Keyword arguments can be used to replace variables in the equation string. Arguments have to be of the form `varname=replacement`, where `varname` has to correspond to a variable name in the given equation. The replacement can be either a string (replacing a name with a new name, e.g. `tau='tau_e'`) or a value (replacing the variable name with the value, e.g. `tau=tau_e` or `tau=10*ms`).

Attributes

diff_eq_expressions	A list of (variable name, expression) tuples of all differential equations.
diff_eq_names	All differential equation names.
eq_expressions	A list of (variable name, expression) tuples of all equations.
eq_names	All equation names (including subexpressions).
identifier_checks	A set of functions that are used to check identifiers (class attribute).
identifiers	Set of all identifiers used in the equations, excluding the variables defined in the equations
is_stochastic	Whether the equations are stochastic.
names	All variable names defined in the equations.
ordered	A list of all equations, sorted according to the order in which they should be updated
parameter_names	All parameter names.
stochastic_type	Returns the type of stochastic differential equations (additivive or multiplicative).
stochastic_variables	
subexpr_names	All subexpression names.
substituted_expressions	Return a list of `(varname, expr)` tuples, containing all differential equations with all the sul
units	Dictionary of all internal variables and their corresponding units.

Methods

check_flags(allowed_flags)	Check the list of flags.
check_identifier(identifier)	Perform all the registered checks.
check_identifiers()	Check all identifiers for conformity with the rules.
check_units(group[, run_namespace, level])	Check all the units for consistency.
register_identifier_check(func)	Register a function for checking identifiers.

Details

diff_eq_expressions
A list of (variable name, expression) tuples of all differential equations.

diff_eq_names
All differential equation names.

eq_expressions
A list of (variable name, expression) tuples of all equations.

eq_names
All equation names (including subexpressions).

identifier_checks
A set of functions that are used to check identifiers (class attribute). Functions can be registered with the static method *Equations.register_identifier_check* and will be automatically used when checking identifiers

identifiers
Set of all identifiers used in the equations, excluding the variables defined in the equations

is_stochastic
Whether the equations are stochastic.

names
All variable names defined in the equations.

ordered
A list of all equations, sorted according to the order in which they should be updated

parameter_names
All parameter names.

stochastic_type
Returns the type of stochastic differential equations (additivive or multiplicative). The system is only classified as `additive` if *all* equations have only additive noise (or no noise).

Returns type : str

Either `None` (no noise variables), `'additive'` (factors for all noise variables are independent of other state variables or time), `'multiplicative'` (at least one of the noise factors depends on other state variables and/or time).

stochastic_variables

subexpr_names
All subexpression names.

substituted_expressions
Return a list of `(varname, expr)` tuples, containing all differential equations with all the subexpression variables substituted with the respective expressions.

Returns expr_tuples : list of (str, `CodeString`)

A list of `(varname, expr)` tuples, where `expr` is a `CodeString` object with all subexpression variables substituted with the respective expression.

units
Dictionary of all internal variables and their corresponding units.

check_flags(*allowed_flags*)
Check the list of flags.

Parameters allowed_flags : dict

A dictionary mapping equation types (PARAMETER, DIFFERENTIAL_EQUATION, SUBEXPRESSION) to a list of strings (the allowed flags for that equation type)

Raises

ValueError If any flags are used that are not allowed.

Notes

Not specifying allowed flags for an equation type is the same as specifying an empty list for it.

static **check_identifier**(*identifier*)
Perform all the registered checks. Checks can be registered via *Equations.register_identifier_check*.

Parameters identifier : str

The identifier that should be checked

Raises

ValueError If any of the registered checks fails.

check_identifiers()
Check all identifiers for conformity with the rules.

Raises

ValueError If an identifier does not conform to the rules.

See also:

Equations.check_identifier The function that is called for each identifier.

check_units(*group*, *run_namespace=None*, *level=0*)
Check all the units for consistency.

Parameters **group** : *Group*

The group providing the context

run_namespace : dict, optional

A namespace provided to the *Network.run()* function.

level : int, optional

How much further to go up in the stack to find the calling frame

Raises

DimensionMismatchError In case of any inconsistencies.

static register_identifier_check(*func*)
Register a function for checking identifiers.

Parameters **func** : callable

The function has to receive a single argument, the name of the identifier to check, and raise a ValueError if the identifier violates any rule.

Tutorials and examples using this

- Example IF_curve_Hodgkin_Huxley
- Example COBAHH
- Example frompapers/Diesmann_et_al_1999
- Example frompapers/Rossant_et_al_2011bis

SingleEquation(type, varname, unit[, ...])	Class for internal use, encapsulates a single equation or parameter.

SingleEquation class

(*Shortest import*: `from brian2.equations.equations import SingleEquation`)

class brian2.equations.equations.**SingleEquation**(*type*, *varname*, *unit*, *var_type='float'*, *expr=None*, *flags=None*)

Bases: object

Class for internal use, encapsulates a single equation or parameter.

Note: This class should never be used directly, it is only useful as part of the *Equations* class.

Parameters **type** : {PARAMETER, DIFFERENTIAL_EQUATION, SUBEXPRESSION}

The type of the equation.

varname : str

The variable that is defined by this equation.

unit : Unit

The unit of the variable

var_type : {FLOAT, BOOLEAN}

The type of the variable (floating point value or boolean).

expr : *Expression*, optional

The expression defining the variable (or `None` for parameters).

flags: list of str, optional :

A list of flags that give additional information about this equation. What flags are possible depends on the type of the equation and the context.

Attributes

identifiers	All identifiers in the RHS of this equation.
stochastic_variables	Stochastic variables in the RHS of this equation

Details

identifiers
All identifiers in the RHS of this equation.

stochastic_variables
Stochastic variables in the RHS of this equation

Functions

check_identifier_basic(identifier)	Check an identifier (usually resulting from an equation string provided by the user) for

check_identifier_basic function

(*Shortest import*: `from brian2.equations.equations import check_identifier_basic`)

`brian2.equations.equations.`**`check_identifier_basic`**(*identifier*)
Check an identifier (usually resulting from an equation string provided by the user) for conformity with the rules. The rules are:

1. Only ASCII characters
2. Starts with a character, then mix of alphanumerical characters and underscore

3.Is not a reserved keyword of Python

Parameters **identifier** : str

The identifier that should be checked

Raises

ValueError If the identifier does not conform to the above rules.

check_identifier_constants(identifier)	Make sure that identifier names do not clash with function names.

check_identifier_constants function

(*Shortest import*: `from brian2.equations.equations import check_identifier_constants`)

brian2.equations.equations.**check_identifier_constants**(*identifier*)

Make sure that identifier names do not clash with function names.

check_identifier_functions(identifier)	Make sure that identifier names do not clash with function names.

check_identifier_functions function

(*Shortest import*: `from brian2.equations.equations import check_identifier_functions`)

brian2.equations.equations.**check_identifier_functions**(*identifier*)

Make sure that identifier names do not clash with function names.

check_identifier_reserved(identifier)	Check that an identifier is not using a reserved special variable name.

check_identifier_reserved function

(*Shortest import*: `from brian2.equations.equations import check_identifier_reserved`)

brian2.equations.equations.**check_identifier_reserved**(*identifier*)

Check that an identifier is not using a reserved special variable name. The special variables are: ‘t’, ‘dt’, and ‘xi’, as well as everything starting with `xi_`.

Parameters **identifier: str** :

The identifier that should be checked

Raises

ValueError If the identifier is a special variable name.

check_identifier_units(identifier)	Make sure that identifier names do not clash with unit names.

check_identifier_units function

(*Shortest import*: `from brian2.equations.equations import check_identifier_units`)

`brian2.equations.equations.`**`check_identifier_units`**(*identifier*)
Make sure that identifier names do not clash with unit names.

parse_string_equations(eqns)	Parse a string defining equations.

parse_string_equations function

(*Shortest import*: `from brian2.equations.equations import parse_string_equations`)

`brian2.equations.equations.`**`parse_string_equations`**(*eqns*)
Parse a string defining equations.

Parameters **eqns** : str

The (possibly multi-line) string defining the equations. See the documentation of the *Equations* class for details.

Returns **equations** : dict

A dictionary mapping variable names to *Equations* objects

unit_and_type_from_string(unit_string)	Returns the unit that results from evaluating a string like “siemens / metre ** 2”,

unit_and_type_from_string function

(*Shortest import*: `from brian2.equations.equations import unit_and_type_from_string`)

`brian2.equations.equations.`**`unit_and_type_from_string`**(*unit_string*)
Returns the unit that results from evaluating a string like “siemens / metre ** 2”, allowing for the special string “1” to signify dimensionless units, the string “boolean” for a boolean and “integer” for an integer variable.

Parameters **unit_string** : str

The string that should evaluate to a unit

Returns **u, type** : (Unit, {FLOAT, INTEGER or BOOL})

The resulting unit and the type of the variable.

Raises

`ValueError` If the string cannot be evaluated to a unit.

`refractory` module

Module implementing Brian’s refractory mechanism.

Exported members: *add_refractoriness*

Functions

add_refractoriness(eqs)	Extends a given set of equations with the refractory mechanism.

add_refractoriness function

(*Shortest import*: `from brian2.equations.refractory import add_refractoriness`)

`brian2.equations.refractory.`**`add_refractoriness`**(*eqs*)

Extends a given set of equations with the refractory mechanism. New parameters are added and differential equations with the "unless refractory" flag are changed so that their right-hand side is 0 when the neuron is refractory (by multiplication with the `not_refractory` variable).

Parameters **eqs** : *Equations*

The equations without refractory mechanism.

Returns **new_eqs** : *Equations*

New equations, with added parameters and changed differential equations having the "unless refractory" flag.

check_identifier_refractory(identifier)	Check that the identifier is not using a name reserved for the refractory mechani

check_identifier_refractory function

(*Shortest import*: `from brian2.equations.refractory import check_identifier_refractory`)

`brian2.equations.refractory.`**`check_identifier_refractory`**(*identifier*)

Check that the identifier is not using a name reserved for the refractory mechanism. The reserved names are `not_refractory`, `refractory`, `refractory_until`.

Parameters **identifier** : str

The identifier to check.

Raises

`ValueError` If the identifier is a variable name used for the refractory mechanism.

`unitcheck` module

Utility functions for handling the units in *Equations*.

Exported members: `unit_from_expression`, *check_unit*, *check_units_statements*

Functions

check_unit(expression, unit, variables)	Compares the unit for an expression to an expected unit in a given namespace.

check_unit function

(*Shortest import*: `from brian2.equations.unitcheck import check_unit`)

`brian2.equations.unitcheck.`**`check_unit`**(*expression*, *unit*, *variables*)

Compares the unit for an expression to an expected unit in a given namespace.

Parameters **expression** : str

The expression to evaluate.

unit : *Unit*

The expected unit for the `expression`.

variables : dict

Dictionary of all variables (including external constants) used in the `expression`.

Raises

KeyError In case on of the identifiers cannot be resolved.

DimensionMismatchError If an unit mismatch occurs during the evaluation.

check_units_statements(code, variables)	Check the units for a series of statements.

check_units_statements function

(*Shortest import*: `from brian2.equations.unitcheck import check_units_statements`)

`brian2.equations.unitcheck.`**check_units_statements**(*code*, *variables*)

Check the units for a series of statements. Setting a model variable has to use the correct unit. For newly introduced temporary variables, the unit is determined and used to check the following statements to ensure consistency.

Parameters **code** : str

The statements as a (multi-line) string

variables : dict of `Variable` objects

The information about all variables used in `code` (including `Constant` objects for external variables)

Raises

KeyError In case on of the identifiers cannot be resolved.

DimensionMismatchError If an unit mismatch occurs during the evaluation.

6.4.5 groups package

Package providing groups such as *NeuronGroup* or *PoissonGroup*.

group module

This module defines the *Group* object, a mix-in class for everything that saves state variables, e.g. *NeuronGroup* or *StateMonitor*.

Exported members: *Group*, *CodeRunner*

Classes

CodeRunner(group, template[, code, ...])	A "code runner" that runs a *CodeObject* every timestep and keeps a reference to the

CodeRunner class

(*Shortest import*: `from brian2 import CodeRunner`)

class `brian2.groups.group.`**CodeRunner**(*group*, *template*, *code=''*, *user_code=None*, *dt=None*, *clock=None*, *when='start'*, *order=0*, *name='coderunner*'*, *check_units=True*, *template_kwds=None*, *needed_variables=None*, *override_conditional_write=None*, *codeobj_class=None*)

Bases: *brian2.core.base.BrianObject*

A "code runner" that runs a *CodeObject* every timestep and keeps a reference to the *Group*. Used in *NeuronGroup* for `Thresholder`, `Resetter` and `StateUpdater`.

On creation, we try to run the before_run method with an empty additional namespace (see *Network.before_run()*). If the namespace is already complete this might catch unit mismatches.

Parameters **group** : *Group*

The group to which this object belongs.

template : `Template`

The template that should be used for code generation

code : str, optional

The abstract code that should be executed every time step. The *update_abstract_code* method might generate this code dynamically before every run instead.

dt : *Quantity*, optional

The time step to be used for the simulation. Cannot be combined with the `clock` argument.

user_code : str, optional

The abstract code as specified by the user, i.e. without any additions of internal code that the user not necessarily knows about. This will be used for warnings and error messages.

clock : *Clock*, optional

The update clock to be used. If neither a clock, nor the `dt` argument is specified, the *defaultclock* will be used.

when : str, optional

In which scheduling slot to execute the operation during a time step. Defaults to `'start'`.

order : int, optional

The priority of this operation for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

name : str, optional

The name for this object.

check_units : bool, optional

Whether the units should be checked for consistency before a run. Is activated (`True`) by default but should be switched off for state updaters (units are already checked for the equations and the generated abstract code might have already replaced variables with their unit-less values)

template_kwds : dict, optional

A dictionary of additional information that is passed to the template.

needed_variables: list of str, optional :

A list of variables that are neither present in the abstract code, nor in the `USES_VARIABLES` statement in the template. This is only rarely necessary, an example being a *`StateMonitor`* where the names of the variables are neither known to the template nor included in the abstract code statements.

override_conditional_write: list of str, optional :

A list of variable names which are used as conditions (e.g. for refractoriness) which should be ignored.

codeobj_class : class, optional

The *`CodeObject`* class to run code with. If not specified, defaults to the `group`'s `codeobj_class` attribute.

Methods

`before_run`([run_namespace, level])	
`update_abstract_code`([run_namespace, level])	Update the abstract code for the code object.

Details

`before_run`(*run_namespace=None*, *level=0*)

`update_abstract_code`(*run_namespace=None*, *level=0*)

Update the abstract code for the code object. Will be called in *`before_run`* and should update the `CodeRunner.abstract_code` attribute.

Does nothing by default.

`Group`(*args, **kwds)	Mix-in class for accessing arrays by attribute.

Group class

(*Shortest import*: `from brian2 import Group`)

class `brian2.groups.group.`**`Group`**(**args*, ***kwds*)

Bases: *`brian2.core.base.BrianObject`*

Mix-in class for accessing arrays by attribute.

TODO: Overwrite the __dir__ method to return the state variables # (should make autocompletion work)

Methods

add_attribute(name)	Add a new attribute to this group.
custom_operation(*args, **kwds)	
get_states([vars, units, format, ...])	Return a copy of the current state variable values.
resolve(identifier[, user_identifier, ...])	Resolve an identifier (i.e.
resolve_all(identifiers[, user_identifiers, ...])	Resolve a list of identifiers.
run_regularly(code[, dt, clock, when, ...])	Run abstract code in the group's namespace.
runner(*args, **kwds)	
set_states(values[, units, format, level])	Set the state variables.
state(name[, use_units, level])	Return the state variable in a way that properly supports indexing in

Details

add_attribute(*name*)

Add a new attribute to this group. Using this method instead of simply assigning to the new attribute name is necessary because Brian will raise an error in that case, to avoid bugs passing unnoticed (misspelled state variable name, un-declared state variable, ...).

Parameters **name** : str

The name of the new attribute

Raises

AttributeError If the name already exists as an attribute or a state variable.

custom_operation(**args*, ***kwds*)

get_states(*vars=None*, *units=True*, *format='dict'*, *subexpressions=False*, *level=0*)

Return a copy of the current state variable values. The returned arrays are copies of the actual arrays that store the state variable values, therefore changing the values in the returned dictionary will not affect the state variables.

Parameters **vars** : list of str, optional

The names of the variables to extract. If not specified, extract all state variables (except for internal variables, i.e. names that start with `'_'`). If the `subexpressions` argument is `True`, the current values of all subexpressions are returned as well.

units : bool, optional

Whether to include the physical units in the return value. Defaults to `True`.

format : str, optional

The output format. Defaults to `'dict'`.

subexpressions: bool, optional :

Whether to return subexpressions when no list of variable names is given. Defaults to `False`. This argument is ignored if an explicit list of variable names is given in `vars`.

level : int, optional

How much higher to go up the stack to resolve external variables. Only relevant if extracting subexpressions that refer to external variables.

Returns values :

The variables specified in `vars`, in the specified `format`.

resolve(*identifier*, *user_identifier=True*, *additional_variables=None*, *run_namespace=None*, *level=0*)

Resolve an identifier (i.e. variable, constant or function name) in the context of this group. This function will first lookup the name in the state variables, then look for a standard function or unit of that name and finally look in `Group.namespace` and in `run_namespace`. If the latter is not given, it will try to find the variable in the local namespace where the original function call took place. See *External variables and functions*.

Parameters identifiers : str

The name to look up.

user_identifier : bool, optional

Whether this is an identifier that was used by the user (and not something automatically generated that the user might not even know about). Will be used to determine whether to display a warning in the case of namespace clashes. Defaults to `True`.

additional_variables : dict-like, optional

An additional mapping of names to `Variable` objects that will be checked before `Group.variables`.

run_namespace : dict-like, optional

An additional namespace, provided as an argument to the *`Network.run()`* method.

level : int, optional

How far to go up in the stack to find the original call frame.

Returns obj : `Variable` or *`Function`*

Returns a `Variable` object describing the variable or a *`Function`* object for a function. External variables are represented as `Constant` objects

Raises

KeyError If the `identifier` could not be resolved

resolve_all(*identifiers*, *user_identifiers=None*, *additional_variables=None*, *run_namespace=None*, *level=0*)

Resolve a list of identifiers. Calls *`Group.resolve()`* for each identifier.

Parameters identifiers : iterable of str

The names to look up.

user_identifiers : iterable of str, optional

The names in `identifiers` that were provided by the user (i.e. are part of user-specified equations, abstract code, etc.). Will be used to determine when to issue namespace conflict warnings. If not specified, will be assumed to be identical to `identifiers`.

additional_variables : dict-like, optional

An additional mapping of names to `Variable` objects that will be checked before `Group.variables`.

run_namespace : dict-like, optional

An additional namespace, provided as an argument to the *Network.run()* method.

level : int, optional

How far to go up in the stack to find the original call frame.

do_warn : bool, optional

Whether to warn about names that are defined both as an internal variable (i.e. in `Group.variables`) and in some other namespace. Defaults to `True` but can be switched off for internal variables used in templates that the user might not even know about.

Returns **variables** : dict of `Variable` or *Function*

A mapping from name to `Variable`/*Function* object for each of the names given in `identifiers`

Raises

KeyError If one of the names in `identifier` cannot be resolved

run_regularly(*code*, *dt=None*, *clock=None*, *when='start'*, *order=0*, *name=None*, *codeobj_class=None*)

Run abstract code in the group's namespace. The created *CodeRunner* object will be automatically added to the group, it therefore does not need to be added to the network manually. However, a reference to the object will be returned, which can be used to later remove it from the group or to set it to inactive.

Parameters **code** : str

The abstract code to run.

dt : *Quantity*, optional

The time step to use for this custom operation. Cannot be combined with the `clock` argument.

clock : *Clock*, optional

The update clock to use for this operation. If neither a clock nor the `dt` argument is specified, defaults to the clock of the group.

when : str, optional

When to run within a time step, defaults to the `'start'` slot.

name : str, optional

A unique name, if non is given the name of the group appended with 'run_regularly', 'run_regularly_1', etc. will be used. If a name is given explicitly, it will be used as given (i.e. the group name will not be prepended automatically).

codeobj_class : class, optional

The *CodeObject* class to run code with. If not specified, defaults to the `group`'s `codeobj_class` attribute.

Returns **obj** : *CodeRunner*

A reference to the object that will be run.

runner(**args*, ***kwds*)

set_states(*values*, *units=True*, *format='dict'*, *level=0*)
Set the state variables.

Parameters **values** : depends on `format`

The values according to `format`.

units : bool, optional

Whether the `values` include physical units. Defaults to `True`.

format : str, optional

The format of `values`. Defaults to `'dict'`

level : int, optional

How much higher to go up the stack to resolve external variables. Only relevant when using string expressions to set values.

state(*name*, *use_units=True*, *level=0*)
Return the state variable in a way that properly supports indexing in the context of this group

Parameters **name** : str

The name of the state variable

use_units : bool, optional

Whether to use the state variable's unit.

level : int, optional

How much farther to go down in the stack to find the namespace.

Returns :

——- :

var : `VariableView` or scalar value

The state variable's value that can be indexed (for non-scalar values).

IndexWrapper(group)	Convenience class to allow access to the indices via indexing syntax.

IndexWrapper class

(*Shortest import*: `from brian2.groups.group import IndexWrapper`)

class `brian2.groups.group.`**IndexWrapper**(*group*)
Bases: `object`

Convenience class to allow access to the indices via indexing syntax. This allows for example to get all indices for synapses originating from neuron 10 by writing `synapses.indices[10, :]` instead of `synapses._indices.((10, slice(None))`.

Indexing(group[, default_index])	Object responsible for calculating flat index arrays from arbitrary group- specific indices.

Indexing class

(*Shortest import*: `from brian2.groups.group import Indexing`)

class `brian2.groups.group.`**Indexing**(*group*, *default_index='_idx'*)

Bases: object

Object responsible for calculating flat index arrays from arbitrary group- specific indices. Stores strong references to the necessary variables so that basic indexing (i.e. slicing, integer arrays/values, ...) works even when the respective *Group* no longer exists. Note that this object does not handle string indexing.

Methods

__call__([item, index_var])	Return flat indices to index into state variables from arbitrary group specific indices.

Details

__call__(*item=None*, *index_var=None*)

Return flat indices to index into state variables from arbitrary group specific indices. In the default implementation, raises an error for multidimensional indices and transforms slices into arrays.

Parameters **item** : slice, array, int

The indices to translate.

Returns **indices** : numpy.ndarray

The flat indices corresponding to the indices given in item.

See also:

SynapticIndexing

Functions

get_dtype(equation[, dtype])	Helper function to interpret the dtype keyword argument in *NeuronGroup* etc.

get_dtype function

(*Shortest import*: from brian2.groups.group import get_dtype)

brian2.groups.group.**get_dtype**(*equation*, *dtype=None*)

Helper function to interpret the dtype keyword argument in *NeuronGroup* etc.

Parameters **equation** : SingleEquation

The equation for which a dtype should be returned

dtype : dtype or dict, optional

Either the dtype to be used as a default dtype for all float variables (instead of the *core.default_float_dtype* preference) or a dictionary stating the dtype for some variables; all other variables will use the preference default

Returns **d** : dtype

The dtype for the variable defined in equation

neurongroup module

This model defines the *NeuronGroup*, the core of most simulations.

Exported members: *NeuronGroup*

Classes

NeuronGroup(N, model[, method, threshold, ...])	A group of neurons.

NeuronGroup class

(*Shortest import*: `from brian2 import NeuronGroup`)

class `brian2.groups.neurongroup.`**`NeuronGroup`**(*N*, *model*, *method=('linear', 'euler', 'milstein')*, *threshold=None*, *reset=None*, *refractory=False*, *events=None*, *namespace=None*, *dtype=None*, *dt=None*, *clock=None*, *order=0*, *name='neurongroup*'*, *codeobj_class=None*)

Bases: *brian2.groups.group.Group*, *brian2.core.spikesource.SpikeSource*

A group of neurons.

Parameters **N** : int

Number of neurons in the group.

model : (str, *Equations*)

The differential equations defining the group

method : (str, function), optional

The numerical integration method. Either a string with the name of a registered method (e.g. "euler") or a function that receives an *Equations* object and returns the corresponding abstract code. If no method is specified, a suitable method will be chosen automatically.

threshold : str, optional

The condition which produces spikes. Should be a single line boolean expression.

reset : str, optional

The (possibly multi-line) string with the code to execute on reset.

refractory : {str, *Quantity*}, optional

Either the length of the refractory period (e.g. `2*ms`), a string expression that evaluates to the length of the refractory period after each spike (e.g. `'(1 + rand())*ms'`), or a string expression evaluating to a boolean value, given the condition under which the neuron stays refractory after a spike (e.g. `'v > -20*mV'`)

events : dict, optional

User-defined events in addition to the "spike" event defined by the `threshold`. Has to be a mapping of strings (the event name) to

strings (the condition) that will be checked.

namespace: dict, optional :

A dictionary mapping variable/function names to the respective objects. If no *namespace* is given, the "implicit" namespace, consisting of the local and global namespace surrounding the creation of the class, is used.

dtype : (`dtype`, `dict`), optional

The `numpy.dtype` that will be used to store the values, or a dictionary specifying the type for variable names. If a value is not provided for a variable (or no value is provided at all), the preference setting *core.default_float_dtype* is used.

codeobj_class : class, optional

The *CodeObject* class to run code with.

dt : *Quantity*, optional

The time step to be used for the simulation. Cannot be combined with the `clock` argument.

clock : *Clock*, optional

The update clock to be used. If neither a clock, nor the `dt` argument is specified, the *defaultclock* will be used.

order : int, optional

The priority of of this group for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

name : str, optional

A unique name for the group, otherwise use `neurongroup_0`, etc.

Notes

NeuronGroup contains a *StateUpdater*, *Thresholder* and *Resetter*, and these are run at the 'groups', 'thresholds' and 'resets' slots (i.e. the values of their `when` attribute take these values). The `order` attribute will be passed down to the contained objects but can be set individually by setting the `order` attribute of the *state_updater*, *thresholder* and *resetter* attributes, respectively.

Attributes

_refractory	The refractory condition or timespan
event_codes	Code that is triggered on events (e.g.
events	Events supported by this group
method_choice	The state update method selected by the user
namespace	The group-specific namespace
resetter	Reset neurons which have spiked (or perform arbitrary actions for
spikes	The spikes returned by the most recent thresholding operation.
state_updater	Performs numerical integration step
thresholder	Checks the spike threshold (or abitrary user-defined events)

Methods

before_run([run_namespace, level])	
run_on_event(event, code[, when, order])	Run code triggered by a custom-defined event (see *NeuronGroup* documentatio
set_event_schedule(event[, when, order])	Change the scheduling slot for checking the condition of an event.
state(name[, use_units, level])	

Details

_refractory
The refractory condition or timespan

event_codes
Code that is triggered on events (e.g. reset)

events
Events supported by this group

method_choice
The state update method selected by the user

namespace
The group-specific namespace

resetter
Reset neurons which have spiked (or perform arbitrary actions for user-defined events)

spikes
The spikes returned by the most recent thresholding operation.

state_updater
Performs numerical integration step

thresholder
Checks the spike threshold (or abitrary user-defined events)

before_run(*run_namespace=None*, *level=0*)

run_on_event(*event*, *code*, *when='after_resets'*, *order=None*)
Run code triggered by a custom-defined event (see *NeuronGroup* documentation for the specification of events).The created *Resetter* object will be automatically added to the group, it therefore does not need to be added to the network manually. However, a reference to the object will be returned, which can be used to later remove it from the group or to set it to inactive.

Parameters **event** : str

The name of the event that should trigger the code

code : str

The code that should be executed

when : str, optional

The scheduling slot that should be used to execute the code. Defaults to `'after_resets'`.

order : int, optional

The order for operations in the same scheduling slot. Defaults to the order of the *NeuronGroup*.

Returns **obj** : *Resetter*

A reference to the object that will be run.

set_event_schedule(*event*, *when='after_thresholds'*, *order=None*)
Change the scheduling slot for checking the condition of an event.

Parameters **event** : str

The name of the event for which the scheduling should be changed

when : str, optional

The scheduling slot that should be used to check the condition. Defaults to `'after_thresholds'`.

order : int, optional

The order for operations in the same scheduling slot. Defaults to the order of the *NeuronGroup*.

state(*name*, *use_units=True*, *level=0*)

Tutorials and examples using this

- Tutorial 1-intro-to-brian-neurons
- Tutorial 2-intro-to-brian-synapses
- Example reliability
- Example adaptive_threshold
- Example non_reliability
- Example IF_curve_LIF
- Example IF_curve_Hodgkin_Huxley
- Example phase_locking
- Example CUBA
- Example COBAHH
- Example compartmental/lfp
- Example compartmental/bipolar_with_inputs2
- Example compartmental/bipolar_with_inputs
- Example advanced/stochastic_odes
- Example advanced/opencv_movie
- Example standalone/STDP_standalone
- Example standalone/cuba_openmp
- Example synapses/nonlinear
- Example synapses/synapses
- Example synapses/STDP
- Example synapses/spatial_connections
- Example synapses/jeffress
- Example synapses/state_variables
- Example synapses/gapjunctions
- Example synapses/licklider
- Example frompapers/Brette_Gerstner_2005
- Example frompapers/Brette_Guigon_2003
- Example frompapers/Diesmann_et_al_1999

- Example frompapers/Rossant_et_al_2011bis
- Example frompapers/Rothman_Manis_2003
- Example frompapers/Brette_2004
- Example frompapers/kremer_et_al_2011_barrel_cortex
- Example frompapers/Brunel_Hakim_1999
- Example frompapers/Touboul_Brette_2008
- Example frompapers/Vogels_et_al_2011
- Example frompapers/Wang_Buszaki_1996
- Example frompapers/Sturzl_et_al_2000

Resetter(group[, when, order, event])	The *CodeRunner* that applies the reset statement(s) to the state variables of neurons that

Resetter class

(*Shortest import*: `from brian2.groups.neurongroup import Resetter`)

class `brian2.groups.neurongroup.`**`Resetter`**(*group*, *when='resets'*, *order=None*, *event='spike'*)
Bases: *brian2.groups.group.CodeRunner*

The *CodeRunner* that applies the reset statement(s) to the state variables of neurons that have spiked in this timestep.

Methods

update_abstract_code([run_namespace, level])	

Details

`update_abstract_code`(*run_namespace=None*, *level=0*)

StateUpdater(group, method)	The *CodeRunner* that updates the state variables of a *NeuronGroup* at every timestep.

StateUpdater class

(*Shortest import*: `from brian2.groups.neurongroup import StateUpdater`)

class `brian2.groups.neurongroup.`**`StateUpdater`**(*group*, *method*)
Bases: *brian2.groups.group.CodeRunner*

The *CodeRunner* that updates the state variables of a *NeuronGroup* at every timestep.

Methods

update_abstract_code([run_namespace, level])	

Details

update_abstract_code(*run_namespace=None*, *level=0*)

Thresholder(group[, when, event])	The *CodeRunner* that applies the threshold condition to the state variables of a *NeuronG*

Thresholder class

(*Shortest import*: `from brian2.groups.neurongroup import Thresholder`)

class `brian2.groups.neurongroup.`**Thresholder**(*group*, *when='thresholds'*, *event='spike'*)
Bases: *brian2.groups.group.CodeRunner*

The *CodeRunner* that applies the threshold condition to the state variables of a *NeuronGroup* at every timestep and sets its `spikes` and `refractory_until` attributes.

Methods

update_abstract_code([run_namespace, level])

Details

update_abstract_code(*run_namespace=None*, *level=0*)

subgroup module

Exported members: *Subgroup*

Classes

Subgroup(source, start, stop[, name])	Subgroup of any *Group*

Subgroup class

(*Shortest import*: `from brian2 import Subgroup`)

class `brian2.groups.subgroup.`**Subgroup**(*source*, *start*, *stop*, *name=None*)
Bases: *brian2.groups.group.Group*, *brian2.core.spikesource.SpikeSource*

Subgroup of any *Group*

Parameters **source** : SpikeSource

The source object to subgroup.

start, stop : int

Select only spikes with indices from `start` to `stop-1`.

name : str, optional

A unique name for the group, or use `source.name+'_subgroup_0'`, etc.

Attributes

spikes

Details

spikes

6.4.6 input package

Classes for providing external input to a network.

binomial module

Implementation of *BinomialFunction*

Exported members: *BinomialFunction*

Classes

BinomialFunction(n, p[, approximate, name])	A function that generates samples from a binomial distribution.

BinomialFunction class

(*Shortest import*: `from brian2 import BinomialFunction`)

class `brian2.input.binomial.`**`BinomialFunction`**(*n*, *p*, *approximate=True*, *name='_binomial*'*)

Bases: *brian2.core.functions.Function*, *brian2.core.names.Nameable*

A function that generates samples from a binomial distribution.

Parameters **n** : int

Number of samples

p : float

Probablility

approximate : bool, optional

Whether to approximate the binomial with a normal distribution if $np > 5 \wedge n(1-p) > 5$. Defaults to `True`.

poissongroup module

Implementation of *PoissonGroup*.

Exported members: *PoissonGroup*

Classes

PoissonGroup(*args, **kwds)	Poisson spike source

PoissonGroup class

(*Shortest import*: `from brian2 import PoissonGroup`)

class `brian2.input.poissongroup.`**`PoissonGroup`**(**args*, ***kwds*)

Bases: *brian2.groups.group.Group*, *brian2.core.spikesource.SpikeSource*

Poisson spike source

Parameters **N** : int

Number of neurons

rates : *Quantity*, str

Single rate, array of rates of length N, or a string expression evaluating to a rate

dt : *Quantity*, optional

The time step to be used for the simulation. Cannot be combined with the `clock` argument.

clock : *Clock*, optional

The update clock to be used. If neither a clock, nor the `dt` argument is specified, the *defaultclock* will be used.

when : str, optional

When to run within a time step, defaults to the `'thresholds'` slot.

order : int, optional

The priority of of this group for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

name : str, optional

Unique name, or use poissongroup, poissongroup_1, etc.

Attributes

spikes	The spikes returned by the most recent thresholding operation.

Details

`spikes`

The spikes returned by the most recent thresholding operation.

Tutorials and examples using this

- Example adaptive_threshold
- Example standalone/STDP_standalone
- Example synapses/STDP

poissoninput module

Implementation of *PoissonInput*.

Exported members: *PoissonInput*

Classes

PoissonInput(target, target_var, N, rate, weight)	Adds independent Poisson input to a target variable of a *Group*.

PoissonInput class

(*Shortest import*: `from brian2 import PoissonInput`)

class `brian2.input.poissoninput.`**`PoissonInput`**(*target*, *target_var*, *N*, *rate*, *weight*, *when='synapses'*, *order=0*)

Bases: *brian2.groups.group.CodeRunner*

Adds independent Poisson input to a target variable of a *Group*. For large numbers of inputs, this is much more efficient than creating a *PoissonGroup*. The synaptic events are generated randomly during the simulation and are not preloaded and stored in memory. All the inputs must target the same variable, have the same frequency and same synaptic weight. All neurons in the target *Group* receive independent realizations of Poisson spike trains.

Parameters **target** : *Group*

The group that is targeted by this input.

target_var : str

The variable of `target` that is targeted by this input.

N : int

The number of inputs

rate : *Quantity*

The rate of each of the inputs

weight : str or *Quantity*

Either a string expression (that can be interpreted in the context of `target`) or a *Quantity* that will be added for every event to the `target_var` of `target`. The unit has to match the unit of `target_var`

when : str, optional

When to update the target variable during a time step. Defaults to the `synapses` scheduling slot.

order : int, optional

The priority of of the update compared to other operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

Methods

before_run([run_namespace, level])	

Details

before_run(*run_namespace=None*, *level=0*)

Tutorials and examples using this

- Example frompapers/Rossant_et_al_2011bis

spikegeneratorgroup module

Module defining *SpikeGeneratorGroup*.

Exported members: *SpikeGeneratorGroup*

Classes

SpikeGeneratorGroup(N, indices, times[, dt, ...])	A group emitting spikes at given times.

SpikeGeneratorGroup class

(*Shortest import*: `from brian2 import SpikeGeneratorGroup`)

class brian2.input.spikegeneratorgroup.**SpikeGeneratorGroup**(*N*, *indices*, *times*, *dt=None*, *clock=None*, *period=1e100*second*, *when='thresholds'*, *order=0*, *sorted=False*, *name='spikegeneratorgroup*'*, *codeobj_class=None*)

Bases: *brian2.groups.group.Group*, *brian2.groups.group.CodeRunner*, *brian2.core.spikesource.SpikeSource*

A group emitting spikes at given times.

Parameters **N** : int

The number of "neurons" in this group

indices : array of integers

The indices of the spiking cells

times : *Quantity*

The spike times for the cells given in `indices`. Has to have the same length as `indices`.

period : *Quantity*, optional

If this is specified, it will repeat spikes with this period.

dt : *Quantity*, optional

The time step to be used for the simulation. Cannot be combined with the `clock` argument.

clock : *Clock*, optional

The update clock to be used. If neither a clock, nor the `dt` argument is specified, the *defaultclock* will be used.

when : str, optional

When to run within a time step, defaults to the `'thresholds'` slot.

order : int, optional

The priority of of this group for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

sorted : bool, optional

Whether the given indices and times are already sorted. Set to `True` if your events are already sorted (first by spike time, then by index), this can save significant time at construction if your arrays contain large numbers of spikes. Defaults to `False`.

Notes

- In a time step, *SpikeGeneratorGroup* emits all spikes that happened at $t - dt < t_{spike} \leq t$. This might lead to unexpected or missing spikes if you change the time step dt between runs.
- *SpikeGeneratorGroup* does not currently raise any warning if a neuron spikes more that once during a time step, but other code (e.g. for synaptic propagation) might assume that neurons only spike once per time step and will therefore not work properly.
- If `sorted` is set to `True`, the given arrays will not be copied (only affects runtime mode)..

Attributes

_previous_dt	Remember the dt we used the last time when we checked the spike bins
_spikes_changed	"Dirty flag" that will be set when spikes are changed after the
spikes	The spikes returned by the most recent thresholding operation.

Methods

before_run([run_namespace, level])

Details

_previous_dt
Remember the dt we used the last time when we checked the spike bins to not repeat the work for multiple runs with the same dt

_spikes_changed
"Dirty flag" that will be set when spikes are changed after the *before_run* check

spikes
The spikes returned by the most recent thresholding operation.

before_run(*run_namespace=None*, *level=0*)

set_spikes(*indices*, *times*, *period=1e100*second*, *sorted=False*)
Change the spikes that this group will generate.

This can be used to set the input for a second run of a model based on the output of a first run (if the input for the second run is already known before the first run, then all the information should simply be included

in the initial *SpikeGeneratorGroup* initializer call, instead).

Parameters **indices** : array of integers

The indices of the spiking cells

times : *Quantity*

The spike times for the cells given in `indices`. Has to have the same length as `indices`.

period : *Quantity*, optional

If this is specified, it will repeat spikes with this period.

sorted : bool, optional

Whether the given indices and times are already sorted. Set to `True` if your events are already sorted (first by spike time, then by index), this can save significant time at construction if your arrays contain large numbers of spikes. Defaults to `False`.

Tutorials and examples using this

- Example frompapers/Diesmann_et_al_1999

`timedarray` module

Implementation of *TimedArray*.

Exported members: *TimedArray*

Classes

TimedArray(values, dt[, name])	A function of time built from an array of values.

TimedArray class

(*Shortest import*: `from brian2 import TimedArray`)

class `brian2.input.timedarray.`**TimedArray**(*values*, *dt*, *name=None*)

Bases: *brian2.core.functions.Function*, *brian2.core.names.Nameable*

A function of time built from an array of values. The returned object can be used as a function, including in model equations etc. The resulting function has to be called as `funcion_name(t)` if the provided value array is one-dimensional and as `function_name(t, i)` if it is two-dimensional.

Parameters **values** : ndarray or *Quantity*

An array of values providing the values at various points in time. This array can either be one- or two-dimensional. If it is two-dimensional it's first dimension should be the time.

dt : *Quantity*

The time distance between values in the `values` array.

name : str, optional

A unique name for this object, see *Nameable* for details. Defaults to `'_timedarray*'`.

Notes

For time values corresponding to elements outside of the range of `values` provided, the first respectively last element is returned.

Examples

```
>>> from brian2 import *
>>> ta = TimedArray([1, 2, 3, 4] * mV, dt=0.1*ms)
>>> print(ta(0.3*ms))
4. mV
>>> G = NeuronGroup(1, 'v = ta(t) : volt')
>>> mon = StateMonitor(G, 'v', record=True)
>>> net = Network(G, mon)
>>> net.run(1*ms)
...
>>> print(mon[0].v)
[ 1.  2.  3.  4.  4.  4.  4.  4.  4.  4.] mV
>>> ta2d = TimedArray([[1, 2], [3, 4], [5, 6]]*mV, dt=0.1*ms)
>>> G = NeuronGroup(4, 'v = ta2d(t, i%2) : volt')
>>> mon = StateMonitor(G, 'v', record=True)
>>> net = Network(G, mon)
>>> net.run(0.2*ms)
...
>>> print mon.v[:]
[[ 1.  3.]
 [ 2.  4.]
 [ 1.  3.]
 [ 2.  4.]] mV
```

Methods

is_locally_constant(dt)

Details

is_locally_constant(*dt*)

Tutorials and examples using this

- Example synapses/jeffress
- Example frompapers/Sturzl_et_al_2000

6.4.7 memory package

dynamicarray module

TODO: rewrite this (verbatim from Brian 1.x), more efficiency

Exported members: *DynamicArray*, *DynamicArray1D*

Classes

DynamicArray(shape[, dtype, factor, ...])	An N-dimensional dynamic array class

DynamicArray class

(*Shortest import*: `from brian2.memory.dynamicarray import DynamicArray`)

class `brian2.memory.dynamicarray.`**`DynamicArray`**(*shape*, *dtype=<type 'float'>*, *factor=2*, *use_numpy_resize=False*, *refcheck=True*)

Bases: `object`

An N-dimensional dynamic array class

The array can be resized in any dimension, and the class will handle allocating a new block of data and copying when necessary.

> **Warning:** The data will NOT be contiguous for >1D arrays. To ensure this, you will either need to use 1D arrays, or to copy the data, or use the shrink method with the current size (although note that in both cases you negate the memory and efficiency benefits of the dynamic array).

Initialisation arguments:

`shape`, `dtype` The shape and dtype of the array to initialise, as in Numpy. For 1D arrays, shape can be a single int, for ND arrays it should be a tuple.

`factor` The resizing factor (see notes below). Larger values tend to lead to more wasted memory, but more computationally efficient code.

`use_numpy_resize`, `refcheck` Normally, when you resize the array it creates a new array and copies the data. Sometimes, it is possible to resize an array without a copy, and if this option is set it will attempt to do this. However, this can cause memory problems if you are not careful so the option is off by default. You need to ensure that you do not create slices of the array so that no references to the memory exist other than the main array object. If you are sure you know what you're doing, you can switch this reference check off. Note that resizing in this way is only done if you resize in the first dimension.

The array is initialised with zeros. The data is stored in the attribute `data` which is a Numpy array.

Some numpy methods are implemented and can work directly on the array object, including `len(arr)`, `arr[...]` and `arr[...]=...`. In other cases, use the `data` attribute.

Notes

The dynamic array returns a `data` attribute which is a view on the larger `_data` attribute. When a resize operation is performed, and a specific dimension is enlarged beyond the size in the `_data` attribute, the size is increased to the larger of `cursize*factor` and `newsize`. This ensures that the amortized cost of increasing the size of the array is O(1).

Examples

```
>>> x = DynamicArray((2, 3), dtype=int)
>>> x[:] = 1
>>> x.resize((3, 3))
>>> x[:] += 1
>>> x.resize((3, 4))
```

```
>>> x[:] += 1
>>> x.resize((4, 4))
>>> x[:] += 1
>>> x.data[:] = x.data**2
>>> x.data
array([[16, 16, 16,  4],
       [16, 16, 16,  4],
       [ 9,  9,  9,  4],
       [ 1,  1,  1,  1]])
```

Methods

resize(newshape)	Resizes the data to the new shape, which can be a different size to the current data, but shoul
resize_along_first(newshape)	
shrink(newshape)	Reduces the data to the given shape, which should be smaller than the current shape.

Details

resize(*newshape*)
Resizes the data to the new shape, which can be a different size to the current data, but should have the same rank, i.e. same number of dimensions.

resize_along_first(*newshape*)

shrink(*newshape*)
Reduces the data to the given shape, which should be smaller than the current shape. `resize()` can also be used with smaller values, but it will not shrink the allocated memory, whereas *shrink* will reallocate the memory. This method should only be used infrequently, as if it is used frequently it will negate the computational efficiency benefits of the DynamicArray.

DynamicArray1D(shape[, dtype, factor, ...])	Version of *DynamicArray* with specialised `resize` method designed to be more

DynamicArray1D class

(*Shortest import*: `from brian2.memory.dynamicarray import DynamicArray1D`)

class `brian2.memory.dynamicarray.`**DynamicArray1D**(*shape*, *dtype=<type 'float'>*, *factor=2*, *use_numpy_resize=False*, *refcheck=True*)
Bases: *brian2.memory.dynamicarray.DynamicArray*

Version of *DynamicArray* with specialised `resize` method designed to be more efficient.

Methods

resize(newshape)

Details

resize(*newshape*)

Functions

getslices(shape)	

getslices function

(*Shortest import*: `from brian2.memory.dynamicarray import getslices`)

`brian2.memory.dynamicarray.`**`getslices`**`(`*shape*`)`

6.4.8 monitors package

`ratemonitor` module

Exported members: *PopulationRateMonitor*

Classes

PopulationRateMonitor(source[, name, ...])	Record instantaneous firing rates, averaged across neurons from a *NeuronGro*

PopulationRateMonitor class

(*Shortest import*: `from brian2 import PopulationRateMonitor`)

class `brian2.monitors.ratemonitor.`**`PopulationRateMonitor`**`(`*source*, *name='ratemonitor*'*, *codeobj_class=None*`)`

Bases: *brian2.groups.group.Group*, *brian2.groups.group.CodeRunner*

Record instantaneous firing rates, averaged across neurons from a *NeuronGroup* or other spike source.

Parameters **source** : (*NeuronGroup*, *SpikeSource*)

The source of spikes to record.

name : str, optional

A unique name for the object, otherwise will use `source.name+'_ratemonitor_0'`, etc.

codeobj_class : class, optional

The *CodeObject* class to run code with.

Notes

Currently, this monitor can only monitor the instantaneous firing rates at each time step of the source clock. Any binning/smoothing of the firing rates has to be done manually afterwards.

Attributes

source	The group we are recording from

Methods

reinit()	Clears all recorded rates
resize(new_size)	

Details

source
The group we are recording from

reinit()
Clears all recorded rates

resize(*new_size*)

Tutorials and examples using this

- Example standalone/STDP_standalone
- Example synapses/STDP
- Example frompapers/Brunel_Hakim_1999

spikemonitor module

Exported members: *EventMonitor*, *SpikeMonitor*

Classes

EventMonitor(source, event[, variables, ...])	Record events from a *NeuronGroup* or another event source.

EventMonitor class

(*Shortest import*: `from brian2 import EventMonitor`)

class `brian2.monitors.spikemonitor.`**EventMonitor**(*source*, *event*, *variables=None*, *when=None*, *order=None*, *name='eventmonitor*'*, *codeobj_class=None*)

Bases: *brian2.groups.group.Group*, *brian2.groups.group.CodeRunner*

Record events from a *NeuronGroup* or another event source.

The recorded events can be accessed in various ways: the attributes `i` and `t` store all the indices and event times, respectively. Alternatively, you can get a dictionary mapping neuron indices to event trains, by calling the *event_trains* method.

Parameters **source** : *NeuronGroup*

The source of events to record.

record : bool

Whether or not to record each event in `i` and `t` (the `count` will always be recorded).

event : str

The name of the event to record

variables : str or sequence of str, optional

Which variables to record at the time of the event (in addition to the index of the neuron). Can be the name of a variable or a list of names.

when : str, optional

When to record the events, by default records events in the same slot where the event is emitted.

order : int, optional

The priority of of this group for operations occurring at the same time step and in the same scheduling slot. Defaults to the order where the event is emitted + 1, i.e. it will be recorded directly afterwards.

name : str, optional

A unique name for the object, otherwise will use `source.name+'_eventmonitor_0'`, etc.

codeobj_class : class, optional

The *CodeObject* class to run code with.

See also:

SpikeMonitor

Attributes

event	The event that we are listening to
it	Returns the pair (`i`, `t`).
*it_*	Returns the pair (`i`, `t_`).
num_events	Returns the total number of recorded events.
record_variables	The additional variables that will be recorded
source	The source we are recording from

Methods

all_values()	Return a dictionary mapping recorded variable names (including `t`) to a dictionary mapping neuron indices to a
event_trains()	Return a dictionary mapping event indices to arrays of event times.
reinit()	Clears all recorded spikes
resize(new_size)	
values(var)	Return a dictionary mapping neuron indices to arrays of variable values at the time of the events (sorted by time

Details

event
The event that we are listening to

it
Returns the pair (`i`, `t`).

it_
Returns the pair (`i`, `t_`).

num_events

Returns the total number of recorded events.

record_variables
The additional variables that will be recorded

source
The source we are recording from

all_values()
Return a dictionary mapping recorded variable names (including `t`) to a dictionary mapping neuron indices to arrays of variable values at the time of the events (sorted by time). This is equivalent to (but more efficient than) calling *values* for each variable and storing the result in a dictionary.

Returns **all_values** : dict

Dictionary mapping variable names to dictionaries which themselves are mapping neuron indicies to arrays of variable values at the time of the events.

Examples

```
>>> from brian2 import *
>>> G = NeuronGroup(2, """dv/dt = 100*Hz : 1
...                       v_th : 1""", threshold='v>v_th', reset='v=0')
>>> G.v_th = [0.5, 1]
>>> mon = EventMonitor(G, event='spike', variables='v')
>>> run(20*ms)
>>> all_values = mon.all_values()
>>> all_values['t'][0]
array([  4.9,   9.9,  14.9,  19.9]) * msecond
>>> all_values['v'][0]
array([ 0.5,  0.5,  0.5,  0.5])
```

event_trains()
Return a dictionary mapping event indices to arrays of event times. Equivalent to calling `values('t')`.

Returns **event_trains** : dict

Dictionary that stores an array with the event times for each neuron index.

See also:

SpikeMonitor.spike_trains()

reinit()
Clears all recorded spikes

resize(*new_size*)

values(*var*)
Return a dictionary mapping neuron indices to arrays of variable values at the time of the events (sorted by time). Parameters ———- var : str

The name of the variable.

Returns **values** : dict

Dictionary mapping each neuron index to an array of variable values at the time of the events

Examples

```
>>> from brian2 import *
>>> G = NeuronGroup(2, """dv/dt = 100*Hz : 1
...                       v_th : 1""", threshold='v>v_th', reset='v=0')
>>> G.v_th = [0.5, 1]
>>> mon = EventMonitor(G, event='spike', variables='v')
>>> run(20*ms)
>>> v_values = mon.values('v')
>>> v_values[0]
array([ 0.5,  0.5,  0.5,  0.5])
>>> v_values[1]
array([ 1.,  1.])
```

SpikeMonitor(source[, variables, when, ...])	Record spikes from a *NeuronGroup* or other spike source.

SpikeMonitor class

(*Shortest import*: `from brian2 import SpikeMonitor`)

class `brian2.monitors.spikemonitor.`**SpikeMonitor**(*source*, *variables=None*, *when=None*, *order=None*, *name='spikemonitor*'*, *codeobj_class=None*)

Bases: *brian2.monitors.spikemonitor.EventMonitor*

Record spikes from a *NeuronGroup* or other spike source.

The recorded spikes can be accessed in various ways (see Examples below): the attributes `i` and `t` store all the indices and spike times, respectively. Alternatively, you can get a dictionary mapping neuron indices to spike trains, by calling the *spike_trains* method. If you record additional variables with the `variables` argument, these variables can be accessed by their name (see Examples).

Parameters **source** : (*NeuronGroup*, *SpikeSource*)

The source of spikes to record.

record : bool

Whether or not to record each spike in `i` and `t` (the `count` will always be recorded).

variables : str or sequence of str, optional

Which variables to record at the time of the spike (in addition to the index of the neuron). Can be the name of a variable or a list of names.

when : str, optional

When to record the events, by default records events in the same slot where the event is emitted.

order : int, optional

The priority of of this group for operations occurring at the same time step and in the same scheduling slot. Defaults to the order where the event is emitted + 1, i.e. it will be recorded directly afterwards.

name : str, optional

A unique name for the object, otherwise will use `source.name+'_spikemonitor_0'`, etc.

codeobj_class : class, optional

The *CodeObject* class to run code with.

Examples

```
>>> from brian2 import *
>>> spikes = SpikeGeneratorGroup(3, [0, 1, 2], [0, 1, 2]*ms)
>>> spike_mon = SpikeMonitor(spikes)
>>> net = Network(spikes, spike_mon)
>>> net.run(3*ms)
>>> print(spike_mon.i[:])
[0 1 2]
>>> print(spike_mon.t[:])
[ 0.  1.  2.] ms
>>> print(spike_mon.t_[:])
[ 0.     0.001  0.002]
>>> G = NeuronGroup(1, """dv/dt = (1 - v)/(10*ms) : 1
...                       dv_th/dt = (0.5 - v_th)/(20*ms) : 1""",
...                 threshold='v>v_th',
...                 reset='v = 0; v_th += 0.1')
>>> crossings = SpikeMonitor(G, variables='v', name='crossings')
>>> net = Network(G, crossings)
>>> net.run(10*ms)
>>> crossings.t
<crossings.t: array([ 0. ,  1.4,  4.6,  9.7]) * msecond>
>>> crossings.v
<crossings.v: array([ 0.00995017,  0.13064176,  0.27385096,  0.39950442])>
```

Attributes

num_spikes	Returns the total number of recorded spikes.

Methods

all_values()	Return a dictionary mapping recorded variable names (including `t`) to a dictionary mapping neuron indices to a
spike_trains()	Return a dictionary mapping spike indices to arrays of spike times.
values(var)	Return a dictionary mapping neuron indices to arrays of variable values at the time of the spikes (sorted by time

Details

num_spikes

Returns the total number of recorded spikes.

all_values()

Return a dictionary mapping recorded variable names (including `t`) to a dictionary mapping neuron indices to arrays of variable values at the time of the spikes (sorted by time). This is equivalent to (but more efficient than) calling *values* for each variable and storing the result in a dictionary.

Returns **all_values** : dict

Dictionary mapping variable names to dictionaries which themselves are mapping neu-

ron indicies to arrays of variable values at the time of the spikes.

Examples

```
>>> from brian2 import *
>>> G = NeuronGroup(2, """dv/dt = 100*Hz : 1
...                       v_th : 1""", threshold='v>v_th', reset='v=0')
>>> G.v_th = [0.5, 1]
>>> mon = SpikeMonitor(G, variables='v')
>>> run(20*ms)
>>> all_values = mon.all_values()
>>> all_values['t'][0]
array([  4.9,   9.9,  14.9,  19.9]) * msecond
>>> all_values['v'][0]
array([ 0.5,  0.5,  0.5,  0.5])
```

spike_trains()

Return a dictionary mapping spike indices to arrays of spike times.

Returns **spike_trains** : dict

Dictionary that stores an array with the spike times for each neuron index.

Examples

```
>>> from brian2 import *
>>> spikes = SpikeGeneratorGroup(3, [0, 1, 2], [0, 1, 2]*ms)
>>> spike_mon = SpikeMonitor(spikes)
>>> run(3*ms)
>>> spike_trains = spike_mon.spike_trains()
>>> spike_trains[1]
array([ 1.]) * msecond
```

values(*var*)

Return a dictionary mapping neuron indices to arrays of variable values at the time of the spikes (sorted by time).

Parameters **var** : str

The name of the variable.

Returns **values** : dict

Dictionary mapping each neuron index to an array of variable values at the time of the spikes.

Examples

```
>>> from brian2 import *
>>> G = NeuronGroup(2, """dv/dt = 100*Hz : 1
...                       v_th : 1""", threshold='v>v_th', reset='v=0')
>>> G.v_th = [0.5, 1]
>>> mon = SpikeMonitor(G, variables='v')
>>> run(20*ms)
>>> v_values = mon.values('v')
>>> v_values[0]
```

```
        array([ 0.5,  0.5,  0.5,  0.5])
        >>> v_values[1]
        array([ 1.,  1.])
```

Tutorials and examples using this

- Tutorial 1-intro-to-brian-neurons
- Example reliability
- Example adaptive_threshold
- Example non_reliability
- Example IF_curve_LIF
- Example IF_curve_Hodgkin_Huxley
- Example phase_locking
- Example CUBA
- Example compartmental/hh_with_spikes
- Example advanced/opencv_movie
- Example standalone/STDP_standalone
- Example standalone/cuba_openmp
- Example synapses/STDP
- Example synapses/jeffress
- Example synapses/licklider
- Example frompapers/Brette_Gerstner_2005
- Example frompapers/Brette_Guigon_2003
- Example frompapers/Diesmann_et_al_1999
- Example frompapers/Rossant_et_al_2011bis
- Example frompapers/Brette_2004
- Example frompapers/Brunel_Hakim_1999
- Example frompapers/Touboul_Brette_2008
- Example frompapers/Vogels_et_al_2011
- Example frompapers/Sturzl_et_al_2000
- Example frompapers/Brette_2012/Fig5A

`statemonitor` module

Exported members: `StateMonitor`

Classes

`StateMonitor`(source, variables[, record, ...])	Record values of state variables during a run

StateMonitor class

(*Shortest import*: `from brian2 import StateMonitor`)

class `brian2.monitors.statemonitor.`**`StateMonitor`**(*source*, *variables*, *record=None*, *dt=None*, *clock=None*, *when='start'*, *order=0*, *name='statemonitor*'*, *codeobj_class=None*)

Bases: *brian2.groups.group.Group*, *brian2.groups.group.CodeRunner*

Record values of state variables during a run

To extract recorded values after a run, use the `t` attribute for the array of times at which values were recorded, and variable name attribute for the values. The values will have shape `(len(indices), len(t))`, where `indices` are the array indices which were recorded. When indexing the *StateMonitor* directly, the returned object can be used to get the recorded values for the specified indices, i.e. the indexing semantic refers to the indices in `source`, not to the relative indices of the recorded values. For example, when recording only neurons with even numbers, `mon[[0, 2]].v` will return the values for neurons 0 and 2, whereas `mon.v[[0, 2]]` will return the values for the first and third *recorded* neurons, i.e. for neurons 0 and 4.

Parameters **source** : *Group*

Which object to record values from.

variables : str, sequence of str, True

Which variables to record, or `True` to record all variables (note that this may use a great deal of memory).

record : None, False, True, sequence of ints, optional

Which indices to record, nothing is recorded for `None` or `False`, everything is recorded for `True` (warning: may use a great deal of memory), or a specified subset of indices. Defaults to `None`.

dt : *Quantity*, optional

The time step to be used for the monitor. Cannot be combined with the `clock` argument.

clock : *Clock*, optional

The update clock to be used. If neither a clock, nor the `dt` argument is specified, the clock of the `source()` will be used.

when : str, optional

At which point during a time step the values should be recorded. Defaults to `'start'`.

order : int, optional

The priority of of this group for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

name : str, optional

A unique name for the object, otherwise will use `source.name+'statemonitor_0'`, etc.

codeobj_class : *CodeObject*, optional

The *CodeObject* class to create.

Notes

Since this monitor by default records in the `'start'` time slot, recordings of the membrane potential in integrate-and-fire models may look unexpected: the recorded membrane potential trace will never be above threshold in an integrate-and-fire model, because the reset statement will have been applied already. Set the `when` keyword to a different value if this is not what you want.

Examples

Record all variables, first 5 indices:

```
eqs = """
dV/dt = (2-V)/(10*ms) : 1
"""
threshold = 'V>1'
reset = 'V = 0'
G = NeuronGroup(100, eqs, threshold=threshold, reset=reset)
G.V = rand(len(G))
M = StateMonitor(G, True, record=range(5))
run(100*ms)
plot(M.t, M.V.T)
show()
```

Attributes

record	The array of recorded indices
record_variables	The variables to record

Methods

record_single_timestep()	Records a single time step.
reinit()	
resize(new_size)	

Details

record
The array of recorded indices

record_variables
The variables to record

record_single_timestep()
Records a single time step. Useful for recording the values at the end of the simulation – otherwise a *StateMonitor* will not record the last simulated values since its `when` attribute defaults to `'start'`, i.e. the last recording is at the *beginning* of the last time step.

Notes

This function will only work if the *StateMonitor* has been already run, but a run with a length of `0*ms` does suffice.

Examples

```
>>> from brian2 import *
>>> G = NeuronGroup(1, 'dv/dt = -v/(5*ms) : 1')
>>> G.v = 1
>>> mon = StateMonitor(G, 'v', record=True)
>>> run(0.5*ms)
>>> mon.v
array([[ 1.        ,  0.98019867,  0.96078944,  0.94176453,  0.92311635]])
>>> mon.t[:]
array([   0.,  100.,  200.,  300.,  400.]) * usecond
>>> G.v[:]  # last value had not been recorded
array([ 0.90483742])
>>> mon.record_single_timestep()
>>> mon.t[:]
array([   0.,  100.,  200.,  300.,  400.,  500.]) * usecond
>>> mon.v[:]
array([[ 1.        ,  0.98019867,  0.96078944,  0.94176453,  0.92311635,
         0.90483742]])
```

reinit()

resize(*new_size*)

Tutorials and examples using this

- Tutorial 1-intro-to-brian-neurons
- Tutorial 2-intro-to-brian-synapses
- Example adaptive_threshold
- Example phase_locking
- Example COBAHH
- Example compartmental/infinite_cable
- Example compartmental/bipolar_cell
- Example compartmental/hh_with_spikes
- Example compartmental/hodgkin_huxley_1952
- Example compartmental/lfp
- Example compartmental/bipolar_with_inputs2
- Example compartmental/spike_initiation
- Example compartmental/bipolar_with_inputs
- Example advanced/stochastic_odes
- Example standalone/STDP_standalone
- Example synapses/nonlinear

- Example synapses/synapses
- Example synapses/STDP
- Example synapses/jeffress
- Example synapses/gapjunctions
- Example frompapers/Brette_Gerstner_2005
- Example frompapers/Brette_Guigon_2003
- Example frompapers/Rossant_et_al_2011bis
- Example frompapers/Rothman_Manis_2003
- Example frompapers/Touboul_Brette_2008
- Example frompapers/Wang_Buszaki_1996
- Example frompapers/Brette_2012/Fig1
- Example frompapers/Brette_2012/Fig5A
- Example frompapers/Brette_2012/Fig3CF
- Example frompapers/Brette_2012/Fig3AB
- Example frompapers/Brette_2012/Fig4

StateMonitorView(monitor, item)

StateMonitorView class

(*Shortest import*: `from brian2.monitors.statemonitor import StateMonitorView`)

class `brian2.monitors.statemonitor.`**`StateMonitorView`**(*monitor*, *item*)
Bases: `object`

6.4.9 parsing package

`dependencies` module

Exported members: *abstract_code_dependencies*

Functions

abstract_code_dependencies(code[, ...])	Analyses identifiers used in abstract code blocks

abstract_code_dependencies function

(*Shortest import*: `from brian2.parsing.dependencies import abstract_code_dependencies`)

`brian2.parsing.dependencies.`**`abstract_code_dependencies`**(*code*, *known_vars=None*, *known_funcs=None*)

Analyses identifiers used in abstract code blocks

Parameters **code** : str

The abstract code block.

known_vars : set

The set of known variable names.

known_funcs : set

The set of known function names.

Returns **results** : namedtuple with the following fields

all The set of all identifiers that appear in this code block, including functions.

read The set of values that are read, excluding functions.

write The set of all values that are written to.

funcs The set of all function names.

known_all The set of all identifiers that appear in this code block and are known.

known_read The set of known values that are read, excluding functions.

known_write The set of known values that are written to.

known_funcs The set of known functions that are used.

unknown_read The set of all unknown variables whose values are read. Equal to `read-known_vars`.

unknown_write The set of all unknown variables written to. Equal to `write-known_vars`.

unknown_funcs The set of all unknown function names, equal to `funcs-known_funcs`.

undefined_read The set of all unknown variables whose values are read before they are written to. If this set is nonempty it usually indicates an error, since a variable that is read should either have been defined in the code block (in which case it will appear in `newly_defined`) or already be known.

newly_defined The set of all variable names which are newly defined in this abstract code block.

get_read_write_funcs(parsed_code)

get_read_write_funcs function

(*Shortest import*: `from brian2.parsing.dependencies import get_read_write_funcs`)

`brian2.parsing.dependencies.`**`get_read_write_funcs`**(*parsed_code*)

`expressions` module

AST parsing based analysis of expressions

Exported members: *is_boolean_expression*, *parse_expression_unit*

Functions

is_boolean_expression(expr, variables)	Determines if an expression is of boolean type or not

is_boolean_expression function

(*Shortest import*: `from brian2.parsing.expressions import is_boolean_expression`)

`brian2.parsing.expressions.`**`is_boolean_expression`**(*expr*, *variables*)

Determines if an expression is of boolean type or not

Parameters **expr** : str

The expression to test

variables : dict-like of `Variable`

The variables used in the expression.

Returns **isbool** : bool

Whether or not the expression is boolean.

Raises

`SyntaxError` If the expression ought to be boolean but is not, for example `x<y and z` where `z` is not a boolean variable.

Notes

We test the following cases recursively on the abstract syntax tree:

- The node is a boolean operation. If all the subnodes are boolean expressions we return `True`, otherwise we raise the `SyntaxError`.
- The node is a function call, we return `True` or `False` depending on whether the function description has the `_returns_bool` attribute set.
- The node is a variable name, we return `True` or `False` depending on whether `is_boolean` attribute is set or if the name is `True` or `False`.
- The node is a comparison, we return `True`.
- The node is a unary operation, we return `True` if the operation is `not`, otherwise `False`.
- Otherwise we return `False`.

parse_expression_unit(expr, variables)	Returns the unit value of an expression, and checks its validity

parse_expression_unit function

(*Shortest import*: `from brian2.parsing.expressions import parse_expression_unit`)

`brian2.parsing.expressions.`**`parse_expression_unit`**(*expr*, *variables*)

Returns the unit value of an expression, and checks its validity

Parameters **expr** : str

The expression to check.

variables : dict

Dictionary of all variables used in the `expr` (including `Constant` objects for external variables)

Returns **unit** : Quantity

The output unit of the expression

Raises

SyntaxError If the expression cannot be parsed, or if it uses `a**b` for `b` anything other than a constant number.

DimensionMismatchError If any part of the expression is dimensionally inconsistent.

`functions` module

Exported members: *AbstractCodeFunction*, *abstract_code_from_function*, *extract_abstract_code_functions*, *substitute_abstract_code_functions*

Classes

AbstractCodeFunction(name, args, code, ...)	The information defining an abstract code function

AbstractCodeFunction class

(*Shortest import*: `from brian2.parsing.functions import AbstractCodeFunction`)

class `brian2.parsing.functions.`**`AbstractCodeFunction`**(*name*, *args*, *code*, *return_expr*)

Bases: `object`

The information defining an abstract code function

Has attributes corresponding to initialisation parameters

Parameters **name** : str

The function name.

args : list of str

The arguments to the function.

code : str

The abstract code string consisting of the body of the function less the return statement.

return_expr : str or None

The expression returned, or None if there is nothing returned.

FunctionRewriter(func[, numcalls])	Inlines a function call using temporary variables

FunctionRewriter class

(*Shortest import*: `from brian2.parsing.functions import FunctionRewriter`)

class `brian2.parsing.functions.`**`FunctionRewriter`**(*func*, *numcalls=0*)

Bases: `ast.NodeTransformer`

Inlines a function call using temporary variables

numcalls is the number of times the function rewriter has been called so far, this is used to make sure that when

recursively inlining there is no name aliasing. The substitute_abstract_code_functions ensures that this is kept up to date between recursive runs.

The pre attribute is the set of lines to be inserted above the currently being processed line, i.e. the inline code.

The visit method returns the current line processed so that the function call is replaced with the output of the inlining.

Methods

visit_Call(node)	

Details

visit_Call (*node*)

VarRewriter(pre)	Rewrites all variable names in names by prepending pre

VarRewriter class

(*Shortest import*: `from brian2.parsing.functions import VarRewriter`)

class `brian2.parsing.functions.`**VarRewriter** (*pre*)
Bases: `ast.NodeTransformer`

Rewrites all variable names in names by prepending pre

Methods

visit_Call(node)	
visit_Name(node)	

Details

visit_Call (*node*)

visit_Name (*node*)

Functions

abstract_code_from_function(func)	Converts the body of the function to abstract code

abstract_code_from_function function

(*Shortest import*: `from brian2.parsing.functions import abstract_code_from_function`)

`brian2.parsing.functions.`**abstract_code_from_function** (*func*)
Converts the body of the function to abstract code

Parameters **func** : function, str or ast.FunctionDef

The function object to convert. Note that the arguments to the function are ignored.

Returns **func** : AbstractCodeFunction

The corresponding abstract code function

Raises

SyntaxError If unsupported features are used such as if statements or indexing.

extract_abstract_code_functions(code)	Returns a set of abstract code functions from function definitions.

extract_abstract_code_functions function

(*Shortest import*: `from brian2.parsing.functions import extract_abstract_code_functions`)

brian2.parsing.functions.**extract_abstract_code_functions**(*code*)

Returns a set of abstract code functions from function definitions.

Returns all functions defined at the top level and ignores any other code in the string.

Parameters **code** : str

The code string defining some functions.

Returns **funcs** : dict

A mapping `(name, func)` for `func` an *AbstractCodeFunction*.

substitute_abstract_code_functions(code, funcs)	Performs inline substitution of all the functions in the code

substitute_abstract_code_functions function

(*Shortest import*: `from brian2.parsing.functions import substitute_abstract_code_functions`)

brian2.parsing.functions.**substitute_abstract_code_functions**(*code*, *funcs*)

Performs inline substitution of all the functions in the code

Parameters **code** : str

The abstract code to make inline substitutions into.

funcs : list, dict or set of AbstractCodeFunction

The function substitutions to use, note in the case of a dict, the keys are ignored and the function name is used.

Returns **code** : str

The code with inline substitutions performed.

rendering module

Exported members: *NodeRenderer*, *NumpyNodeRenderer*, *CPPNodeRenderer*, *SympyNodeRenderer*

Classes

CPPNodeRenderer([use_vectorisation_idx]) **Methods**

CPPNodeRenderer class

(*Shortest import*: `from brian2.parsing.rendering import CPPNodeRenderer`)

class `brian2.parsing.rendering.`**CPPNodeRenderer**(*use_vectorisation_idx=True*)
Bases: *brian2.parsing.rendering.NodeRenderer*

Methods

render_Assign(node)
render_BinOp(node)
render_Name(node)
render_NameConstant(node)

Details

render_Assign(*node*)

render_BinOp(*node*)

render_Name(*node*)

render_NameConstant(*node*)

NodeRenderer([use_vectorisation_idx]) **Methods**

NodeRenderer class

(*Shortest import*: `from brian2.parsing.rendering import NodeRenderer`)

class `brian2.parsing.rendering.`**NodeRenderer**(*use_vectorisation_idx=True*)
Bases: `object`

Methods

render_Assign(node)
render_AugAssign(node)
render_BinOp(node)
render_BinOp_parentheses(left, right, op)
render_BoolOp(node)
render_Call(node)
render_Compare(node)
render_Name(node)
render_NameConstant(node)
render_Num(node)

C

Table 6.290 – continued from previous page

render_UnaryOp(node)	
render_code(code)	
render_element_parentheses(node)	Render an element with parentheses around it or leave them away for numbers, n
render_expr(expr[, strip])	
render_func(node)	
render_node(node)	

Details

render_Assign(*node*)

render_AugAssign(*node*)

render_BinOp(*node*)

render_BinOp_parentheses(*left*, *right*, *op*)

render_BoolOp(*node*)

render_Call(*node*)

render_Compare(*node*)

render_Name(*node*)

render_NameConstant(*node*)

render_Num(*node*)

render_UnaryOp(*node*)

render_code(*code*)

render_element_parentheses(*node*)
Render an element with parentheses around it or leave them away for numbers, names and function calls.

render_expr(*expr*, *strip=True*)

render_func(*node*)

render_node(*node*)

NumpyNodeRenderer([use_vectorisation_idx]) **Methods**

NumpyNodeRenderer class

(*Shortest import*: `from brian2.parsing.rendering import NumpyNodeRenderer`)

class `brian2.parsing.rendering.`**NumpyNodeRenderer**(*use_vectorisation_idx=True*)
Bases: *brian2.parsing.rendering.NodeRenderer*

Methods

render_UnaryOp(node)

Details

render_UnaryOp(*node*)

SympyNodeRenderer([use_vectorisation_idx])	**Methods**

SympyNodeRenderer class

(*Shortest import*: `from brian2.parsing.rendering import SympyNodeRenderer`)

class `brian2.parsing.rendering.`**SympyNodeRenderer**(*use_vectorisation_idx=True*)
Bases: *brian2.parsing.rendering.NodeRenderer*

Methods

render_Compare(node)
render_Name(node)
render_Num(node)
render_func(node)

Details

render_Compare(*node*)

render_Name(*node*)

render_Num(*node*)

render_func(*node*)

statements module

Functions

parse_statement(code)	Parses a single line of code into "var op expr".

parse_statement function

(*Shortest import*: `from brian2.parsing.statements import parse_statement`)

`brian2.parsing.statements.`**parse_statement**(*code*)
Parses a single line of code into "var op expr".

Parameters **code** : str

A string containing a single statement of the form `var op expr # comment`, where the `# comment` part is optional.

Returns **var, op, expr, comment** : str, str, str, str

The four parts of the statement.

Examples

```
>>> parse_statement('v = -65*mV  # reset the membrane potential')
('v', '=', '-65*mV', 'reset the membrane potential')
>>> parse_statement('v += dt*(-v/tau)')
('v', '+=', 'dt*(-v/tau)', '')
```

sympytools module

Utility functions for parsing expressions and statements.

Classes

CustomSympyPrinter([settings])	Printer that overrides the printing of some basic sympy objects.

CustomSympyPrinter class

(*Shortest import*: `from brian2.parsing.sympytools import CustomSympyPrinter`)

class `brian2.parsing.sympytools.`**`CustomSympyPrinter`**(*settings=None*)

Bases: `sympy.printing.str.StrPrinter`

Printer that overrides the printing of some basic sympy objects. E.g. print "a and b" instead of "And(a, b)".

Functions

replace_constants(sympy_expr[, variables])	Replace constant values in a sympy expression with their numerical value.

replace_constants function

(*Shortest import*: `from brian2.parsing.sympytools import replace_constants`)

`brian2.parsing.sympytools.`**`replace_constants`**(*sympy_expr*, *variables=None*)

Replace constant values in a sympy expression with their numerical value.

Parameters **sympy_expr** : `sympy.Expr`

The expression

variables : dict-like, optional

Dictionary of `Variable` objects

Returns **new_expr** : `sympy.Expr`

Expressions with all constants replaced

str_to_sympy(expr)	Parses a string into a sympy expression.

str_to_sympy function

(*Shortest import*: `from brian2.parsing.sympytools import str_to_sympy`)

`brian2.parsing.sympytools.`**`str_to_sympy`**(*expr*)

Parses a string into a sympy expression. There are two reasons for not using `sympify` directly: 1) sympify

does a `from sympy import *`, adding all functions to its namespace. This leads to issues when trying to use sympy function names as variable names. For example, both `beta` and `factor` – quite reasonable names for variables – are sympy functions, using them as variables would lead to a parsing error. 2) We want to use a common syntax across expressions and statements, e.g. we want to allow to use `and` (instead of `&`) and function names like `ceil` (instead of `ceiling`).

Parameters **expr** : str

The string expression to parse..

Returns **s_expr** :

A sympy expression

Raises

`SyntaxError` In case of any problems during parsing.

Notes

Parsing is done in two steps: First, the expression is parsed and rendered as a new string by `SympyNodeRenderer`, translating function names (e.g. `ceil` to `ceiling`) and operator names (e.g. `and` to `&`), all unknown names are wrapped in `Symbol(...)` or `Function(...)`. The resulting string is then evaluated in the `from sympy import *` namespace.

sympy_to_str(sympy_expr)	Converts a sympy expression into a string.

sympy_to_str function

(*Shortest import*: `from brian2.parsing.sympytools import sympy_to_str`)

brian2.parsing.sympytools.**sympy_to_str**(*sympy_expr*)

Converts a sympy expression into a string. This could be as easy as `str(sympy_exp)` but it is possible that the sympy expression contains functions like `Abs` (for example, if an expression such as `sqrt(x**2)` appeared somewhere). We do want to re-translate `Abs` into `abs` in this case.

Parameters **sympy_expr** : sympy.core.expr.Expr

The expression that should be converted to a string.

Returns :

str_expr : str

A string representing the sympy expression.

Objects

PRINTER	Printer that overrides the printing of some basic sympy objects.

PRINTER object

(*Shortest import*: `from brian2.parsing.sympytools import PRINTER`)

brian2.parsing.sympytools.**PRINTER = <brian2.parsing.sympytools.CustomSympyPrinter object>**

Printer that overrides the printing of some basic sympy objects. E.g. print “a and b” instead of “And(a, b)”.

6.4.10 spatialneuron package

morphology module

Neuronal morphology module. This module defines classes to load and build neuronal morphologies.

Exported members: *Morphology*, *MorphologyData*, *Cylinder*, *Soma*

Classes

Cylinder(*args, **kwds)	A cylinder.

Cylinder class

(*Shortest import*: `from brian2 import Cylinder`)

class `brian2.spatialneuron.morphology.`**`Cylinder`**(**args*, ***kwds*)
Bases: *brian2.spatialneuron.morphology.Morphology*

A cylinder.

Parameters **length** : *Quantity*, optional

The total length in `meter`. If unspecified, inferred from `x`, `y`, `z`.

diameter : *Quantity*

The diameter in `meter`.

n : int, optional

Number of compartments (default 1).

type : str, optional

Type of segment, `soma`, 'axon' or 'dendrite'.

x : *Quantity*, optional

x position of end point in `meter` units. If not specified, inferred from `length` with a random direction.

y : *Quantity*, optional

x position of end point in `meter` units.

z : *Quantity*, optional

x position of end point in `meter` units.

Tutorials and examples using this

- Example compartmental/rall
- Example compartmental/infinite_cable
- Example compartmental/bipolar_cell
- Example compartmental/hh_with_spikes
- Example compartmental/hodgkin_huxley_1952
- Example compartmental/lfp

- Example compartmental/morphotest
- Example compartmental/bipolar_with_inputs2
- Example compartmental/cylinder
- Example compartmental/spike_initiation
- Example compartmental/bipolar_with_inputs
- Example frompapers/Brette_2012/Fig1
- Example frompapers/Brette_2012/Fig5A
- Example frompapers/Brette_2012/Fig3CF
- Example frompapers/Brette_2012/Fig3AB
- Example frompapers/Brette_2012/Fig4

Morphology([filename, n])	Neuronal morphology (=tree of branches).

Morphology class

(*Shortest import*: `from brian2 import Morphology`)

class `brian2.spatialneuron.morphology.`**Morphology** (*filename=None*, *n=None*)
Bases: `object`

Neuronal morphology (=tree of branches).

The data structure is a tree where each node is a segment consisting of a number of connected compartments, each one defined by its geometrical properties (length, area, diameter, position).

Parameters **filename** : str, optional

The name of a swc file defining the morphology. If not specified, makes a segment (if `n` is specified) or an empty morphology.

n : int, optional

Number of compartments.

Methods

compress(morphology_data[, origin])	Compresses the tree by changing the compartment vectors to views on a matrix (
create_from_segments(segments[, origin])	Recursively create the morphology from a list of segments.
loadswc(filename)	Reads a SWC file containing a neuronal morphology.
plot([axes, simple, origin])	Plots the morphology in 3D.
set_area()	Sets the area of compartments according to diameter and length
set_coordinates()	Sets the coordinates of compartments according to their lengths (taking
set_distance()	Sets the distance to the soma (or more generally start point of the
set_length()	Sets the length of compartments according to their coordinates

Details

compress (*morphology_data*, *origin=0*)
Compresses the tree by changing the compartment vectors to views on a matrix (or vectors). The morphol-

ogy cannot be changed anymore but all other functions should work normally. Units are discarded in the process.

origin : offset in the base matrix

create_from_segments(*segments*, *origin=0*)
Recursively create the morphology from a list of segments. Each segments has attributes: x,y,z,diameter,area,length (vectors) and children (list). It also creates a dictionary of names (_namedkid).

loadswc(*filename*)
Reads a SWC file containing a neuronal morphology. Large database at http://neuromorpho.org/neuroMorpho Information below from http://www.mssm.edu/cnic/swc.html

SWC File Format

The format of an SWC file is fairly simple. It is a text file consisting of a header with various fields beginning with a # character, and a series of three dimensional points containing an index, radius, type, and connectivity information. The lines in the text file representing points have the following layout. n T x y z R P n is an integer label that identifies the current point and increments by one from one line to the next. T is an integer representing the type of neuronal segment, such as soma, axon, apical dendrite, etc. The standard accepted integer values are given below: * 0 = undefined * 1 = soma * 2 = axon * 3 = dendrite * 4 = apical dendrite * 5 = fork point * 6 = end point * 7 = custom

x, y, z gives the cartesian coordinates of each node. R is the radius at that node. P indicates the parent (the integer label) of the current point or -1 to indicate an origin (soma).

By default, the soma is assumed to have spherical geometry. If several compartments

plot(*axes=None*, *simple=True*, *origin=None*)
Plots the morphology in 3D. Units are um.

Parameters **axes** : `Axes3D`

the figure axes (new figure if not given)

simple : bool, optional

if `True`, the diameter of branches is ignored (defaults to `True`)

set_area()
Sets the area of compartments according to diameter and length (assuming cylinders)

set_coordinates()
Sets the coordinates of compartments according to their lengths (taking a random direction)

set_distance()
Sets the distance to the soma (or more generally start point of the morphology)

set_length()
Sets the length of compartments according to their coordinates

Tutorials and examples using this

- Example compartmental/rall
- Example compartmental/infinite_cable
- Example compartmental/bipolar_cell
- Example compartmental/morphotest
- Example compartmental/bipolar_with_inputs2
- Example compartmental/cylinder

- Example compartmental/spike_initiation
- Example compartmental/bipolar_with_inputs
- Example frompapers/Brette_2012/Fig1
- Example frompapers/Brette_2012/Fig5A
- Example frompapers/Brette_2012/Fig3CF
- Example frompapers/Brette_2012/Fig3AB
- Example frompapers/Brette_2012/Fig4

MorphologyData(N)	

MorphologyData class

(*Shortest import*: `from brian2 import MorphologyData`)

class `brian2.spatialneuron.morphology.`**`MorphologyData`**(*N*)
Bases: `object`

MorphologyIndexWrapper(morphology)	A simpler version of *IndexWrapper*, not allowing for string indexing (*Morphol*

MorphologyIndexWrapper class

(*Shortest import*: `from brian2.spatialneuron.morphology import MorphologyIndexWrapper`)

class `brian2.spatialneuron.morphology.`**`MorphologyIndexWrapper`**(*morphology*)
Bases: `object`

A simpler version of *IndexWrapper*, not allowing for string indexing (*Morphology* is not a *Group*). It allows to use `morphology.indices[...]` instead of `morphology[...]._indices()`.

Soma(*args, **kwds)	A spherical soma.

Soma class

(*Shortest import*: `from brian2 import Soma`)

class `brian2.spatialneuron.morphology.`**`Soma`**(**args*, ***kwds*)
Bases: *brian2.spatialneuron.morphology.Morphology*

A spherical soma.

Parameters **diameter** : *Quantity*, optional

Diameter of the sphere.

Tutorials and examples using this

- Example compartmental/bipolar_cell
- Example compartmental/morphotest
- Example compartmental/bipolar_with_inputs2

- Example compartmental/spike_initiation
- Example compartmental/bipolar_with_inputs
- Example frompapers/Brette_2012/Fig1
- Example frompapers/Brette_2012/Fig5A
- Example frompapers/Brette_2012/Fig3CF
- Example frompapers/Brette_2012/Fig3AB
- Example frompapers/Brette_2012/Fig4

`spatialneuron` module

Compartmental models. This module defines the SpatialNeuron class, which defines multicompartmental models.

Exported members: `SpatialNeuron`

Classes

`SpatialNeuron`([morphology, model, ...])	A single neuron with a morphology and possibly many compartments.

SpatialNeuron class

(*Shortest import*: `from brian2 import SpatialNeuron`)

class `brian2.spatialneuron.spatialneuron.`**`SpatialNeuron`**(*morphology=None*, *model=None*, *threshold=None*, *refractory=False*, *reset=None*, *threshold_location=None*, *dt=None*, *clock=None*, *order=0*, *Cm=0.009 * metre ** -4 * kilogram ** -1 * second ** 4 * amp ** 2*, *Ri=1.5 * metre ** 3 * kilogram * second ** -3 * amp ** -2*, *name='spatialneuron*'*, *dtype=None*, *namespace=None*, *method=('linear'*, *'exponential_euler'*, *'rk2'*, *'milstein')*)

Bases: `brian2.groups.neurongroup.NeuronGroup`

A single neuron with a morphology and possibly many compartments.

Parameters **morphology** : `Morphology`

The morphology of the neuron.

model : (str, `Equations`)

The equations defining the group.

method : (str, function), optional

The numerical integration method. Either a string with the name of a registered method (e.g. "euler") or a function that receives an `Equations` object and returns the corresponding abstract code. If no method is specified, a suitable method will be chosen

automatically.

threshold : str, optional

The condition which produces spikes. Should be a single line boolean expression.

threshold_location : (int, *Morphology*), optional

Compartment where the threshold condition applies, specified as an integer (compartment index) or a *Morphology* object corresponding to the compartment (e.g. `morpho.axon[10*um]`). If unspecified, the threshold condition applies at all compartments.

Cm : *Quantity*, optional

Specific capacitance in uF/cm**2 (default 0.9). It can be accessed and modified later as a state variable. In particular, its value can differ in different compartments.

Ri : *Quantity*, optional

Intracellular resistivity in ohm.cm (default 150). It can be accessed as a shared state variable, but modified only before the first run. It is uniform across the neuron.

reset : str, optional

The (possibly multi-line) string with the code to execute on reset.

refractory : {str, *Quantity*}, optional

Either the length of the refractory period (e.g. `2*ms`), a string expression that evaluates to the length of the refractory period after each spike (e.g. `'(1 + rand())*ms'`), or a string expression evaluating to a boolean value, given the condition under which the neuron stays refractory after a spike (e.g. `'v > -20*mV'`)

namespace : dict, optional

A dictionary mapping variable/function names to the respective objects. If no `namespace` is given, the "implicit" namespace, consisting of the local and global namespace surrounding the creation of the class, is used.

dtype : (`dtype`, `dict`), optional

The `numpy.dtype` that will be used to store the values, or a dictionary specifying the type for variable names. If a value is not provided for a variable (or no value is provided at all), the preference setting *core.default_float_dtype* is used.

dt : *Quantity*, optional

The time step to be used for the simulation. Cannot be combined with the `clock` argument.

clock : *Clock*, optional

The update clock to be used. If neither a clock, nor the `dt` argument is specified, the *defaultclock* will be used.

order : int, optional

The priority of of this group for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

name : str, optional

A unique name for the group, otherwise use `spatialneuron_0`, etc.

Methods

spatialneuron_attribute(neuron, x)	Selects a subtree from *SpatialNeuron* neuron and returns a *SpatialSubgrou*
spatialneuron_segment(neuron, x)	Selects a segment from *SpatialNeuron* neuron, where x is a slice of either compa

Details

static spatialneuron_attribute(*neuron*, *x*)
Selects a subtree from *SpatialNeuron* neuron and returns a *SpatialSubgroup*. If it does not exist, returns the *Group* attribute.

static spatialneuron_segment(*neuron*, *x*)
Selects a segment from *SpatialNeuron* neuron, where x is a slice of either compartment indexes or distances. Note a: segment is not a *SpatialNeuron*, only a *Group*.

Tutorials and examples using this

- Example compartmental/rall
- Example compartmental/infinite_cable
- Example compartmental/bipolar_cell
- Example compartmental/hh_with_spikes
- Example compartmental/hodgkin_huxley_1952
- Example compartmental/lfp
- Example compartmental/morphotest
- Example compartmental/bipolar_with_inputs2
- Example compartmental/cylinder
- Example compartmental/spike_initiation
- Example compartmental/bipolar_with_inputs
- Example frompapers/Brette_2012/Fig1
- Example frompapers/Brette_2012/Fig5A
- Example frompapers/Brette_2012/Fig3CF
- Example frompapers/Brette_2012/Fig3AB
- Example frompapers/Brette_2012/Fig4

SpatialStateUpdater(group, method, clock[, ...])	The *CodeRunner* that updates the state variables of a *SpatialNeuron*

SpatialStateUpdater class

(*Shortest import*: `from brian2.spatialneuron.spatialneuron import SpatialStateUpdater`)

class `brian2.spatialneuron.spatialneuron.`**SpatialStateUpdater**(*group*, *method*, *clock*, *order=0*)
Bases: *brian2.groups.group.CodeRunner*, *brian2.groups.group.Group*

The *CodeRunner* that updates the state variables of a *SpatialNeuron* at every timestep.

TODO: all internal variables (u_minus etc) could be inserted in the SpatialNeuron.

Methods

before_run([run_namespace, level])	
number_branches(morphology[, n, parent])	Recursively number the branches and return their total number.

Details

before_run(*run_namespace=None*, *level=0*)

number_branches(*morphology*, *n=0*, *parent=-1*)
Recursively number the branches and return their total number. n is the index number of the current branch. parent is the index number of the parent branch.

SpatialSubgroup(source, start, stop, morphology)	A subgroup of a *SpatialNeuron*.

SpatialSubgroup class

(*Shortest import*: `from brian2.spatialneuron.spatialneuron import SpatialSubgroup`)

class `brian2.spatialneuron.spatialneuron.`**SpatialSubgroup**(*source*, *start*, *stop*, *morphology*, *name=None*)

Bases: *brian2.groups.subgroup.Subgroup*

A subgroup of a *SpatialNeuron*.

Parameters **source** : int

First compartment.

stop : int

Ending compartment, not included (as in slices).

morphology : *Morphology*

Morphology corresponding to the subgroup (not the full morphology).

name : str, optional

Name of the subgroup.

6.4.11 stateupdaters package

Module for transforming model equations into "abstract code" that can be then be further translated into executable code by the `codegen` module.

base module

This module defines the *StateUpdateMethod* class that acts as a base class for all stateupdaters and allows to register stateupdaters so that it is able to return a suitable stateupdater object for a given set of equations. This is used for example in *NeuronGroup* when no state updater is given explicitly.

Exported members: *StateUpdateMethod*

Classes

StateUpdateMethod	**Attributes**

StateUpdateMethod class

(*Shortest import*: `from brian2 import StateUpdateMethod`)

class `brian2.stateupdaters.base.`**`StateUpdateMethod`**
Bases: `object`

Attributes

stateupdaters	A dictionary mapping state updater names to *StateUpdateMethod* objects

Methods

__call__(equations[, variables])	Generate abstract code from equations.
can_integrate(equations, variables)	Determine whether the state updater is a suitable choice.
determine_stateupdater(equations, variables, ...)	Determine a suitable state updater.
register(name, stateupdater)	Register a state updater.

Details

`stateupdaters`
A dictionary mapping state updater names to *StateUpdateMethod* objects

`__call__`(*equations*, *variables=None*)
Generate abstract code from equations. The method also gets the the variables because some state updaters have to check whether variable names reflect other state variables (which can change from timestep to timestep) or are external values (which stay constant during a run) For convenience, this arguments are optional – this allows to directly see what code a state updater generates for a set of equations by simply writing `euler(eqs)`, for example.

Parameters **equations** : *Equations*

The model equations.

variables : dict, optional

The `Variable` objects for the model variables.

Returns **code** : str

The abstract code performing a state update step.

`can_integrate`(*equations*, *variables*)
Determine whether the state updater is a suitable choice. Should return `False` if it is not appropriate (e.g. non-linear equations for a linear state updater) and a `True` if it is appropriate.

Parameters **equations** : *Equations*

The model equations.

variables : dict

The `Variable` objects for the model variables.

Returns **ability** : bool

`True` if this state updater is able to integrate the given equations, `False` otherwise.

static `determine_stateupdater`(*equations*, *variables*, *method*)
Determine a suitable state updater. If a `method` is given, the state updater with the given name is used. In case it is a callable, it will be used even if it is a state updater that claims it is not applicable. If it is a string, the state updater registered with that name will be used, but in this case an error will be raised if it claims not to be applicable. If a `method` is a list of names, all the methods will be tried until one that can integrate the equations is found.

Parameters **equations** : *Equations*

The model equations.

variables : dict

The dictionary of `Variable` objects, describing the internal model variables.

method : {callable, str, list of str}

A callable usable as a state updater, the name of a registered state updater or a list of names of state updaters.

static `register`(*name*, *stateupdater*)
Register a state updater. Registered state updaters can be referred to via their name.

Parameters **name** : str

A short name for the state updater (e.g. `'euler'`)

stateupdater : `StateUpdaterMethod`

The state updater object, e.g. an *ExplicitStateUpdater*.

`exact` module

Exact integration for linear equations.

Exported members: *linear*, *independent*

Classes

IndependentStateUpdater	A state update for equations that do not depend on other state variables, i.e.

IndependentStateUpdater class

(*Shortest import*: `from brian2.stateupdaters.exact import IndependentStateUpdater`)

class `brian2.stateupdaters.exact.`**`IndependentStateUpdater`**
Bases: *brian2.stateupdaters.base.StateUpdateMethod*

A state update for equations that do not depend on other state variables, i.e. 1-dimensional differential equations. The individual equations are solved by sympy.

Methods

__call__(equations[, variables])	
can_integrate(equations, variables)	

Details

__call__(*equations*, *variables=None*)

can_integrate(*equations*, *variables*)

LinearStateUpdater	A state updater for linear equations.

LinearStateUpdater class

(*Shortest import*: `from brian2.stateupdaters.exact import LinearStateUpdater`)

class `brian2.stateupdaters.exact.`**LinearStateUpdater**
Bases: *brian2.stateupdaters.base.StateUpdateMethod*

A state updater for linear equations. Derives a state updater step from the analytical solution given by sympy. Uses the matrix exponential (which is only implemented for diagonalizable matrices in sympy).

Methods

__call__(equations[, variables])	
can_integrate(equations, variables)	

Details

__call__(*equations*, *variables=None*)

can_integrate(*equations*, *variables*)

Functions

get_linear_system(eqs)	Convert equations into a linear system using sympy.

get_linear_system function

(*Shortest import*: `from brian2.stateupdaters.exact import get_linear_system`)

`brian2.stateupdaters.exact.`**get_linear_system**(*eqs*)
Convert equations into a linear system using sympy.

Parameters **eqs** : *Equations*

The model equations.

Returns **(diff_eq_names, coefficients, constants)** : (list of str, `sympy.Matrix`, `sympy.Matrix`)

A tuple containing the variable names (`diff_eq_names`) corresponding to the rows of the matrix `coefficients` and the vector `constants`, representing the system of equations in the form M * X + B

Raises

`ValueError` If the equations cannot be converted into an M * X + B form.

Objects

independent	A state update for equations that do not depend on other state variables, i.e.

independent object

(*Shortest import*: `from brian2 import independent`)

`brian2.stateupdaters.exact.`**`independent`** **= <brian2.stateupdaters.exact.IndependentStateUpdater object>**

A state update for equations that do not depend on other state variables, i.e. 1-dimensional differential equations. The individual equations are solved by sympy.

linear	A state updater for linear equations.

linear object

(*Shortest import*: `from brian2 import linear`)

`brian2.stateupdaters.exact.`**`linear`** **= LinearStateUpdater()**

A state updater for linear equations. Derives a state updater step from the analytical solution given by sympy. Uses the matrix exponential (which is only implemented for diagonalizable matrices in sympy).

`explicit` module

Numerical integration functions.

Exported members: *milstein*, *euler*, *rk2*, *rk4*, *ExplicitStateUpdater*

Classes

ExplicitStateUpdater(description[, ...])	An object that can be used for defining state updaters via a simple description (see b

ExplicitStateUpdater class

(*Shortest import*: `from brian2 import ExplicitStateUpdater`)

class `brian2.stateupdaters.explicit.`**`ExplicitStateUpdater`**(*description*, *stochastic=None*, *custom_check=None*)

Bases: *brian2.stateupdaters.base.StateUpdateMethod*

An object that can be used for defining state updaters via a simple description (see below). Resulting instances can be passed to the `method` argument of the *NeuronGroup* constructor. As other state updater functions the *ExplicitStateUpdater* objects are callable, returning abstract code when called with an *Equations*

object.

A description of an explicit state updater consists of a (multi-line) string, containing assignments to variables and a final "x_new = ...", stating the integration result for a single timestep. The assignments can be used to define an arbitrary number of intermediate results and can refer to `f(x, t)` (the function being integrated, as a function of `x`, the previous value of the state variable and `t`, the time) and `dt`, the size of the timestep.

For example, to define a Runge-Kutta 4 integrator (already provided as *rk4*), use:

```
k1 = dt*f(x,t)
k2 = dt*f(x+k1/2,t+dt/2)
k3 = dt*f(x+k2/2,t+dt/2)
k4 = dt*f(x+k3,t+dt)
x_new = x+(k1+2*k2+2*k3+k4)/6
```

Note that for stochastic equations, the function `f` only corresponds to the non-stochastic part of the equation. The additional function `g` corresponds to the stochastic part that has to be multiplied with the stochastic variable xi (a standard normal random variable – if the algorithm needs a random variable with a different variance/mean you have to multiply/add it accordingly). Equations with more than one stochastic variable do not have to be treated differently, the part referring to `g` is repeated for all stochastic variables automatically.

Stochastic integrators can also make reference to `dW` (a normal distributed random number with variance `dt`) and `g(x, t)`, the stochastic part of an equation. A stochastic state updater could therefore use a description like:

```
x_new = x + dt*f(x,t) + g(x, t) * dW
```

For simplicity, the same syntax is used for state updaters that only support additive noise, even though `g(x, t)` does not depend on `x` or `t` in that case.

There a some restrictions on the complexity of the expressions (but most can be worked around by using intermediate results as in the above Runge- Kutta example): Every statement can only contain the functions `f` and `g` once; The expressions have to be linear in the functions, e.g. you can use `dt*f(x, t)` but not `f(x, t)**2`.

Parameters **description** : str

A state updater description (see above).

stochastic : {None, 'additive', 'multiplicative'}

What kind of stochastic equations this state updater supports: `None` means no support of stochastic equations, `'additive'` means only equations with additive noise and `'multiplicative'` means supporting arbitrary stochastic equations.

Raises

`ValueError` If the parsing of the description failed.

See also:

euler, *rk2*, *rk4*, *milstein*

Notes

Since clocks are updated *after* the state update, the time `t` used in the state update step is still at its previous value. Enumerating the states and discrete times, `x_new = x + dt*f(x, t)` is therefore understood as $x_{i+1} = x_i + dtf(x_i, t_i)$, yielding the correct forward Euler integration. If the integrator has to refer to the time at the end of the timestep, simply use `t + dt` instead of `t`.

Attributes

Methods

__call__(eqs[, variables])	Apply a state updater description to model equations.
can_integrate(equations, variables)	
replace_func(x, t, expr, temp_vars, eq_symbols)	Used to replace a single occurance of `f(x, t)` or `g(x, t)`: `expr` is the n

Details

DESCRIPTION = {[Group:({~{"x_new"} W:(abcd...,abcd...) Suppress:("=") Re:('.*')})]... Group:({Suppress:("x_new") Su
A complete state updater description

EXPRESSION = Re:('.*')
A single expression

OUTPUT = Group:({Suppress:("x_new") Suppress:("=") Re:('.*')})
The last line of a state updater description

STATEMENT = Group:({~{"x_new"} W:(abcd...,abcd...) Suppress:("=") Re:('.*')})
An assignment statement

TEMP_VAR = {~{"x_new"} W:(abcd...,abcd...)}
Legal names for temporary variables

DESCRIPTION()
Requires all given C{ParseExpression}s to be found in the given order. Expressions may be separated by whitespace. May be constructed using the C{'+'} operator.

EXPRESSION()
Token for matching strings that match a given regular expression. Defined with string specifying the regular expression in a form recognized by the inbuilt Python re module.

OUTPUT()
Converter to return the matched tokens as a list - useful for returning tokens of C{L{ZeroOrMore}} and C{L{OneOrMore}} expressions.

STATEMENT()
Converter to return the matched tokens as a list - useful for returning tokens of C{L{ZeroOrMore}} and C{L{OneOrMore}} expressions.

TEMP_VAR()
Requires all given C{ParseExpression}s to be found in the given order. Expressions may be separated by whitespace. May be constructed using the C{'+'} operator.

__call__(*eqs*, *variables=None*)
Apply a state updater description to model equations.

Parameters **eqs** : *Equations*

The equations describing the model

variables: dict-like, optional :

The `Variable` objects for the model. Ignored by the explicit state updater.

Examples

```
>>> from brian2 import *
>>> eqs = Equations('dv/dt = -v / tau : volt')
>>> print(euler(eqs))
_v = -dt*v/tau + v
v = _v
>>> print(rk4(eqs))
__k_1_v = -dt*v/tau
__k_2_v = -dt*(0.5*__k_1_v + v)/tau
__k_3_v = -dt*(0.5*__k_2_v + v)/tau
__k_4_v = -dt*(__k_3_v + v)/tau
_v = 0.166666666666667*__k_1_v + 0.333333333333333*__k_2_v + 0.333333333333333*__k_3_v + 0.1
v = _v
```

can_integrate(*equations*, *variables*)

replace_func(*x*, *t*, *expr*, *temp_vars*, *eq_symbols*, *stochastic_variable=None*)

Used to replace a single occurance of `f(x, t)` or `g(x, t)`: `expr` is the non-stochastic (in the case of `f`) or stochastic part (`g`) of the expression defining the right-hand-side of the differential equation describing `var()`. It replaces the variable `var()` with the value given as `x` and `t` by the value given for `t`. Intermediate variables will be replaced with the appropriate replacements as well.

For example, in the *rk2* integrator, the second step involves the calculation of `f(k/2 + x, dt/2 + t)`. If `var()` is `v` and `expr` is `-v / tau`, this will result in `-(_k_v/2 + v)/tau`.

Note that this deals with only one state variable `var()`, given as an argument to the surrounding `_generate_RHS` function.

Functions

diagonal_noise(equations, variables)	Checks whether we deal with diagonal noise, i.e.

diagonal_noise function

(*Shortest import*: `from brian2.stateupdaters.explicit import diagonal_noise`)

brian2.stateupdaters.explicit.**diagonal_noise**(*equations*, *variables*)

Checks whether we deal with diagonal noise, i.e. one independent noise variable per variable.

split_expression(expr)	Split an expression into a part containing the function `f` and another one containing the function `g`.

split_expression function

(*Shortest import*: `from brian2.stateupdaters.explicit import split_expression`)

brian2.stateupdaters.explicit.**split_expression**(*expr*)

Split an expression into a part containing the function `f` and another one containing the function `g`. Returns a tuple of the two expressions (as sympy expressions).

Parameters **expr** : str

An expression containing references to functions `f` and `g`.

Returns **(non_stochastic, stochastic)** : tuple of sympy expressions

A pair of expressions representing the non-stochastic (containing function-independent

terms and terms involving `f`) and the stochastic part of the expression (terms involving `g` and/or `dW`).

Examples

```
>>> split_expression('dt * __f(__x, __t)')
(dt*__f(__x, __t), None)
>>> split_expression('dt * __f(__x, __t) + __dW * __g(__x, __t)')
(dt*__f(__x, __t), __dW*__g(__x, __t))
>>> split_expression('1/(2*dt**.5)*(__g_support - __g(__x, __t))*(__dW**2)')
(0, __dW**2*__g_support*dt**(-0.5)/2 - __dW**2*dt**(-0.5)*__g(__x, __t)/2)
```

Objects

euler	Forward Euler state updater

euler object

(*Shortest import*: `from brian2 import euler`)

brian2.stateupdaters.explicit.**euler = ExplicitStateUpdater(‘‘x_new = __dW*__g(__x, __t) + __x + dt*__f(__x, _**
Forward Euler state updater

milstein	Derivative-free Milstein method

milstein object

(*Shortest import*: `from brian2 import milstein`)

brian2.stateupdaters.explicit.**milstein = ExplicitStateUpdater(‘‘__x_support = __x + dt**0.5*__g(__x, __t) + dt**
Derivative-free Milstein method

rk2	Second order Runge-Kutta method (midpoint method)

rk2 object

(*Shortest import*: `from brian2 import rk2`)

brian2.stateupdaters.explicit.**rk2 = ExplicitStateUpdater(‘‘__k = dt*__f(__x, __t) x_new = __x + dt*__f(0.5*__k +**
Second order Runge-Kutta method (midpoint method)

rk4	Classical Runge-Kutta method (RK4)

rk4 object

(*Shortest import*: `from brian2 import rk4`)

brian2.stateupdaters.explicit.**rk4 = ExplicitStateUpdater(‘‘__k_1 = dt*__f(__x, __t) __k_2 = dt*__f(0.5*__k_1 + _**
Classical Runge-Kutta method (RK4)

exponential_euler module

Exported members: exponential_euler

Classes

ExponentialEulerStateUpdater	A state updater for conditionally linear equations, i.e.

ExponentialEulerStateUpdater class

(*Shortest import*: `from brian2.stateupdaters.exponential_euler import ExponentialEulerStateUpdater`)

class brian2.stateupdaters.exponential_euler.**ExponentialEulerStateUpdater**
Bases: brian2.stateupdaters.base.StateUpdateMethod

A state updater for conditionally linear equations, i.e. equations where each variable only depends linearly on itself (but possibly non-linearly on other variables). Typical Hodgkin-Huxley equations fall into this category, it is therefore the default integration method used in the GENESIS simulator, for example.

Methods

__call__(equations[, variables])	Generate abstract code from equations.
can_integrate(equations, variables)	Return whether the given equations can be integrated using this state updater.

Details

__call__(*equations*, *variables=None*)
Generate abstract code from equations. The method also gets the the variables because some state updaters have to check whether variable names reflect other state variables (which can change from timestep to timestep) or are external values (which stay constant during a run) For convenience, this arguments are optional – this allows to directly see what code a state updater generates for a set of equations by simply writing `euler(eqs)`, for example.

Parameters **equations** : Equations

The model equations.

variables : dict, optional

The `Variable` objects for the model variables.

Returns **code** : str

The abstract code performing a state update step.

can_integrate(*equations*, *variables*)
Return whether the given equations can be integrated using this state updater. This method tests for conditional linearity by executing get_conditionally_linear_system and returns `True` in case this call succeeds.

Functions

get_conditionally_linear_system(eqs)	Convert equations into a linear system using sympy.

get_conditionally_linear_system function

(*Shortest import*: `from brian2.stateupdaters.exponential_euler import get_conditionally_linear_system`)

brian2.stateupdaters.exponential_euler.**get_conditionally_linear_system**(*eqs*)

Convert equations into a linear system using sympy.

Parameters **eqs** : *Equations*

The model equations.

Returns **coefficients** : dict of (sympy expression, sympy expression) tuples

For every variable x, a tuple (M, B) containing the coefficients M and B (as sympy expressions) for M * x + B

Raises

ValueError If one of the equations cannot be converted into a M * x + B form.

Examples

```
>>> from brian2 import Equations
>>> eqs = Equations("""
... dv/dt = (-v + w**2) / tau : 1
... dw/dt = -w / tau : 1
... """)
>>> system = get_conditionally_linear_system(eqs)
>>> print(system['v'])
(-1/tau, w**2.0/tau)
>>> print(system['w'])
(-1/tau, 0)
```

Objects

exponential_euler	A state updater for conditionally linear equations, i.e.

exponential_euler object

(*Shortest import*: `from brian2 import exponential_euler`)

brian2.stateupdaters.exponential_euler.**exponential_euler = <brian2.stateupdaters.exponential_euler.Expo**

A state updater for conditionally linear equations, i.e. equations where each variable only depends linearly on itself (but possibly non-linearly on other variables). Typical Hodgkin-Huxley equations fall into this category, it is therefore the default integration method used in the GENESIS simulator, for example.

6.4.12 synapses package

Package providing synapse support.

spikequeue module

The spike queue class stores future synaptic events produced by a given presynaptic neuron group (or postsynaptic for backward propagation in STDP).

Exported members: SpikeQueue

Classes

SpikeQueue(source_start, source_end[, ...])	Data structure saving the spikes and taking care of delays.

SpikeQueue class

(*Shortest import*: `from brian2.synapses.spikequeue import SpikeQueue`)

class brian2.synapses.spikequeue.**SpikeQueue**(*source_start*, *source_end*, *dtype=<type 'numpy.int32'>*, *precompute_offsets=True*)

Bases: object

Data structure saving the spikes and taking care of delays.

Parameters **synapses** : list of ndarray

A list of synapses (synapses[i]=array of synapse indices for neuron i).

delays : ndarray

An array of delays (delays[k]=delay of synapse k).

dt : *Quantity*

The timestep of the source group

max_delay : *Quantity*, optional

The maximum delay (in second) of synaptic events. At run time, the structure is resized to the maximum delay in `delays`, and thus the `max_delay` should only be specified if delays can change during the simulation (in which case offsets should not be precomputed).

precompute_offsets : bool, optional

A flag to precompute offsets. Defaults to `True`, i.e. offsets (an internal array derived from `delays`, used to insert events in the data structure, see below) are precomputed for all neurons when the object is prepared with the `compress()` method. This usually results in a speed up but takes memory, which is why it can be disabled.

Notes

Data structure

A spike queue is implemented as a 2D array `X` that is circular in the time direction (rows) and dynamic in the events direction (columns). The row index corresponding to the current timestep is `currentime`. Each element contains the target synapse index.

Offsets

Offsets are used to solve the problem of inserting multiple synaptic events with the same delay. This is difficult to vectorise. If there are n synaptic events with the same delay, these events are given an offset between 0 and n-1, corresponding to their relative position in the data structure. They can be either precalculated (faster), or

determined at run time (saves memory). Note that if they are determined at run time, then it is possible to also vectorise over presynaptic spikes.

Attributes

_dt	The dt used for storing the spikes (will be set in *prepare*)
_offsets	precalculated offsets
_precompute_offsets	Whether the offsets should be precomputed
_source_end	The end of the source indices (for subgroups)
_source_start	The start of the source indices (for subgroups)
_stored_spikes	Storage for the store/restore mechanism
currenttime	The current time (in time steps)
n	number of events in each time step

Methods

advance()	Advances by one timestep
peek()	Returns the all the synaptic events corresponding to the current time, as an array of synapse
prepare(delays, dt, synapse_sources)	Prepare the data structure and pre-compute offsets.
push(sources)	Push spikes to the queue.

Details

_dt
The dt used for storing the spikes (will be set in *prepare*)

_offsets
precalculated offsets

_precompute_offsets
Whether the offsets should be precomputed

_source_end
The end of the source indices (for subgroups)

_source_start
The start of the source indices (for subgroups)

_stored_spikes
Storage for the store/restore mechanism

currenttime
The current time (in time steps)

n
number of events in each time step

advance()
Advances by one timestep

peek()
Returns the all the synaptic events corresponding to the current time, as an array of synapse indexes.

prepare(*delays*, *dt*, *synapse_sources*)
Prepare the data structure and pre-compute offsets. This is called every time the network is run. The

size of the of the data structure (number of rows) is adjusted to fit the maximum delay in `delays`, if necessary. Offsets are calculated, unless the option `precompute_offsets` is set to `False`. A flag is set if delays are homogeneous, in which case insertion will use a faster method implemented in `insert_homogeneous`.

push(*sources*)
Push spikes to the queue.

Parameters **sources** : ndarray of int

The indices of the neurons that spiked.

synapses module

Module providing the *Synapses* class and related helper classes/functions.

Exported members: *Synapses*

Classes

StateUpdater(group, method, clock, order)	The *CodeRunner* that updates the state variables of a *Synapses* at every timeste

StateUpdater class

(*Shortest import*: `from brian2.synapses.synapses import StateUpdater`)

class `brian2.synapses.synapses.`**StateUpdater**(*group*, *method*, *clock*, *order*)
Bases: *brian2.groups.group.CodeRunner*

The *CodeRunner* that updates the state variables of a *Synapses* at every timestep.

Methods

update_abstract_code([run_namespace, level])	

Details

update_abstract_code(*run_namespace=None*, *level=0*)

SummedVariableUpdater(expression, ...)	The *CodeRunner* that updates a value in the target group with the sum over values

SummedVariableUpdater class

(*Shortest import*: `from brian2.synapses.synapses import SummedVariableUpdater`)

class `brian2.synapses.synapses.`**SummedVariableUpdater**(*expression*, *target_varname*, *synapses*, *target*)
Bases: *brian2.groups.group.CodeRunner*

The *CodeRunner* that updates a value in the target group with the sum over values in the *Synapses* object.

Synapses(source[, target, model, pre, post, ...])	Class representing synaptic connections.

Synapses class

(*Shortest import*: `from brian2 import Synapses`)

class `brian2.synapses.synapses.`**`Synapses`**(*source*, *target=None*, *model=None*, *pre=None*, *post=None*, *connect=False*, *delay=None*, *on_event='spike'*, *namespace=None*, *dtype=None*, *codeobj_class=None*, *dt=None*, *clock=None*, *order=0*, *method=('linear', 'euler', 'milstein')*, *name='synapses*'*)*

Bases: *brian2.groups.group.Group*

Class representing synaptic connections. Creating a new *Synapses* object does by default not create any synapses – you either have to provide the `connect()` argument or call the *Synapses.connect()* method for that.

Parameters **source** : *SpikeSource*

The source of spikes, e.g. a *NeuronGroup*.

target : *Group*, optional

The target of the spikes, typically a *NeuronGroup*. If none is given, the same as `source()`

model : {`str`, *Equations*}, optional

The model equations for the synapses.

pre : {str, dict}, optional

The code that will be executed after every pre-synaptic spike. Can be either a single (possibly multi-line) string, or a dictionary mapping pathway names to code strings. In the first case, the pathway will be called `pre` and made available as an attribute of the same name. In the latter case, the given names will be used as the pathway/attribute names. Each pathway has its own code and its own delays.

post : {str, dict}, optional

The code that will be executed after every post-synaptic spike. Same conventions as for `pre`, the default name for the pathway is `post`.

connect : {str, bool}. optional

Determines whether any actual synapses are created. `False` (the default) means not to create any synapses, `True` means to create synapses between all source/target pairs. Also accepts a string expression that evaluates to `True` for every synapse that should be created, e.g. `'i == j'` for a one-to-one connectivity. See *Synapses.connect()* for more details.

delay : {*Quantity*, dict}, optional

The delay for the "pre" pathway (same for all synapses) or a dictionary mapping pathway names to delays. If a delay is specified in this way for a pathway, it is stored as a single scalar value. It can still be changed afterwards, but only to a single scalar value. If you want to have delays that vary across synapses, do not use the keyword argument, but instead set the delays via the attribute of the pathway, e.g. `S.pre.delay = ...` (or `S.delay = ...` as an abbreviation), `S.post.delay = ...`, etc.

on_event : str or dict, optional

Define the events which trigger the pre and post pathways. By default, both pathways are triggered by the `'spike'` event, i.e. the event that is triggered by the `threshold` condition in the connected groups.

namespace : dict, optional

A dictionary mapping identifier names to objects. If not given, the namespace will be filled in at the time of the call of *`Network.run()`*, with either the values from the `network` argument of the *`Network.run()`* method or from the local context, if no such argument is given.

dtype : (`dtype`, `dict`), optional

The `numpy.dtype` that will be used to store the values, or a dictionary specifying the type for variable names. If a value is not provided for a variable (or no value is provided at all), the preference setting *core.default_float_dtype* is used.

codeobj_class : class, optional

The *`CodeObject`* class to use to run code.

dt : *`Quantity`*, optional

The time step to be used for the update of the state variables. Cannot be combined with the `clock` argument.

clock : *`Clock`*, optional

The update clock to be used. If neither a clock, nor the `dt` argument is specified, the *`defaultclock`* will be used.

order : int, optional

The priority of of this group for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

method : {str, *`StateUpdateMethod`*}, optional

The numerical integration method to use. If none is given, an appropriate one is automatically determined.

name : str, optional

The name for this object. If none is given, a unique name of the form `synapses`, `synapses_1`, etc. will be automatically chosen.

Attributes

`_pathways`	List of all *`SynapticPathway`* objects
`_registered_variables`	Set of `Variable` objects that should be resized when the
`_synaptic_updaters`	List of names of all updaters, e.g.
`events`	"Events" for all the pathways
`namespace`	The group-specific namespace
`state_updater`	Performs numerical integration step
`summed_updaters`	"Summed variable" mechanism – sum over all synapses of a

Methods

before_run([run_namespace, level])	
connect(pre_or_cond[, post, p, n, ...])	Add synapses.
register_variable(variable)	Register a `DynamicArray` to be automatically resized when the size of the indices chang
unregister_variable(variable)	Unregister a `DynamicArray` from the automatic resizing mechanism.

Details

_pathways
List of all *SynapticPathway* objects

_registered_variables
Set of `Variable` objects that should be resized when the number of synapses changes

_synaptic_updaters
List of names of all updaters, e.g. ['pre', 'post']

events
"Events" for all the pathways

namespace
The group-specific namespace

state_updater
Performs numerical integration step

summed_updaters
"Summed variable" mechanism – sum over all synapses of a pre-/postsynaptic target

before_run(*run_namespace=None*, *level=0*)

connect(*pre_or_cond*, *post=None*, *p=1.0*, *n=1*, *namespace=None*, *level=0*)
Add synapses. The first argument can be either a presynaptic index (int or array) or a condition for synapse creation in the form of a string that evaluates to a boolean value (or directly a boolean value). If it is given as an index, also `post` has to be present. A string condition will be evaluated for all pre-/postsynaptic indices, which can be referred to as `i` and `j`.

Parameters **pre_or_cond** : {int, ndarray of int, bool, str}

The presynaptic neurons (in the form of an index or an array of indices) or a boolean value or a string that evaluates to a boolean value. If it is an index, then also `post` has to be given.

post_neurons : {int, ndarray of int), optional

GroupIndices of neurons from the target group. Non-optional if one or more presynaptic indices have been given.

p : float, optional

The probability to create `n` synapses wherever the condition given as `pre_or_cond` evaluates to true or for the given pre/post indices.

n : int, optional

The number of synapses to create per pre/post connection pair. Defaults to 1.

namespace : dict-like, optional

A namespace that will be used in addition to the group-specific namespaces (if defined). If not specified, the locals and globals around the run function will be used.

level : int, optional

How deep to go up the stack frame to look for the locals/global (see *namespace* argument).

Examples

```
>>> from brian2 import *
>>> import numpy as np
>>> G = NeuronGroup(10, 'dv/dt = -v / tau : 1', threshold='v>1', reset='v=0')
>>> S = Synapses(G, G, 'w:1', pre='v+=w')
>>> S.connect('i != j') # all-to-all but no self-connections
>>> S.connect(0, 0) # connect neuron 0 to itself
>>> S.connect(np.array([1, 2]), np.array([2, 1])) # connect 1->2 and 2->1
>>> S.connect(True) # connect all-to-all
>>> S.connect('i != j', p=0.1)  # Connect neurons with 10% probability, exclude self-connect
>>> S.connect('i == j', n=2)  # Connect all neurons to themselves with 2 synapses
```

register_variable(*variable*)
Register a `DynamicArray` to be automatically resized when the size of the indices change. Called automatically when a `SynapticArrayVariable` specifier is created.

unregister_variable(*variable*)
Unregister a `DynamicArray` from the automatic resizing mechanism.

Tutorials and examples using this

- Tutorial 1-intro-to-brian-neurons
- Tutorial 2-intro-to-brian-synapses
- Example adaptive_threshold
- Example CUBA
- Example COBAHH
- Example compartmental/lfp
- Example compartmental/bipolar_with_inputs2
- Example compartmental/bipolar_with_inputs
- Example standalone/STDP_standalone
- Example standalone/cuba_openmp
- Example synapses/nonlinear
- Example synapses/synapses
- Example synapses/STDP
- Example synapses/spatial_connections
- Example synapses/jeffress
- Example synapses/state_variables
- Example synapses/gapjunctions
- Example synapses/licklider
- Example frompapers/Diesmann_et_al_1999

- Example frompapers/kremer_et_al_2011_barrel_cortex
- Example frompapers/Brunel_Hakim_1999
- Example frompapers/Vogels_et_al_2011
- Example frompapers/Sturzl_et_al_2000
- Example frompapers/Brette_2012/Fig5A

SynapticIndexing(synapses) **Methods**

SynapticIndexing class

(*Shortest import*: `from brian2.synapses.synapses import SynapticIndexing`)

class `brian2.synapses.synapses.`**`SynapticIndexing`**(*synapses*)
Bases: `object`

Methods

__call__([index, index_var])	Returns synaptic indices for `index`, which can be a tuple of indices (including arrays and slices), a

Details

`__call__`(*index=None*, *index_var='_idx'*)
Returns synaptic indices for `index`, which can be a tuple of indices (including arrays and slices), a single index or a string.

SynapticPathway(synapses, code, prepost[, ...])	The *CodeRunner* that applies the pre/post statement(s) to the state variables

SynapticPathway class

(*Shortest import*: `from brian2.synapses.synapses import SynapticPathway`)

class `brian2.synapses.synapses.`**`SynapticPathway`**(*synapses*, *code*, *prepost*, *objname=None*, *delay=None*, *event='spike'*)
Bases: *brian2.groups.group.CodeRunner*, *brian2.groups.group.Group*

The *CodeRunner* that applies the pre/post statement(s) to the state variables of synapses where the pre-/postsynaptic group spiked in this time step.

Parameters **synapses** : *Synapses*

Reference to the main *Synapses* object

prepost : {'pre', 'post'}

Whether this object should react to pre- or postsynaptic spikes

objname : str, optional

The name to use for the object, will be appendend to the name of `synapses` to create a name in the sense of *Nameable*. The `synapses` object should allow access to this object via `synapses.getattr(objname)`. It has to use the actual `objname`

attribute instead of relying on the provided argument, since the name may have changed to become unique. If `None` is provided (the default), `prepost+'*'` will be used (see *Nameable* for an explanation of the wildcard operator).

delay : *Quantity*, optional

A scalar delay (same delay for all synapses) for this pathway. If not given, delays are expected to vary between synapses.

Attributes

_initialise_queue_codeobj	The *CodeObject* initalising the `SpikeQueue` at the begin of a run
queue	The `SpikeQueue`

Methods

initialise_queue()
push_spikes()

Details

_initialise_queue_codeobj
The *CodeObject* initalising the `SpikeQueue` at the begin of a run

queue
The `SpikeQueue`

before_run(**args*, ***kwds*)

initialise_queue()

push_spikes()

update_abstract_code(**args*, ***kwds*)

SynapticSubgroup(synapses, indices)	A simple subgroup of *Synapses* that can be used for indexing.

SynapticSubgroup class

(*Shortest import*: `from brian2.synapses.synapses import SynapticSubgroup`)

class `brian2.synapses.synapses.`**SynapticSubgroup**(*synapses*, *indices*)
Bases: `object`

A simple subgroup of *Synapses* that can be used for indexing.

Parameters **indices** : `ndarray` of int

The synaptic indices represented by this subgroup.

synaptic_pre : `DynamicArrayVariable`

References to all pre-synaptic indices. Only used to throw an error when new synapses where added after creating this object.

Functions

find_synapses(index, synaptic_neuron)	

find_synapses function

(*Shortest import*: `from brian2.synapses.synapses import find_synapses`)

`brian2.synapses.synapses.`**`find_synapses`**(*index*, *synaptic_neuron*)

slice_to_test(x)	Returns a testing function corresponding to whether an index is in slice x.

slice_to_test function

(*Shortest import*: `from brian2.synapses.synapses import slice_to_test`)

`brian2.synapses.synapses.`**`slice_to_test`**(*x*)

Returns a testing function corresponding to whether an index is in slice x. x can also be an int.

6.4.13 units package

The unit system.

Exported members: `pamp`, `namp`, `uamp`, `mamp`, `amp`, `kamp`, `Mamp`, `Gamp`, `Tamp`, `kilogram`, `pmetre`, `nmetre`, `umetre`, `mmetre`, `metre`, `kmetre`, `Mmetre`, `Gmetre`, `Tmetre`, `pmeter`, `nmeter`, `umeter`, `mmeter`, `meter`, `kmeter` ... (174 more members)

`allunits` module

THIS FILE IS AUTOMATICALLY GENERATED BY A STATIC CODE GENERATION TOOL DO NOT EDIT BY HAND

Instead edit the template:

dev/tools/static_codegen/units_template.py

Exported members: `metre`, `meter`, `gram`, `second`, `amp`, `kelvin`, `mole`, `candle`, `gramme`, `kilogram`, `radian`, `steradian`, `hertz`, `newton`, `pascal`, `joule`, `watt`, `coulomb`, `volt`, `farad`, `ohm`, `siemens`, `weber`, `tesla`, `henry` ... (1991 more members)

`fundamentalunits` module

Defines physical units and quantities

Quantity	Unit	Symbol
Length	metre	m
Mass	kilogram	kg
Time	second	s
Electric current	ampere	A
Temperature	kelvin	K
Quantity of substance	mole	mol
Luminosity	candle	cd

Exported members: *DimensionMismatchError*, *get_or_create_dimension()*, *get_dimensions()*, *is_dimensionless()*, *have_same_dimensions()*, *in_unit()*, *in_best_unit()*, *Quantity*, *Unit*, *register_new_unit()*, *check_units()*, *is_scalar_type()*, *get_unit()*, *get_unit_fast()*, unit_checking

Classes

Dimension(dims)	Stores the indices of the 7 basic SI unit dimension (length, mass, etc.).

Dimension class

(*Shortest import*: `from brian2.units.fundamentalunits import Dimension`)

class brian2.units.fundamentalunits.**Dimension**(*dims*)

Bases: object

Stores the indices of the 7 basic SI unit dimension (length, mass, etc.).

Provides a subset of arithmetic operations appropriate to dimensions: multiplication, division and powers, and equality testing.

Parameters **dims** : sequence of float

The dimension indices of the 7 basic SI unit dimensions.

Notes

Users shouldn't use this class directly, it is used internally in Quantity and Unit. Even internally, never use `Dimension(...)` to create a new instance, use *get_or_create_dimension()* instead. This function makes sure that only one Dimension instance exists for every combination of indices, allowing for a very fast dimensionality check with `is`.

Attributes

is_dimensionless	Whether this Dimension is dimensionless.

Methods

get_dimension(d)	Return a specific dimension.

Details

is_dimensionless

Whether this Dimension is dimensionless.

Notes

Normally, instead one should check dimension for being identical to *DIMENSIONLESS*.

get_dimension(*d*)

Return a specific dimension.

Parameters **d** : `str`

A string identifying the SI basic unit dimension. Can be either a description like "length" or a basic unit like "m" or "metre".

Returns **dim** : `float`

The dimensionality of the dimension `d`.

DimensionMismatchError(description, *dims)	Exception class for attempted operations with inconsistent dimensions.

DimensionMismatchError class

(*Shortest import*: `from brian2 import DimensionMismatchError`)

class `brian2.units.fundamentalunits.`**DimensionMismatchError**(*description*, **dims*)
Bases: `exceptions.Exception`

Exception class for attempted operations with inconsistent dimensions.

For example, `3*mvolt + 2*amp` raises this exception. The purpose of this class is to help catch errors based on incorrect units. The exception will print a representation of the dimensions of the two inconsistent objects that were operated on.

Parameters **description** : `str`

A description of the type of operation being performed, e.g. Addition, Multiplication, etc.

dims : `Dimension`

The dimensions of the objects involved in the operation, any number of them is possible

Tutorials and examples using this

- Tutorial 1-intro-to-brian-neurons

Quantity	A number with an associated physical dimension.

Quantity class

(*Shortest import*: `from brian2 import Quantity`)

class `brian2.units.fundamentalunits.`**Quantity**
Bases: `numpy.ndarray`, `object`

A number with an associated physical dimension. In most cases, it is not necessary to create a Quantity object by hand, instead use multiplication and division of numbers with the constant unit names `second`, `kilogram`, etc.

See also:

Unit

Notes

The *Quantity* class defines arithmetic operations which check for consistency of dimensions and raise the DimensionMismatchError exception if they are inconsistent. It also defines default and other representations for

a number for printing purposes.

See the documentation on the Unit class for more details about the available unit names like mvolt, etc.

Casting rules

The rules that define the casting operations for Quantity object are:

1.Quantity op Quantity = Quantity Performs dimension checking if appropriate

2.(Scalar or Array) op Quantity = Quantity Assumes that the scalar or array is dimensionless

There is one exception to the above rule, the number `0` is interpreted as having "any dimension".

Examples

```
>>> from brian2 import *
>>> I = 3 * amp # I is a Quantity object
>>> R = 2 * ohm # same for R
>>> I * R
6. * volt
>>> (I * R).in_unit(mvolt)
'6000. mV'
>>> (I * R) / mvolt
6000.0
>>> X = I + R
Traceback (most recent call last):
    ...
DimensionMismatchError: Addition, dimensions were (A) (m^2 kg s^-3 A^-2)
>>> Is = np.array([1, 2, 3]) * amp
>>> Is * R
array([ 2.,  4.,  6.]) * volt
>>> np.asarray(Is * R) # gets rid of units
array([ 2.,  4.,  6.])
```

Attributes

dimensions	The dimensions of this quantity.
is_dimensionless	Whether this is a dimensionless quantity.
dim	The dimensions of this quantity.

Methods

with_dimensions(value, *args, **keywords)	Create a *Quantity* object with dimensions.
has_same_dimensions(other)	Return whether this object has the same dimensions as another.
in_unit(u[, precision, python_code])	Represent the quantity in a given unit.
in_best_unit([precision, python_code])	Represent the quantity in the "best" unit.

Details

dimensions
The dimensions of this quantity.

is_dimensionless
Whether this is a dimensionless quantity.

dim
The dimensions of this quantity.

static with_dimensions(*value*, **args*, ***keywords*)
Create a *Quantity* object with dimensions.

Parameters **value** : {array_like, number}

The value of the dimension

args : {*Dimension*, sequence of float}

Either a single argument (a *Dimension*) or a sequence of 7 values.

kwds :

Keywords defining the dimensions, see *Dimension* for details.

Returns **q** : *Quantity*

A *Quantity* object with the given dimensions

Examples

All of these define an equivalent *Quantity* object:

```
>>> from brian2 import *
>>> Quantity.with_dimensions(2, get_or_create_dimension(length=1))
2. * metre
>>> Quantity.with_dimensions(2, length=1)
2. * metre
>>> 2 * metre
2. * metre
```

has_same_dimensions(*other*)
Return whether this object has the same dimensions as another.

Parameters **other** : {*Quantity*, array-like, number}

The object to compare the dimensions against.

Returns **same** : `bool`

`True` if `other` has the same dimensions.

in_unit(*u*, *precision=None*, *python_code=False*)
Represent the quantity in a given unit. If `python_code` is `True`, this will return valid python code, i.e. a string like `5.0 * um ** 2` instead of `5.0 um^2`

Parameters **u** : {*Quantity*, *Unit*}

The unit in which to show the quantity.

precision : `int`, optional

The number of digits of precision (in the given unit, see Examples). If no value is given, numpy's `get_printoptions()` value is used.

python_code : `bool`, optional

Whether to return valid python code (`True`) or a human readable string (`False`, the default).

Returns **s** : `str`

String representation of the object in unit `u`.

See also:

in_unit()

Examples

```
>>> from brian2.units import *
>>> from brian2.units.stdunits import *
>>> x = 25.123456 * mV
>>> x.in_unit(volt)
'0.02512346 V'
>>> x.in_unit(volt, 3)
'0.025 V'
>>> x.in_unit(mV, 3)
'25.123 mV'
```

in_best_unit (*precision=None*, *python_code=False*, **regs*)

Represent the quantity in the "best" unit.

Parameters **python_code** : `bool`, optional

If set to `False` (the default), will return a string like `5.0 um^2` which is not a valid Python expression. If set to `True`, it will return `5.0 * um ** 2` instead.

precision : `int`, optional

The number of digits of precision (in the best unit, see Examples). If no value is given, numpy's `get_printoptions()` value is used.

regs : *UnitRegistry* objects

The registries where to search for units. If none are given, the standard, user-defined and additional registries are searched in that order.

Returns **representation** : `str`

A string representation of this *Quantity*.

See also:

in_best_unit()

Examples

```
>>> from brian2.units import *
```

```
>>> x = 0.00123456 * volt
```

```
>>> x.in_best_unit()
'1.23456 mV'
```

```
>>> x.in_best_unit(3)
'1.235 mV'
```

Unit(value[, dim, scale])	A physical unit.

Unit class

(*Shortest import*: `from brian2 import Unit`)

class `brian2.units.fundamentalunits.`**`Unit`**(*value*, *dim=None*, *scale=None*)

Bases: *brian2.units.fundamentalunits.Quantity*

A physical unit.

Normally, you do not need to worry about the implementation of units. They are derived from the *Quantity* object with some additional information (name and string representation).

Basically, a unit is just a number with given dimensions, e.g. mvolt = 0.001 with the dimensions of voltage. The units module defines a large number of standard units, and you can also define your own (see below).

The unit class also keeps track of various things that were used to define it so as to generate a nice string representation of it. See below.

When creating scaled units, you can use the following prefixes:

Factor	Name	Prefix
10^24	yotta	Y
10^21	zetta	Z
10^18	exa	E
10^15	peta	P
10^12	tera	T
10^9	giga	G
10^6	mega	M
10^3	kilo	k
10^2	hecto	h
10^1	deka	da
1		
10^-1	deci	d
10^-2	centi	c
10^-3	milli	m
10^-6	micro	u (mu in SI)
10^-9	nano	n
10^-12	pico	p
10^-15	femto	f
10^-18	atto	a
10^-21	zepto	z
10^-24	yocto	y

Defining your own

It can be useful to define your own units for printing purposes. So for example, to define the newton metre, you write >>> from brian2.units.allunits import metre, newton >>> Nm = newton * metre

You can then do

```
>>> (1*Nm).in_unit(Nm)
'1. N m'
```

which returns `"1 N m"` because the *Unit* class generates a new display name of `"N m"` from the display names `"N"` and `"m"` for newtons and metres automatically.

To register this unit for use in the automatic printing of the *Quantity.in_best_unit()* method, see the documentation for the *UnitRegistry* class.

Construction

The best way to construct a new unit is to use standard units already defined and arithmetic operations, e.g. `newton*metre`. See the documentation for the static methods *Unit.create* and `Unit.create_scaled_units` for more details.

If you don't like the automatically generated display name for the unit, use the `Unit.set_display_name()` method.

Representation

A new unit defined by multiplication, division or taking powers generates a name for the unit automatically, so that for example the name for `pfarad/mmetre**2` is `"pF/mm^2"`, etc. If you don't like the automatically generated name, use the `Unit.set_display_name()` method.

Attributes

dim	The Dimensions of this unit

Methods

create(dim[, name, dispname, latexname, ...])	Create a new named unit.

Details

dim
The Dimensions of this unit

static create (*dim*, *name=''*, *dispname=''*, *latexname=None*, *scalefactor=''*, ***keywords*)
Create a new named unit.

Parameters **dim** : *Dimension*

The dimensions of the unit.

name : `str`, optional

The full name of the unit, e.g. `'volt'`

dispname : `str`, optional

The display name, e.g. `'V'`

latexname : str, optional

The name as a LaTeX expression (math mode is assumed, do not add $ signs or similar), e.g. `'\omega'`. If no `latexname` is specified, `dispname` will be used.

scalefactor : str, optional

The scaling factor, e.g. `'m'` for mvolt

keywords :

The scaling for each SI dimension, e.g. `length="m"`, `mass="-1"`, etc.

Returns **u** : *Unit*

The new unit.

UnitRegistry()	Stores known units for printing in best units.

UnitRegistry class

(*Shortest import*: `from brian2.units.fundamentalunits import UnitRegistry`)

class `brian2.units.fundamentalunits.`**`UnitRegistry`**
Bases: `object`

Stores known units for printing in best units.

All a user needs to do is to use the *register_new_unit()* function.

Default registries:

The units module defines three registries, the standard units, user units, and additional units. Finding best units is done by first checking standard, then user, then additional. New user units are added by using the *register_new_unit()* function.

Standard units includes all the basic non-compound unit names built in to the module, including volt, amp, etc. Additional units defines some compound units like newton metre (Nm) etc.

Methods

add(u)	Add a unit to the registry
__getitem__(x)	Returns the best unit for quantity x

Details

`add`(*u*)
Add a unit to the registry

`__getitem__`(*x*)
Returns the best unit for quantity x

The algorithm is to consider the value:

m=abs(x/u)

for all matching units u. We select the unit where this ratio is the closest to 10 (if it is an array with several values, we select the unit where the deviations from that are the smallest. More precisely, the unit that minimizes the sum of (log10(m)-1)**2 over all entries).

Functions

all_registered_units(*regs)	Generator returning all registered units.

all_registered_units function

(*Shortest import*: `from brian2.units.fundamentalunits import all_registered_units`)

`brian2.units.fundamentalunits.`**`all_registered_units`**(**regs*)
Generator returning all registered units.

Parameters **regs** : any number of *UnitRegistry* objects.

If given, units from the given registries are returned. If none are given, units are returned from the standard units, the user-registered units and the "additional units" (e.g. `newton * metre`) in that order.

Returns **u** : *Unit*

A single unit from the registry.

check_units(**au)	Decorator to check units of arguments passed to a function

check_units function

(*Shortest import*: `from brian2 import check_units`)

brian2.units.fundamentalunits.**check_units**(***au*)

Decorator to check units of arguments passed to a function

Raises

DimensionMismatchError In case the input arguments or the return value do not have the expected dimensions.

Notes

This decorator will destroy the signature of the original function, and replace it with the signature `(*args, **kwds)`. Other decorators will do the same thing, and this decorator critically needs to know the signature of the function it is acting on, so it is important that it is the first decorator to act on a function. It cannot be used in combination with another decorator that also needs to know the signature of the function.

Examples

```
>>> from brian2.units import *
>>> @check_units(I=amp, R=ohm, wibble=metre, result=volt)
... def getvoltage(I, R, **k):
...     return I*R
```

You don't have to check the units of every variable in the function, and you can define what the units should be for variables that aren't explicitly named in the definition of the function. For example, the code above checks that the variable wibble should be a length, so writing

```
>>> getvoltage(1*amp, 1*ohm, wibble=1)
Traceback (most recent call last):
...
DimensionMismatchError: Function "getvoltage" variable "wibble" has wrong dimensions, dimensions
```

fails, but

```
>>> getvoltage(1*amp, 1*ohm, wibble=1*metre)
1. * volt
```

passes. String arguments or `None` are not checked

```
>>> getvoltage(1*amp, 1*ohm, wibble='hello')
1. * volt
```

By using the special name `result`, you can check the return value of the function.

fail_for_dimension_mismatch(obj1[, obj2, ...])	Compare the dimensions of two objects.

fail_for_dimension_mismatch function

(*Shortest import*: `from brian2.units.fundamentalunits import fail_for_dimension_mismatch`)

brian2.units.fundamentalunits.**fail_for_dimension_mismatch**(*obj1*, *obj2=None*, *error_message=None*)

Compare the dimensions of two objects.

Parameters **obj1, obj2** : {array-like, *Quantity*}

The object to compare. If `obj2` is `None`, assume it to be dimensionless

error_message : `str`, optional

An error message that is used in the DimensionMismatchError

Raises

DimensionMismatchError If the dimensions of `obj1` and `obj2` do not match (or, if `obj2` is `None`, in case `obj1` is not dimensionsless).

Notes

Implements special checking for `0`, treating it as having "any dimensions".

get_dimensions(obj)	Return the dimensions of any object that has them.

get_dimensions function

(*Shortest import*: `from brian2 import get_dimensions`)

brian2.units.fundamentalunits.**get_dimensions**(*obj*)

Return the dimensions of any object that has them.

Slightly more general than *Quantity.dimensions* because it will return *DIMENSIONLESS* if the object is of number type but not a *Quantity* (e.g. a `float` or `int`).

Parameters **obj** : `object`

The object to check.

Returns **dim: 'Dimension'** :

The dimensions of the `obj`.

get_or_create_dimension(*args, **kwds)	Create a new Dimension object or get a reference to an existing one.

get_or_create_dimension function

(*Shortest import*: `from brian2 import get_or_create_dimension`)

`brian2.units.fundamentalunits.`**`get_or_create_dimension`**(**args*, ***kwds*)

Create a new Dimension object or get a reference to an existing one. This function takes care of only creating new objects if they were not created before and otherwise returning a reference to an existing object. This allows to compare dimensions very efficiently using `is`.

Parameters **args** : sequence of `float`

A sequence with the indices of the 7 elements of an SI dimension.

kwds : keyword arguments

a sequence of `keyword=value` pairs where the keywords are the names of the SI dimensions, or the standard unit.

Notes

The 7 units are (in order):

Length, Mass, Time, Electric Current, Temperature, Quantity of Substance, Luminosity

and can be referred to either by these names or their SI unit names, e.g. length, metre, and m all refer to the same thing here.

Examples

The following are all definitions of the dimensions of force

```
>>> from brian2 import *
>>> get_or_create_dimension(length=1, mass=1, time=-2)
metre * kilogram * second ** -2
>>> get_or_create_dimension(m=1, kg=1, s=-2)
metre * kilogram * second ** -2
>>> get_or_create_dimension([1, 1, -2, 0, 0, 0, 0])
metre * kilogram * second ** -2
```

get_unit(x, *regs)	Find the most appropriate consistent unit from the unit registries.

get_unit function

(*Shortest import*: `from brian2 import get_unit`)

`brian2.units.fundamentalunits.`**`get_unit`**(*x*, **regs*)

Find the most appropriate consistent unit from the unit registries.

Parameters **x** : {*Quantity*, array-like, number}

The value to find a unit for.

Returns **q** : *Unit*

The equivalent *Unit* for the quantity `x`.

Continued on next page

Table 6.370 – continued from previous page

get_unit_fast(x)	Return a *Quantity* with value 1 and the same dimensions.

get_unit_fast function

(*Shortest import*: `from brian2 import get_unit_fast`)

brian2.units.fundamentalunits.**get_unit_fast**(*x*)
Return a *Quantity* with value 1 and the same dimensions.

have_same_dimensions(obj1, obj2)	Test if two values have the same dimensions.

have_same_dimensions function

(*Shortest import*: `from brian2 import have_same_dimensions`)

brian2.units.fundamentalunits.**have_same_dimensions**(*obj1*, *obj2*)
Test if two values have the same dimensions.

Parameters **obj1, obj2** : {*Quantity*, array-like, number}

The values of which to compare the dimensions.

Returns **same** : `bool`

`True` if `obj1` and `obj2` have the same dimensions.

in_best_unit(x[, precision])	Represent the value in the "best" unit.

in_best_unit function

(*Shortest import*: `from brian2 import in_best_unit`)

brian2.units.fundamentalunits.**in_best_unit**(*x*, *precision=None*)
Represent the value in the "best" unit.

Parameters **x** : {*Quantity*, array-like, number}

The value to display

precision : `int`, optional

The number of digits of precision (in the best unit, see Examples). If no value is given, numpy's `get_printoptions()` value is used.

Returns **representation** : `str`

A string representation of this *Quantity*.

See also:

Quantity.in_best_unit()

Examples

```
>>> from brian2.units import *
```

```
>>> in_best_unit(0.00123456 * volt)
'1.23456 mV'
>>> in_best_unit(0.00123456 * volt, 2)
'1.23 mV'
>>> in_best_unit(0.123456)
'0.123456'
>>> in_best_unit(0.123456, 2)
'0.12'
```

in_unit(x, u[, precision])	Display a value in a certain unit with a given precision.

in_unit function

(*Shortest import*: `from brian2 import in_unit`)

`brian2.units.fundamentalunits.`**`in_unit`**(*x*, *u*, *precision=None*)

Display a value in a certain unit with a given precision.

Parameters **x** : {*Quantity*, array-like, number}

The value to display

u : {*Quantity*, *Unit*}

The unit to display the value `x` in.

precision : `int`, optional

The number of digits of precision (in the given unit, see Examples). If no value is given, numpy's `get_printoptions()` value is used.

Returns **s** : `str`

A string representation of `x` in units of `u`.

See also:

Quantity.in_unit()

Examples

```
>>> from brian2 import *
>>> in_unit(3 * volt, mvolt)
'3000. mV'
>>> in_unit(123123 * msecond, second, 2)
'123.12 s'
>>> in_unit(10 * uA/cm**2, nA/um**2)
'1.00000000e-04 nA/um^2'
>>> in_unit(10 * mV, ohm * amp)
'0.01 ohm A'
>>> in_unit(10 * nS, ohm)
...
Traceback (most recent call last):
    ...
```

```
    DimensionMismatchError: Non-matching unit for method "in_unit",
    dimensions were (m^-2 kg^-1 s^3 A^2) (m^2 kg s^-3 A^-2)
```

is_dimensionless(obj)	Test if a value is dimensionless or not.

is_dimensionless function

(*Shortest import*: `from brian2 import is_dimensionless`)

`brian2.units.fundamentalunits.`**`is_dimensionless`**(*obj*)

Test if a value is dimensionless or not.

Parameters **obj** : `object`

The object to check.

Returns **dimensionless** : `bool`

`True` if `obj` is dimensionless.

is_scalar_type(obj)	Tells you if the object is a 1d number type.

is_scalar_type function

(*Shortest import*: `from brian2 import is_scalar_type`)

`brian2.units.fundamentalunits.`**`is_scalar_type`**(*obj*)

Tells you if the object is a 1d number type.

Parameters **obj** : `object`

The object to check.

Returns **scalar** : `bool`

`True` if `obj` is a scalar that can be interpreted as a dimensionless *Quantity*.

quantity_with_dimensions(floatval, dims)	Create a new *Quantity* with the given dimensions.

quantity_with_dimensions function

(*Shortest import*: `from brian2.units.fundamentalunits import quantity_with_dimensions`)

`brian2.units.fundamentalunits.`**`quantity_with_dimensions`**(*floatval*, *dims*)

Create a new *Quantity* with the given dimensions. Calls `get_or_create_dimensions` with the dimension tuple of the `dims` argument to make sure that unpickling (which calls this function) does not accidentally create new Dimension objects which should instead refer to existing ones.

Parameters **floatval** : `float`

The floating point value of the quantity.

dims : *Dimension*

The dimensions of the quantity.

Returns **q** : *Quantity*

A quantity with the given dimensions.

See also:

`get_or_create_dimensions`

Examples

```
>>> from brian2 import *
>>> quantity_with_dimensions(0.001, volt.dim)
1. * mvolt
```

register_new_unit(u)	Register a new unit for automatic displaying of quantities

register_new_unit function

(*Shortest import*: `from brian2 import register_new_unit`)

`brian2.units.fundamentalunits.`**`register_new_unit`**`(`*`u`*`)`

Register a new unit for automatic displaying of quantities

Parameters **u** : *Unit*

The unit that should be registered.

Examples

```
>>> from brian2 import *
>>> 2.0*farad/metre**2
2. * metre ** -4 * kilogram ** -1 * second ** 4 * amp ** 2
>>> register_new_unit(pfarad / mmetre**2)
>>> 2.0*farad/metre**2
2000000. * pfarad / mmetre ** 2
>>> unregister_unit(pfarad / mmetre**2)
>>> 2.0*farad/metre**2
2. * metre ** -4 * kilogram ** -1 * second ** 4 * amp ** 2
```

unregister_unit(u)	Remove a previously registered unit for automatic displaying of

unregister_unit function

(*Shortest import*: `from brian2.units.fundamentalunits import unregister_unit`)

`brian2.units.fundamentalunits.`**`unregister_unit`**`(`*`u`*`)`

Remove a previously registered unit for automatic displaying of quantities

Parameters **u** : *Unit*

The unit that should be unregistered.

wrap_function_change_dimensions(func, ...)	Returns a new function that wraps the given function `func` so that it chang

wrap_function_change_dimensions function

(*Shortest import*: `from brian2.units.fundamentalunits import wrap_function_change_dimensions`)

brian2.units.fundamentalunits.**wrap_function_change_dimensions**(*func*, *change_dim_func*)

Returns a new function that wraps the given function `func` so that it changes the dimensions of its input. Quantities are transformed to unitless numpy arrays before calling `func`, the output is a quantity with the original dimensions passed through the function `change_dim_func`. A typical use would be a `sqrt` function that uses `lambda d: d ** 0.5` as `change_dim_func`.

These transformations apply only to the very first argument, all other arguments are ignored/untouched.

wrap_function_dimensionless(func)	Returns a new function that wraps the given function `func` so that it raises a Dimen

wrap_function_dimensionless function

(*Shortest import*: `from brian2.units.fundamentalunits import wrap_function_dimensionless`)

brian2.units.fundamentalunits.**wrap_function_dimensionless**(*func*)

Returns a new function that wraps the given function `func` so that it raises a DimensionMismatchError if the function is called on a quantity with dimensions (excluding dimensionless quantitities). Quantities are transformed to unitless numpy arrays before calling `func`.

These checks/transformations apply only to the very first argument, all other arguments are ignored/untouched.

wrap_function_keep_dimensions(func)	Returns a new function that wraps the given function `func` so that it keeps the di

wrap_function_keep_dimensions function

(*Shortest import*: `from brian2.units.fundamentalunits import wrap_function_keep_dimensions`)

brian2.units.fundamentalunits.**wrap_function_keep_dimensions**(*func*)

Returns a new function that wraps the given function `func` so that it keeps the dimensions of its input. Quantities are transformed to unitless numpy arrays before calling `func`, the output is a quantity with the original dimensions re-attached.

These transformations apply only to the very first argument, all other arguments are ignored/untouched, allowing to work functions like `sum` to work as expected with additional `axis` etc. arguments.

wrap_function_remove_dimensions(func)	Returns a new function that wraps the given function `func` so that it removes

wrap_function_remove_dimensions function

(*Shortest import*: `from brian2.units.fundamentalunits import wrap_function_remove_dimensions`)

brian2.units.fundamentalunits.**wrap_function_remove_dimensions**(*func*)

Returns a new function that wraps the given function `func` so that it removes any dimensions from its input. Useful for functions that are returning integers (indices) or booleans, irrespective of the datatype contained in the array.

These transformations apply only to the very first argument, all other arguments are ignored/untouched.

Objects

DIMENSIONLESS	The singleton object for dimensionless Dimensions.

DIMENSIONLESS object

(*Shortest import*: `from brian2.units.fundamentalunits import DIMENSIONLESS`)

`brian2.units.fundamentalunits.`**`DIMENSIONLESS`** **= Dimension()**
The singleton object for dimensionless Dimensions.

additional_unit_register	*UnitRegistry* containing additional units (newton*metre, farad / metre, ...)

additional_unit_register object

(*Shortest import*: `from brian2.units.fundamentalunits import additional_unit_register`)

`brian2.units.fundamentalunits.`**`additional_unit_register`** **= <brian2.units.fundamentalunits.UnitRegistry ob**
UnitRegistry containing additional units (newton*metre, farad / metre, ...)

standard_unit_register	*UnitRegistry* containing all the standard units (metre, kilogram, um2...)

standard_unit_register object

(*Shortest import*: `from brian2.units.fundamentalunits import standard_unit_register`)

`brian2.units.fundamentalunits.`**`standard_unit_register`** **= <brian2.units.fundamentalunits.UnitRegistry objec**
UnitRegistry containing all the standard units (metre, kilogram, um2...)

user_unit_register	*UnitRegistry* containing all units defined by the user

user_unit_register object

(*Shortest import*: `from brian2.units.fundamentalunits import user_unit_register`)

`brian2.units.fundamentalunits.`**`user_unit_register`** **= <brian2.units.fundamentalunits.UnitRegistry object>**
UnitRegistry containing all units defined by the user

`stdunits` module

Optional short unit names

This module defines the following short unit names:

mV, mA, uA (micro_amp), nA, pA, mF, uF, nF, mS, uS, ms, Hz, kHz, MHz, cm, cm2, cm3, mm, mm2, mm3, um, um2, um3

Exported members: `mV`, `mA`, `uA`, `nA`, `pA`, `pF`, `uF`, `nF`, `nS`, `uS`, `ms`, `us`, `Hz`, `kHz`, `MHz`, `cm`, `cm2`, `cm3`, `mm`, `mm2`, `mm3`, `um`, `um2`, `um3`

`unitsafefunctions` module

Unit-aware replacements for numpy functions.

Exported members: *log()*, log10(), *exp()*, *sin()*, *cos()*, *tan()*, *arcsin()*, *arccos()*, *arctan()*, *sinh()*, *cosh()*, *tanh()*, *arcsinh()*, *arccosh()*, *arctanh()*, *diagonal()*, *ravel()*, *trace()*, *dot()*, *where()*, ones_like(), zeros_like()

Functions

arccos(x[, out])	Trigonometric inverse cosine, element-wise.

arccos function

(*Shortest import*: `from brian2 import arccos`)

brian2.units.unitsafefunctions.**arccos**(*x*[, *out*])

Trigonometric inverse cosine, element-wise.

The inverse of *cos()* so that, if `y = cos(x)`, then `x = arccos(y)`.

Parameters **x** : array_like

x-coordinate on the unit circle. For real arguments, the domain is [-1, 1].

out : ndarray, optional

Array of the same shape as `a`, to store results in. See `doc.ufuncs` (Section "Output arguments") for more details.

Returns **angle** : ndarray

The angle of the ray intersecting the unit circle at the given `x`-coordinate in radians [0, pi]. If `x` is a scalar then a scalar is returned, otherwise an array of the same shape as `x` is returned.

See also:

cos(), *arctan()*, *arcsin()*, emath.arccos

Notes

arccos() is a multivalued function: for each `x` there are infinitely many numbers `z` such that `cos(z) = x`. The convention is to return the angle `z` whose real part lies in `[0, pi]`.

For real-valued input data types, *arccos()* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the `invalid` floating point error flag.

For complex-valued input, *arccos()* is a complex analytic function that has branch cuts `[-inf, -1]` and `[1, inf]` and is continuous from above on the former and from below on the latter.

The inverse *cos()* is also known as `acos` or cos^-1.

References

M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 79. http://www.math.sfu.ca/~cbm/aands/

Examples

We expect the arccos of 1 to be 0, and of -1 to be pi:

```
>>> np.arccos([1, -1])
array([ 0.        ,  3.14159265])
```

Plot arccos:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-1, 1, num=100)
>>> plt.plot(x, np.arccos(x))
>>> plt.axis('tight')
>>> plt.show()
```

arccosh(x[, out])	Inverse hyperbolic cosine, element-wise.

arccosh function

(*Shortest import*: `from brian2 import arccosh`)

`brian2.units.unitsafefunctions.`**`arccosh`**(*x*[, *out*])

Inverse hyperbolic cosine, element-wise.

Parameters **x** : array_like

Input array.

out : ndarray, optional

Array of the same shape as `x`, to store results in. See `doc.ufuncs` (Section "Output arguments") for details.

Returns **arccosh** : ndarray

Array of the same shape as `x`.

See also:

cosh(), *arcsinh()*, *sinh()*, *arctanh()*, *tanh()*

Notes

arccosh() is a multivalued function: for each `x` there are infinitely many numbers `z` such that `cosh(z) = x`. The convention is to return the `z` whose imaginary part lies in `[-pi, pi]` and the real part in `[0, inf]`.

For real-valued input data types, *arccosh()* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the `invalid` floating point error flag.

For complex-valued input, *arccosh()* is a complex analytical function that has a branch cut `[-inf, 1]` and is continuous from above on it.

References

[R13], *[R14]*

Examples

```
>>> np.arccosh([np.e, 10.0])
array([ 1.65745445,  2.99322285])
>>> np.arccosh(1)
0.0
```

arcsin(x[, out])	Inverse sine, element-wise.

arcsin function

(*Shortest import*: `from brian2 import arcsin`)

brian2.units.unitsafefunctions.**arcsin**(*x*[, *out*])

Inverse sine, element-wise.

Parameters **x** : array_like

y-coordinate on the unit circle.

out : ndarray, optional

Array of the same shape as x, in which to store the results. See `doc.ufuncs` (Section "Output arguments") for more details.

Returns **angle** : ndarray

The inverse sine of each element in x, in radians and in the closed interval `[-pi/2, pi/2]`. If x is a scalar, a scalar is returned, otherwise an array.

See also:

sin(), *cos()*, *arccos()*, *tan()*, *arctan()*, arctan2, emath.arcsin

Notes

arcsin() is a multivalued function: for each x there are infinitely many numbers z such that $sin(z) = x$. The convention is to return the angle z whose real part lies in [-pi/2, pi/2].

For real-valued input data types, *arcsin* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the `invalid` floating point error flag.

For complex-valued input, *arcsin()* is a complex analytic function that has, by convention, the branch cuts [-inf, -1] and [1, inf] and is continuous from above on the former and from below on the latter.

The inverse sine is also known as `asin` or sin^{-1}.

References

Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 79ff. http://www.math.sfu.ca/~cbm/aands/

Examples

```
>>> np.arcsin(1)     # pi/2
1.5707963267948966
>>> np.arcsin(-1)    # -pi/2
-1.5707963267948966
>>> np.arcsin(0)
0.0
```

arcsinh(x[, out])	Inverse hyperbolic sine element-wise.

arcsinh function

(*Shortest import*: `from brian2 import arcsinh`)

`brian2.units.unitsafefunctions.`**`arcsinh`**(*x*[, *out*])

Inverse hyperbolic sine element-wise.

Parameters **x** : array_like

Input array.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See `doc.ufuncs`.

Returns **out** : ndarray

Array of of the same shape as `x`.

Notes

`arcsinh()` is a multivalued function: for each `x` there are infinitely many numbers `z` such that `sinh(z) = x`. The convention is to return the `z` whose imaginary part lies in `[-pi/2, pi/2]`.

For real-valued input data types, *`arcsinh()`* always returns real output. For each value that cannot be expressed as a real number or infinity, it returns `nan` and sets the `invalid` floating point error flag.

For complex-valued input, *`arccos()`* is a complex analytical function that has branch cuts `[1j, infj]` and `[-1j, -infj]` and is continuous from the right on the former and from the left on the latter.

The inverse hyperbolic sine is also known as `asinh` or `sinh^-1`.

References

[R15], *[R16]*

Examples

```
>>> np.arcsinh(np.array([np.e, 10.0]))
array([ 1.72538256,  2.99822295])
```

arctan(x[, out])	Trigonometric inverse tangent, element-wise.

arctan function

(*Shortest import*: `from brian2 import arctan`)

`brian2.units.unitsafefunctions.`**`arctan`**(*x*[, *out*])

Trigonometric inverse tangent, element-wise.

The inverse of tan, so that if `y = tan(x)` then `x = arctan(y)`.

Parameters **x** : array_like

Input values. *`arctan()`* is applied to each element of `x`.

Returns **out** : ndarray

Out has the same shape as `x`. Its real part is in `[-pi/2, pi/2]` (`arctan(+/-inf)` returns `+/-pi/2`). It is a scalar if `x` is a scalar.

See also:

`arctan2` The "four quadrant" arctan of the angle formed by (`x`, `y`) and the positive `x`-axis.

`angle()` Argument of complex values.

Notes

`arctan()` is a multi-valued function: for each `x` there are infinitely many numbers `z` such that tan(`z`) = `x`. The convention is to return the angle `z` whose real part lies in [-pi/2, pi/2].

For real-valued input data types, *`arctan()`* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the `invalid` floating point error flag.

For complex-valued input, *`arctan()`* is a complex analytic function that has `[1j, infj]` and `[-1j, -infj]` as branch cuts, and is continuous from the left on the former and from the right on the latter.

The inverse tangent is also known as `atan` or tan^{-1}.

References

Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 79. http://www.math.sfu.ca/~cbm/aands/

Examples

We expect the arctan of 0 to be 0, and of 1 to be pi/4:

```
>>> np.arctan([0, 1])
array([ 0.        ,  0.78539816])
```

```
>>> np.pi/4
0.7853981633974483
```

Plot arctan:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-10, 10)
>>> plt.plot(x, np.arctan(x))
```

```
>>> plt.axis('tight')
>>> plt.show()
```

arctanh(x[, out])	Inverse hyperbolic tangent element-wise.

arctanh function

(*Shortest import*: `from brian2 import arctanh`)

brian2.units.unitsafefunctions.**arctanh**(*x*[, *out*])

Inverse hyperbolic tangent element-wise.

Parameters **x** : array_like

Input array.

Returns **out** : ndarray

Array of the same shape as `x`.

See also:

`emath.arctanh`

Notes

arctanh() is a multivalued function: for each `x` there are infinitely many numbers `z` such that `tanh(z) = x`. The convention is to return the `z` whose imaginary part lies in `[-pi/2, pi/2]`.

For real-valued input data types, *arctanh()* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the `invalid` floating point error flag.

For complex-valued input, *arctanh()* is a complex analytical function that has branch cuts `[-1, -inf]` and `[1, inf]` and is continuous from above on the former and from below on the latter.

The inverse hyperbolic tangent is also known as `atanh` or `tanh^-1`.

References

[R17], *[R18]*

Examples

```
>>> np.arctanh([0, -0.5])
array([ 0.        , -0.54930614])
```

cos(x[, out])	Cosine element-wise.

cos function

(*Shortest import*: `from brian2 import cos`)

brian2.units.unitsafefunctions.**cos**(*x*[, *out*])

Cosine element-wise.

Parameters **x** : array_like

Input array in radians.

out : ndarray, optional

Output array of same shape as `x`.

Returns **y** : ndarray

The corresponding cosine values.

Raises

`ValueError: invalid return array shape` if `out` is provided and `out.shape` != `x.shape` (See Examples)

Notes

If `out` is provided, the function writes the result into it, and returns a reference to `out`. (See Examples)

References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972.

Examples

```
>>> np.cos(np.array([0, np.pi/2, np.pi]))
array([  1.00000000e+00,   6.12303177e-17,  -1.00000000e+00])
>>>
>>> # Example of providing the optional output parameter
>>> out2 = np.cos([0.1], out1)
>>> out2 is out1
True
>>>
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.cos(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid return array shape
```

cosh(x[, out])	Hyperbolic cosine, element-wise.

cosh function

(*Shortest import*: `from brian2 import cosh`)

`brian2.units.unitsafefunctions.`**`cosh`**(*x*[, *out*])

Hyperbolic cosine, element-wise.

Equivalent to `1/2 * (np.exp(x) + np.exp(-x))` and `np.cos(1j*x)`.

Parameters **x** : array_like

Input array.

Returns **out** : ndarray

Output array of same shape as `x`.

Examples

```
>>> np.cosh(0)
1.0
```

The hyperbolic cosine describes the shape of a hanging cable:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-4, 4, 1000)
>>> plt.plot(x, np.cosh(x))
>>> plt.show()
```

diagonal(x, *args, **kwds) Return specified diagonals.

diagonal function

(*Shortest import*: `from brian2 import diagonal`)

`brian2.units.unitsafefunctions.`**`diagonal`**(*x*, **args*, ***kwds*)

Return specified diagonals.

If `a` is 2-D, returns the diagonal of `a` with the given offset, i.e., the collection of elements of the form `a[i, i+offset]`. If `a` has more than two dimensions, then the axes specified by `axis1` and `axis2` are used to determine the 2-D sub-array whose diagonal is returned. The shape of the resulting array can be determined by removing `axis1` and `axis2` and appending an index to the right equal to the size of the resulting diagonals.

In versions of NumPy prior to 1.7, this function always returned a new, independent array containing a copy of the values in the diagonal.

In NumPy 1.7, it continues to return a copy of the diagonal, but depending on this fact is deprecated. Writing to the resulting array continues to work as it used to, but a FutureWarning will be issued.

In NumPy 1.9, it will switch to returning a read-only view on the original array. Attempting to write to the resulting array will produce an error.

In NumPy 1.10, it will still return a view, but this view will no longer be marked read-only. Writing to the returned array will alter your original array as well.

If you don't write to the array returned by this function, then you can just ignore all of the above.

If you depend on the current behavior, then we suggest copying the returned array explicitly, i.e., use `np.diagonal(a).copy()` instead of just `np.diagonal(a)`. This will work with both past and future versions of NumPy.

Parameters **a** : array_like

Array from which the diagonals are taken.

offset : int, optional

Offset of the diagonal from the main diagonal. Can be positive or negative. Defaults to main diagonal (0).

axis1 : int, optional

Axis to be used as the first axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to first axis (0).

axis2 : int, optional

Axis to be used as the second axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to second axis (1).

Returns **array_of_diagonals** : ndarray

If `a` is 2-D, a 1-D array containing the diagonal is returned. If the dimension of `a` is larger, then an array of diagonals is returned, "packed" from left-most dimension to right-most (e.g., if `a` is 3-D, then the diagonals are "packed" along rows).

Raises

`ValueError` If the dimension of `a` is less than 2.

See also:

`diag()` MATLAB work-a-like for 1-D and 2-D arrays.

`diagflat()` Create diagonal arrays.

`trace()` Sum along diagonals.

Examples

```
>>> a = np.arange(4).reshape(2,2)
>>> a
array([[0, 1],
       [2, 3]])
>>> a.diagonal()
array([0, 3])
>>> a.diagonal(1)
array([1])
```

A 3-D example:

```
>>> a = np.arange(8).reshape(2,2,2); a
array([[[0, 1],
        [2, 3]],
       [[4, 5],
        [6, 7]]])
>>> a.diagonal(0, # Main diagonals of two arrays created by skipping
...            0, # across the outer(left)-most axis last and
...            1) # the "middle" (row) axis first.
array([[0, 6],
       [1, 7]])
```

The sub-arrays whose main diagonals we just obtained; note that each corresponds to fixing the right-most (column) axis, and that the diagonals are "packed" in rows.

```
>>> a[:,:,0] # main diagonal is [0 6]
array([[0, 2],
       [4, 6]])
>>> a[:,:,1] # main diagonal is [1 7]
```

```
    array([[1, 3],
           [5, 7]])
```

dot(a, b[, out])	Dot product of two arrays.

dot function

(*Shortest import*: `from brian2 import dot`)

brian2.units.unitsafefunctions.**dot**(*a*, *b*, *out=None*)

Dot product of two arrays.

For 2-D arrays it is equivalent to matrix multiplication, and for 1-D arrays to inner product of vectors (without complex conjugation). For N dimensions it is a sum product over the last axis of `a` and the second-to-last of `b`:

```
    dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

Parameters **a** : array_like

First argument.

b : array_like

Second argument.

out : ndarray, optional

Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its dtype must be the dtype that would be returned for `dot(a,b)`. This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

Returns **output** : ndarray

Returns the dot product of `a` and `b`. If `a` and `b` are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned. If `out` is given, then it is returned.

Raises

ValueError If the last dimension of `a` is not the same size as the second-to-last dimension of `b`.

See also:

vdot Complex-conjugating dot product.

tensordot() Sum products over arbitrary axes.

einsum Einstein summation convention.

Examples

```
    >>> np.dot(3, 4)
    12
```

Neither argument is complex-conjugated:

```
>>> np.dot([2j, 3j], [2j, 3j])
(-13+0j)
```

For 2-D arrays it's the matrix product:

```
>>> a = [[1, 0], [0, 1]]
>>> b = [[4, 1], [2, 2]]
>>> np.dot(a, b)
array([[4, 1],
       [2, 2]])
```

```
>>> a = np.arange(3*4*5*6).reshape((3,4,5,6))
>>> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
>>> np.dot(a, b)[2,3,2,1,2,2]
499128
>>> sum(a[2,3,2,:] * b[1,2,:,2])
499128
```

exp(x[, out])	Calculate the exponential of all elements in the input array.

exp function

(*Shortest import*: `from brian2 import exp`)

`brian2.units.unitsafefunctions.`**`exp`**(*x*[, *out*])

Calculate the exponential of all elements in the input array.

Parameters **x** : array_like

Input values.

Returns **out** : ndarray

Output array, element-wise exponential of `x`.

See also:

`expm1` Calculate `exp(x) - 1` for all elements in the array.

`exp2` Calculate `2**x` for all elements in the array.

Notes

The irrational number `e` is also known as Euler's number. It is approximately 2.718281, and is the base of the natural logarithm, `ln` (this means that, if $x = \ln y = \log_e y$, then $e^x = y$. For real input, `exp(x)` is always positive.

For complex arguments, `x = a + ib`, we can write $e^x = e^a e^{ib}$. The first term, e^a, is already known (it is the real argument, described above). The second term, e^{ib}, is $\cos b + i \sin b$, a function with magnitude 1 and a periodic phase.

References

[R19], *[R20]*

Examples

Plot the magnitude and phase of `exp(x)` in the complex plane:

```
>>> import matplotlib.pyplot as plt
```

```
>>> x = np.linspace(-2*np.pi, 2*np.pi, 100)
>>> xx = x + 1j * x[:, np.newaxis] # a + ib over complex plane
>>> out = np.exp(xx)
```

```
>>> plt.subplot(121)
>>> plt.imshow(np.abs(out),
...            extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi])
>>> plt.title('Magnitude of exp(x)')
```

```
>>> plt.subplot(122)
>>> plt.imshow(np.angle(out),
...            extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi])
>>> plt.title('Phase (angle) of exp(x)')
>>> plt.show()
```

log(x[, out])	Natural logarithm, element-wise.

log function

(*Shortest import*: `from brian2 import log`)

brian2.units.unitsafefunctions.**log**(*x*[, *out*])

Natural logarithm, element-wise.

The natural logarithm *log()* is the inverse of the exponential function, so that `log(exp(x)) = x`. The natural logarithm is logarithm in base `e`.

Parameters **x** : array_like

Input value.

Returns **y** : ndarray

The natural logarithm of `x`, element-wise.

See also:

`log10()`, `log2`, `log1p`, `emath.log`

Notes

Logarithm is a multivalued function: for each `x` there is an infinite number of `z` such that `exp(z) = x`. The convention is to return the `z` whose imaginary part lies in `[-pi, pi]`.

For real-valued input data types, *log()* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the `invalid` floating point error flag.

For complex-valued input, *log()* is a complex analytical function that has a branch cut `[-inf, 0]` and is continuous from above on it. *log()* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References

[R21], *[R22]*

Examples

```
>>> np.log([1, np.e, np.e**2, 0])
array([  0.,   1.,   2., -Inf])
```

ravel(x, *args, **kwds) Return a flattened array.

ravel function

(*Shortest import*: `from brian2 import ravel`)

`brian2.units.unitsafefunctions.`**`ravel`**(*x*, **args*, ***kwds*)

Return a flattened array.

A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

Parameters **a** : array_like

Input array. The elements in `a` are read in the order specified by `order`, and packed as a 1-D array.

order : {'C','F', 'A', 'K'}, optional

The elements of `a` are read using this index order. 'C' means to index the elements in C-like order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to index the elements in Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of axis indexing. 'A' means to read the elements in Fortran-like index order if `a` is Fortran *contiguous* in memory, C-like order otherwise. 'K' means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By default, 'C' index order is used.

Returns **1d_array** : ndarray

Output of the same dtype as `a`, and of shape `(a.size,)`.

See also:

`ndarray.flat` 1-D iterator over an array.

`ndarray.flatten` 1-D array copy of the elements of an array in row-major order.

Notes

In C-like (row-major) order, in two dimensions, the row index varies the slowest, and the column index the quickest. This can be generalized to multiple dimensions, where row-major order implies that the index along the first axis varies slowest, and the index along the last quickest. The opposite holds for Fortran-like, or column-major, index ordering.

Examples

It is equivalent to `reshape(-1, order=order)`.

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> print np.ravel(x)
[1 2 3 4 5 6]
```

```
>>> print x.reshape(-1)
[1 2 3 4 5 6]
```

```
>>> print np.ravel(x, order='F')
[1 4 2 5 3 6]
```

When `order` is 'A', it will preserve the array's 'C' or 'F' ordering:

```
>>> print np.ravel(x.T)
[1 4 2 5 3 6]
>>> print np.ravel(x.T, order='A')
[1 2 3 4 5 6]
```

When `order` is 'K', it will preserve orderings that are neither 'C' nor 'F', but won't reverse axes:

```
>>> a = np.arange(3)[::-1]; a
array([2, 1, 0])
>>> a.ravel(order='C')
array([2, 1, 0])
>>> a.ravel(order='K')
array([2, 1, 0])
```

```
>>> a = np.arange(12).reshape(2,3,2).swapaxes(1,2); a
array([[[ 0,  2,  4],
        [ 1,  3,  5]],
       [[ 6,  8, 10],
        [ 7,  9, 11]]])
>>> a.ravel(order='C')
array([ 0,  2,  4,  1,  3,  5,  6,  8, 10,  7,  9, 11])
>>> a.ravel(order='K')
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

setup()	Setup function for doctests (used by nosetest).

setup function

(*Shortest import*: `from brian2.units.unitsafefunctions import setup`)

`brian2.units.unitsafefunctions.`**`setup`**`()`

Setup function for doctests (used by nosetest). We do not want to test this module's docstrings as they are inherited from numpy.

sin(x[, out])	Trigonometric sine, element-wise.

sin function

(*Shortest import*: `from brian2 import sin`)

brian2.units.unitsafefunctions.**sin**(*x*[, *out*])

Trigonometric sine, element-wise.

Parameters **x** : array_like

Angle, in radians (2π rad equals 360 degrees).

Returns **y** : array_like

The sine of each element of x.

See also:

arcsin(), *sinh()*, *cos()*

Notes

The sine is one of the fundamental functions of trigonometry (the mathematical study of triangles). Consider a circle of radius 1 centered on the origin. A ray comes in from the $+x$ axis, makes an angle at the origin (measured counter-clockwise from that axis), and departs from the origin. The y coordinate of the outgoing ray's intersection with the unit circle is the sine of that angle. It ranges from -1 for $x = 3\pi/2$ to +1 for $\pi/2$. The function has zeroes where the angle is a multiple of π. Sines of angles between π and 2π are negative. The numerous properties of the sine and related functions are included in any standard trigonometry text.

Examples

Print sine of one angle:

```
>>> np.sin(np.pi/2.)
1.0
```

Print sines of an array of angles given in degrees:

```
>>> np.sin(np.array((0., 30., 45., 60., 90.)) * np.pi / 180. )
array([ 0.        ,  0.5       ,  0.70710678,  0.8660254 ,  1.        ])
```

Plot the sine function:

```
>>> import matplotlib.pylab as plt
>>> x = np.linspace(-np.pi, np.pi, 201)
>>> plt.plot(x, np.sin(x))
>>> plt.xlabel('Angle [rad]')
>>> plt.ylabel('sin(x)')
>>> plt.axis('tight')
>>> plt.show()
```

sinh(x[, out])	Hyperbolic sine, element-wise.

sinh function

(*Shortest import*: `from brian2 import sinh`)

brian2.units.unitsafefunctions.**sinh**(*x*[, *out*])

Hyperbolic sine, element-wise.

Equivalent to `1/2 * (np.exp(x) - np.exp(-x))` or `-1j * np.sin(1j*x)`.

Parameters **x** : array_like

Input array.

out : ndarray, optional

Output array of same shape as `x`.

Returns **y** : ndarray

The corresponding hyperbolic sine values.

Raises

`ValueError: invalid return array shape` if `out` is provided and `out.shape` != `x.shape` (See Examples)

Notes

If `out` is provided, the function writes the result into it, and returns a reference to `out`. (See Examples)

References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972, pg. 83.

Examples

```
>>> np.sinh(0)
0.0
>>> np.sinh(np.pi*1j/2)
1j
>>> np.sinh(np.pi*1j) # (exact value is 0)
1.2246063538223773e-016j
>>> # Discrepancy due to vagaries of floating point arithmetic.
```

```
>>> # Example of providing the optional output parameter
>>> out2 = np.sinh([0.1], out1)
>>> out2 is out1
True
```

```
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.sinh(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid return array shape
```

tan(x[, out])	Compute tangent element-wise.

tan function

(*Shortest import*: `from brian2 import tan`)

`brian2.units.unitsafefunctions.`**`tan`**(*x*[, *out*])

Compute tangent element-wise.

Equivalent to `np.sin(x)/np.cos(x)` element-wise.

Parameters **x** : array_like

Input array.

out : ndarray, optional

Output array of same shape as `x`.

Returns **y** : ndarray

The corresponding tangent values.

Raises

`ValueError: invalid return array shape` if `out` is provided and `out.shape` != `x.shape` (See Examples)

Notes

If `out` is provided, the function writes the result into it, and returns a reference to `out`. (See Examples)

References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972.

Examples

```
>>> from math import pi
>>> np.tan(np.array([-pi,pi/2,pi]))
array([  1.22460635e-16,   1.63317787e+16,  -1.22460635e-16])
>>>
>>> # Example of providing the optional output parameter illustrating
>>> # that what is returned is a reference to said parameter
>>> out2 = np.cos([0.1], out1)
>>> out2 is out1
True
>>>
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.cos(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid return array shape
```

tanh(x[, out])	Compute hyperbolic tangent element-wise.

tanh function

(*Shortest import*: `from brian2 import tanh`)

`brian2.units.unitsafefunctions.`**`tanh`**(*x*[, *out*])

Compute hyperbolic tangent element-wise.

Equivalent to `np.sinh(x)/np.cosh(x)` or `-1j * np.tan(1j*x)`.

Parameters **x** : array_like

Input array.

out : ndarray, optional

Output array of same shape as `x`.

Returns **y** : ndarray

The corresponding hyperbolic tangent values.

Raises

`ValueError: invalid return array shape` if `out` is provided and `out.shape` != `x.shape` (See Examples)

Notes

If `out` is provided, the function writes the result into it, and returns a reference to `out`. (See Examples)

References

[R23], *[R24]*

Examples

```
>>> np.tanh((0, np.pi*1j, np.pi*1j/2))
array([ 0. +0.00000000e+00j,  0. -1.22460635e-16j,  0. +1.63317787e+16j])
```

```
>>> # Example of providing the optional output parameter illustrating
>>> # that what is returned is a reference to said parameter
>>> out2 = np.tanh([0.1], out1)
>>> out2 is out1
True
```

```
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.tanh(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid return array shape
```

trace(x, *args, **kwds)	Return the sum along diagonals of the array.

trace function

(*Shortest import*: `from brian2 import trace`)

`brian2.units.unitsafefunctions.`**`trace`**(*x*, **args*, ***kwds*)

Return the sum along diagonals of the array.

If a is 2-D, the sum along its diagonal with the given offset is returned, i.e., the sum of elements a[i,i+offset] for all i.

If a has more than two dimensions, then the axes specified by axis1 and axis2 are used to determine the 2-D sub-arrays whose traces are returned. The shape of the resulting array is the same as that of a with axis1 and axis2 removed.

Parameters **a** : array_like

Input array, from which the diagonals are taken.

offset : int, optional

Offset of the diagonal from the main diagonal. Can be both positive and negative. Defaults to 0.

axis1, axis2 : int, optional

Axes to be used as the first and second axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults are the first two axes of a.

dtype : dtype, optional

Determines the data-type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and a is of integer type of precision less than the default integer precision, then the default integer precision is used. Otherwise, the precision is the same as that of a.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output.

Returns **sum_along_diagonals** : ndarray

If a is 2-D, the sum along the diagonal is returned. If a has larger dimensions, then an array of sums along diagonals is returned.

See also:

diag(), *diagonal()*, diagflat()

Examples

```
>>> np.trace(np.eye(3))
3.0
>>> a = np.arange(8).reshape((2,2,2))
>>> np.trace(a)
array([6, 8])
```

```
>>> a = np.arange(24).reshape((2,2,2,3))
>>> np.trace(a).shape
(2, 3)
```

where(condition, [x, y])	Return elements, either from x or y, depending on condition.

where function

(*Shortest import*: from brian2 import where)

brian2.units.unitsafefunctions.**where**(*condition*[, *x*, *y*])

Return elements, either from `x` or `y`, depending on `condition`.

If only `condition` is given, return `condition.nonzero()`.

Parameters **condition** : array_like, bool

When True, yield `x`, otherwise yield `y`.

x, y : array_like, optional

Values from which to choose. `x` and `y` need to have the same shape as `condition`.

Returns **out** : ndarray or tuple of ndarrays

If both `x` and `y` are specified, the output array contains elements of `x` where `condition` is True, and elements from `y` elsewhere.

If only `condition` is given, return the tuple `condition.nonzero()`, the indices where `condition` is True.

See also:

`nonzero()`, `choose()`

Notes

If `x` and `y` are given and input arrays are 1-D, *`where()`* is equivalent to:

```
[xv if c else yv for (c,xv,yv) in zip(condition,x,y)]
```

Examples

```
>>> np.where([[True, False], [True, True]],
...          [[1, 2], [3, 4]],
...          [[9, 8], [7, 6]])
array([[1, 8],
       [3, 4]])
```

```
>>> np.where([[0, 1], [1, 0]])
(array([0, 1]), array([1, 0]))
```

```
>>> x = np.arange(9.).reshape(3, 3)
>>> np.where( x > 5 )
(array([2, 2, 2]), array([0, 1, 2]))
>>> x[np.where( x > 3.0 )]               # Note: result is 1D.
array([ 4.,  5.,  6.,  7.,  8.])
>>> np.where(x < 5, x, -1)               # Note: broadcasting.
array([[ 0.,  1.,  2.],
       [ 3.,  4., -1.],
       [-1., -1., -1.]])
```

Find the indices of elements of `x` that are in `goodvalues`.

```
>>> goodvalues = [3, 4, 7]
>>> ix = np.in1d(x.ravel(), goodvalues).reshape(x.shape)
>>> ix
array([[False, False, False],
       [ True,  True, False],
```

```
        [False,  True, False]], dtype=bool)
>>> np.where(ix)
(array([1, 1, 2]), array([0, 1, 1]))
```

wrap_function_to_method(func)	Wraps a function so that it calls the corresponding method on the Quantities object (if call

wrap_function_to_method function

(*Shortest import*: `from brian2.units.unitsafefunctions import wrap_function_to_method`)

brian2.units.unitsafefunctions.**wrap_function_to_method**(*func*)

Wraps a function so that it calls the corresponding method on the Quantities object (if called with a Quantities object as the first argument). All other arguments are left untouched.

6.4.14 utils package

Utility functions for Brian.

environment module

Utility functions to get information about the environment Brian is running in.

Functions

running_from_ipython()	Check whether we are currently running under ipython.

running_from_ipython function

(*Shortest import*: `from brian2.utils.environment import running_from_ipython`)

brian2.utils.environment.**running_from_ipython**()

Check whether we are currently running under ipython.

Returns **ipython** : bool

Whether running under ipython or not.

filetools module

File system tools

Exported members: *ensure_directory*, *ensure_directory_of_file*, *in_directory*, *copy_directory*

Classes

in_directory(new_dir)	Safely temporarily work in a subdirectory

in_directory class

(*Shortest import*: `from brian2.utils.filetools import in_directory`)

class brian2.utils.filetools.**in_directory**(*new_dir*)
Bases: object

Safely temporarily work in a subdirectory

Usage:

```
with in_directory(directory):
    ... do stuff here
```

Guarantees that the code in the with block will be executed in directory, and that after the block is completed we return to the original directory.

Functions

copy_directory(source, target)	Copies directory source to target.

copy_directory function

(*Shortest import*: `from brian2.utils.filetools import copy_directory`)

brian2.utils.filetools.**copy_directory**(*source*, *target*)
Copies directory source to target.

ensure_directory(d)	Ensures that a given directory exists (creates it if necessary)

ensure_directory function

(*Shortest import*: `from brian2.utils.filetools import ensure_directory`)

brian2.utils.filetools.**ensure_directory**(*d*)
Ensures that a given directory exists (creates it if necessary)

ensure_directory_of_file(f)	Ensures that a directory exists for filename to go in (creates if necessary), and returns the dir

ensure_directory_of_file function

(*Shortest import*: `from brian2.utils.filetools import ensure_directory_of_file`)

brian2.utils.filetools.**ensure_directory_of_file**(*f*)
Ensures that a directory exists for filename to go in (creates if necessary), and returns the directory path.

logger module

Brian's logging module.

Preferences

Logging system preferences `logging.console_log_level = 'WARNING'`

What log level to use for the log written to the console.

Has to be one of CRITICAL, ERROR, WARNING, INFO or DEBUG.

`logging.delete_log_on_exit` = `True`

Whether to delete the log and script file on exit.

If set to `True` (the default), log files (and the copy of the main script) will be deleted after the brian process has exited, unless an uncaught exception occured. If set to `False`, all log files will be kept.

`logging.file_log` = `True`

Whether to log to a file or not.

If set to `True` (the default), logging information will be written to a file. The log level can be set via the *logging.file_log_level* preference.

`logging.file_log_level` = `'DEBUG'`

What log level to use for the log written to the log file.

In case file logging is activated (see *logging.file_log*), which log level should be used for logging. Has to be one of CRITICAL, ERROR, WARNING, INFO or DEBUG.

`logging.save_script` = `True`

Whether to save a copy of the script that is run.

If set to `True` (the default), a copy of the currently run script is saved to a temporary location. It is deleted after a successful run (unless *logging.delete_log_on_exit* is `False`) but is kept after an uncaught exception occured. This can be helpful for debugging, in particular when several simulations are running in parallel.

`logging.std_redirection` = `True`

Whether or not to redirect stdout/stderr to null at certain places.

This silences a lot of annoying compiler output, but will also hide error messages making it harder to debug problems. You can always temporarily switch it off when debugging. In any case, the output is saved to a file and if an error occurs the name of this file will be printed.

Exported members: *get_logger()*, *BrianLogger*, *std_silent*

Classes

BrianLogger(name)	Convenience object for logging.

BrianLogger class

(*Shortest import*: `from brian2 import BrianLogger`)

class `brian2.utils.logger.`**`BrianLogger`**(*name*)

Bases: `object`

Convenience object for logging. Call *get_logger()* to get an instance of this class.

Parameters **name** : str

The name used for logging, normally the name of the module.

Methods

debug(msg[, name_suffix, once])	Log a debug message.

Continued on next page

Table 6.414 – continued from previous page

error(msg[, name_suffix, once])	Log an error message.
info(msg[, name_suffix, once])	Log an info message.
log_level_debug()	Set the log level to "debug".
log_level_error()	Set the log level to "error".
log_level_info()	Set the log level to "info".
log_level_warn()	Set the log level to "warn".
suppress_hierarchy(name[, filter_log_file])	Suppress all log messages in a given hiearchy.
suppress_name(name[, filter_log_file])	Suppress all log messages with a given name.
warn(msg[, name_suffix, once])	Log a warn message.

Details

debug(*msg*, *name_suffix=None*, *once=False*)
Log a debug message.

Parameters **msg** : str

The message to log.

name_suffix : str, optional

A suffix to add to the name, e.g. a class or function name.

once : bool, optional

Whether this message should be logged only once and not repeated if sent another time.

error(*msg*, *name_suffix=None*, *once=False*)
Log an error message.

Parameters **msg** : str

The message to log.

name_suffix : str, optional

A suffix to add to the name, e.g. a class or function name.

once : bool, optional

Whether this message should be logged only once and not repeated if sent another time.

info(*msg*, *name_suffix=None*, *once=False*)
Log an info message.

Parameters **msg** : str

The message to log.

name_suffix : str, optional

A suffix to add to the name, e.g. a class or function name.

once : bool, optional

Whether this message should be logged only once and not repeated if sent another time.

static log_level_debug()
Set the log level to "debug".

static log_level_error()
Set the log level to "error".

static log_level_info()
Set the log level to "info".

static log_level_warn()
Set the log level to "warn".

static suppress_hierarchy(*name*, *filter_log_file=False*)
Suppress all log messages in a given hiearchy.

Parameters **name** : str

Suppress all log messages in the given `name` hierarchy. For example, specifying `'brian2'` suppresses all messages logged by Brian, specifying `'brian2.codegen'` suppresses all messages generated by the code generation modules.

filter_log_file : bool, optional

Whether to suppress the messages also in the log file. Defaults to `False` meaning that suppressed messages are not displayed on the console but are still saved to the log file.

static suppress_name(*name*, *filter_log_file=False*)
Suppress all log messages with a given name.

Parameters **name** : str

Suppress all log messages ending in the given `name`. For example, specifying `'resolution_conflict'` would suppress messages with names such as `brian2.equations.codestrings.CodeString.resolution_conflict` or `brian2.equations.equations.Equations.resolution_conflict`.

filter_log_file : bool, optional

Whether to suppress the messages also in the log file. Defaults to `False` meaning that suppressed messages are not displayed on the console but are still saved to the log file.

warn(*msg*, *name_suffix=None*, *once=False*)
Log a warn message.

Parameters **msg** : str

The message to log.

name_suffix : str, optional

A suffix to add to the name, e.g. a class or function name.

once : bool, optional

Whether this message should be logged only once and not repeated if sent another time.

Tutorials and examples using this

- Example frompapers/Rossant_et_al_2011bis

HierarchyFilter(name)	A class for suppressing all log messages in a subtree of the name hierarchy.

HierarchyFilter class

(*Shortest import*: `from brian2.utils.logger import HierarchyFilter`)

class `brian2.utils.logger.`**HierarchyFilter**(*name*)

Bases: object

A class for suppressing all log messages in a subtree of the name hierarchy. Does exactly the opposite as the logging.Filter class, which allows messages in a certain name hierarchy to *pass*.

Parameters **name** : str

The name hiearchy to suppress. See *BrianLogger.suppress_hierarchy* for details.

Methods

filter(record)	Filter out all messages in a subtree of the name hierarchy.

Details

filter(*record*)
Filter out all messages in a subtree of the name hierarchy.

LogCapture(log_list[, log_level])	A class for capturing log warnings.

LogCapture class

(*Shortest import*: `from brian2.utils.logger import LogCapture`)

class brian2.utils.logger.**LogCapture**(*log_list*, *log_level=30*)
Bases: logging.Handler

A class for capturing log warnings. This class is used by *catch_logs* to allow testing in a similar way as with warnings.catch_warnings.

Methods

emit(record)	
install()	Install this handler to catch all warnings.
uninstall()	Uninstall this handler and re-connect the previously installed handlers.

Details

emit(*record*)

install()
Install this handler to catch all warnings. Temporarily disconnect all other handlers.

uninstall()
Uninstall this handler and re-connect the previously installed handlers.

NameFilter(name)	A class for suppressing log messages ending with a certain name.

NameFilter class

(*Shortest import*: `from brian2.utils.logger import NameFilter`)

class `brian2.utils.logger.`**`NameFilter`**(*name*)
Bases: `object`

A class for suppressing log messages ending with a certain name.

Parameters **name** : str

The name to suppress. See *`BrianLogger.suppress_name`* for details.

Methods

`filter`(record)	Filter out all messages ending with a certain name.

Details

`filter`(*record*)
Filter out all messages ending with a certain name.

`catch_logs`([log_level])	A context manager for catching log messages.

catch_logs class

(*Shortest import*: `from brian2.utils.logger import catch_logs`)

class `brian2.utils.logger.`**`catch_logs`**(*log_level=30*)
Bases: `object`

A context manager for catching log messages. Use this for testing the messages that are logged. Defaults to catching warning/error messages and this is probably the only real use case for testing. Note that while this context manager is active, *all* log messages are suppressed. Using this context manager returns a list of (log level, name, message) tuples.

Parameters **log_level** : int or str, optional

The log level above which messages are caught.

Examples

```
>>> logger = get_logger('brian2.logtest')
>>> logger.warn('An uncaught warning')
WARNING  brian2.logtest: An uncaught warning
>>> with catch_logs() as l:
...     logger.warn('a caught warning')
...     print('l contains: %s' % l)
...
l contains: [('WARNING', 'brian2.logtest', 'a caught warning')]
```

`std_silent`([alwaysprint])	Context manager that temporarily silences stdout and stderr but keeps the output saved in a temporary

std_silent class

(*Shortest import*: `from brian2 import std_silent`)

class `brian2.utils.logger.`**`std_silent`**(*alwaysprint=False*)
Bases: `object`

Context manager that temporarily silences stdout and stderr but keeps the output saved in a temporary file and writes it if an exception is raised.

Attributes

dest_stderr
dest_stdout

Methods

close()

Details

`dest_stderr` = None

`dest_stdout` = None

classmethod `close`()

Functions

brian_excepthook(exc_type, exc_obj, exc_tb)	Display a message mentioning the debug log in case of an uncaught exception.

brian_excepthook function

(*Shortest import*: `from brian2.utils.logger import brian_excepthook`)

`brian2.utils.logger.`**`brian_excepthook`**(*exc_type*, *exc_obj*, *exc_tb*)
Display a message mentioning the debug log in case of an uncaught exception.

clean_up_logging()	Shutdown the logging system and delete the debug log file if no error occured.

clean_up_logging function

(*Shortest import*: `from brian2.utils.logger import clean_up_logging`)

`brian2.utils.logger.`**`clean_up_logging`**()
Shutdown the logging system and delete the debug log file if no error occured.

get_logger([module_name])	Get an object that can be used for logging.

get_logger function

(*Shortest import*: `from brian2 import get_logger`)

`brian2.utils.logger.`**`get_logger`**(*module_name='brian2'*)

Get an object that can be used for logging.

Parameters **module_name** : str

The name used for logging, should normally be the module name as returned by `__name__`.

Returns **logger** : *BrianLogger*

---

log_level_validator(log_level)

log_level_validator function

(*Shortest import*: `from brian2.utils.logger import log_level_validator`)

`brian2.utils.logger.`**`log_level_validator`**(*log_level*)

`stringtools` module

A collection of tools for string formatting tasks.

Exported members: *indent*, *deindent*, *word_substitute*, *replace*, *get_identifiers*, *strip_empty_lines*, *stripped_deindented_lines*, *strip_empty_leading_and_trailing_lines*, *code_representation*, *SpellChecker*

Classes

SpellChecker(words[, alphabet])	A simple spell checker that will be used to suggest the correct name if the user made a typo (e

SpellChecker class

(*Shortest import*: `from brian2.utils.stringtools import SpellChecker`)

class `brian2.utils.stringtools.`**`SpellChecker`**(*words*, *alphabet='abcdefghijklmnopqrstuvwxyz0123456789_'*)

Bases: `object`

A simple spell checker that will be used to suggest the correct name if the user made a typo (e.g. for state variable names).

Parameters **words** : iterable of str

The known words

alphabet : iterable of str, optional

The allowed characters. Defaults to the characters allowed for identifiers, i.e. ascii characters, digits and the underscore.

Methods

edits1(word)	
known(words)	
known_edits2(word)	
suggest(word)	

Details

edits1 (*word*)

known (*words*)

known_edits2 (*word*)

suggest (*word*)

Functions

code_representation(code)	Returns a string representation for several different formats of code

code_representation function

(*Shortest import*: `from brian2.utils.stringtools import code_representation`)

brian2.utils.stringtools.**code_representation**(*code*)

Returns a string representation for several different formats of code

Formats covered include: - A single string - A list of statements/strings - A dict of strings - A dict of lists of statements/strings

deindent(text[, numtabs, spacespertab, ...])	Returns a copy of the string with the common indentation removed.

deindent function

(*Shortest import*: `from brian2.utils.stringtools import deindent`)

brian2.utils.stringtools.**deindent**(*text*, *numtabs=None*, *spacespertab=4*, *docstring=False*)

Returns a copy of the string with the common indentation removed.

Note that all tab characters are replaced with `spacespertab` spaces.

If the `docstring` flag is set, the first line is treated differently and is assumed to be already correctly tabulated.

If the `numtabs` option is given, the amount of indentation to remove is given explicitly and not the common indentation.

Examples

Normal strings, e.g. function definitions:

```
>>> multiline = """    def f(x):
...           return x**2"""
>>> print(multiline)
    def f(x):
         return x**2
>>> print(deindent(multiline))
```

```
    def f(x):
         return x**2
    >>> print(deindent(multiline, docstring=True))
        def f(x):
    return x**2
    >>> print(deindent(multiline, numtabs=1, spacespertab=2))
      def f(x):
           return x**2
```

Docstrings:

```
    >>> docstring = """First docstring line.
    ...     This line determines the indentation."""
    >>> print(docstring)
    First docstring line.
        This line determines the indentation.
    >>> print(deindent(docstring, docstring=True))
    First docstring line.
    This line determines the indentation.
```

get_identifiers(expr[, include_numbers])	Return all the identifiers in a given string `expr`, that is everything that matches a p

get_identifiers function

(*Shortest import*: `from brian2.utils.stringtools import get_identifiers`)

`brian2.utils.stringtools.`**`get_identifiers`**(*expr*, *include_numbers=False*)

Return all the identifiers in a given string `expr`, that is everything that matches a programming language variable like expression, which is here implemented as the regexp `\b[A-Za-z_][A-Za-z0-9_]*\b`.

Parameters **expr** : str

The string to analyze

include_numbers : bool, optional

Whether to include number literals in the output. Defaults to `False`.

Returns **identifiers** : set

A set of all the identifiers (and, optionally, numbers) in `expr`.

Examples

```
    >>> expr = '3-a*_b+c5+8+f(A - .3e-10, tau_2)*17'
    >>> ids = get_identifiers(expr)
    >>> print(sorted(list(ids)))
    ['A', '_b', 'a', 'c5', 'f', 'tau_2']
    >>> ids = get_identifiers(expr, include_numbers=True)
    >>> print(sorted(list(ids)))
    ['.3e-10', '17', '3', '8', 'A', '_b', 'a', 'c5', 'f', 'tau_2']
```

indent(text[, numtabs, spacespertab, tab])	Indents a given multiline string.

indent function

(*Shortest import*: `from brian2.utils.stringtools import indent`)

`brian2.utils.stringtools.`**`indent`**(*text*, *numtabs=1*, *spacespertab=4*, *tab=None*)

Indents a given multiline string.

By default, indentation is done using spaces rather than tab characters. To use tab characters, specify the tab character explictly, e.g.:

```
indent(text, tab='\t')
```

Note that in this case `spacespertab` is ignored.

Examples

```
>>> multiline = """def f(x):
...     return x*x"""
>>> print(multiline)
def f(x):
    return x*x
>>> print(indent(multiline))
    def f(x):
        return x*x
>>> print(indent(multiline, numtabs=2))
        def f(x):
            return x*x
>>> print(indent(multiline, spacespertab=2))
  def f(x):
      return x*x
>>> print(indent(multiline, tab='####'))
####def f(x):
####    return x*x
```

replace(s, substitutions)	Applies a dictionary of substitutions.

replace function

(*Shortest import*: `from brian2.utils.stringtools import replace`)

`brian2.utils.stringtools.`**`replace`**(*s*, *substitutions*)

Applies a dictionary of substitutions. Simpler than *word_substitute*, it does not attempt to only replace words

strip_empty_leading_and_trailing_lines(s)	Removes all empty leading and trailing lines in the multi-line string `s`.

strip_empty_leading_and_trailing_lines function

(*Shortest import*: `from brian2.utils.stringtools import strip_empty_leading_and_trailing_lines`)

`brian2.utils.stringtools.`**`strip_empty_leading_and_trailing_lines`**(*s*)

Removes all empty leading and trailing lines in the multi-line string `s`.

strip_empty_lines(s)	Removes all empty lines from the multi-line string s.

strip_empty_lines function

(*Shortest import*: `from brian2.utils.stringtools import strip_empty_lines`)

brian2.utils.stringtools.**strip_empty_lines**(*s*)
Removes all empty lines from the multi-line string s.

Examples

```
>>> multiline = """A string with
...
... an empty line."""
>>> print(strip_empty_lines(multiline))
A string with
an empty line.
```

stripped_deindented_lines(code)	Returns a list of the lines in a multi-line string, deindented.

stripped_deindented_lines function

(*Shortest import*: `from brian2.utils.stringtools import stripped_deindented_lines`)

brian2.utils.stringtools.**stripped_deindented_lines**(*code*)
Returns a list of the lines in a multi-line string, deindented.

word_substitute(expr, substitutions)	Applies a dict of word substitutions.

word_substitute function

(*Shortest import*: `from brian2.utils.stringtools import word_substitute`)

brian2.utils.stringtools.**word_substitute**(*expr*, *substitutions*)
Applies a dict of word substitutions.

The dict `substitutions` consists of pairs `(word, rep)` where each word `word` appearing in `expr` is replaced by `rep`. Here a 'word' means anything matching the regexp `\bword\b`.

Examples

```
>>> expr = 'a*_b+c5+8+f(A)'
>>> print(word_substitute(expr, {'a':'banana', 'f':'func'}))
banana*_b+c5+8+func(A)
```

topsort module

Exported members: *topsort*

Functions

topsort(graph)	Topologically sort a graph

topsort function

(*Shortest import*: `from brian2.utils.topsort import topsort`)

`brian2.utils.topsort.`**`topsort`**(*graph*)

Topologically sort a graph

The graph should be of the form `{node: [list of nodes], ...}`.

CHAPTER 7

Developer's guide

This section is intended as a guide to how Brian functions internally for people developing Brian itself, or extensions to Brian. It may also be of some interest to others wishing to better understand how Brian works internally.

7.1 Coding guidelines

The basic principles of developing Brian are:

1. For the user, the emphasis is on making the package flexible, readable and easy to use. See the paper "The Brian simulator" in Frontiers in Neuroscience for more details.
2. For the developer, the emphasis is on keeping the package maintainable by a small number of people. To this end, we use stable, well maintained, existing open source packages whenever possible, rather than writing our own code.

7.1.1 Development workflow

Brian development is done in a git repository on github. Continuous integration testing is provided by travis CI, code coverage is measured with coveralls.io.

The repository structure

Brian's repository structure is very simple, as we are normally not supporting older versions with bugfixes or other complicated things. The *master* branch of the repository is the basis for releases, a release is nothing more than adding a tag to the branch, creating the tarball, etc. The *master* branch should always be in a deployable state, i.e. one should be able to use it as the base for everyday work without worrying about random breakages due to updates. To ensure this, no commit ever goes into the *master* branch without passing the test suite before (see below). The only exception to this rule is if a commit not touches any code files, e.g. additions to the README file or to the documentation (but even in this case, care should be taken that the documentation is still built correctly).

For every feature that a developer works on, a new branch should be opened (normally based on the *master* branch), with a descriptive name (e.g. `add-numba-support`). For developers that are members of "brian-team", the branch should ideally be created in the main repository. This way, one can easily get an overview over what the "core team" is currently working on. Developers who are not members of the team should fork the repository and work in their own repository (if working on multiple issues/features, also using branches).

Implementing a feature/fixing a bug

Every new feature or bug fix should be done in a dedicated branch and have an issue in the issue database. For bugs, it is important to not only fix the bug but also to introduce a new test case (see Testing) that makes sure that the bug will not ever be reintroduced by other changes. It is often a good idea to first define the test cases (that should fail) and then work on the fix so that the tests pass. As soon as the feature/fix is complete *or* as soon as specific feedback on the code is needed, open a "pull request" to merge the changes from your branch into *master*. In this pull request, others can comment on the code and make suggestions for improvements. New commits to the respective branch automatically appear in the pull request which makes it a great tool for iterative code review. Even more useful, travis will automatically run the test suite on the result of the merge. As a reviewer, always wait for the result of this test (it can take up to 30 minutes or so until it appears) before doing the merge and never merge when a test fails. As soon as the reviewer (someone from the core team and not the author of the feature/fix) decides that the branch is ready to merge, he/she can merge the pull request and optionally delete the corresponding branch (but it will be hidden by default, anyway).

Useful links

- The Brian repository: https://github.com/brian-team/brian2
- Travis testing for Brian: https://travis-ci.org/brian-team/brian2
- Code Coverage for Brian: https://coveralls.io/r/brian-team/brian2
- The Pro Git book: http://git-scm.com/book
- github's documentation on pull requests: https://help.github.com/articles/using-pull-requests

7.1.2 Coding conventions

General recommendations

Syntax is chosen as much as possible from the user point of view, to reflect the concepts as directly as possible. Ideally, a Brian script should be readable by someone who doesn't know Python or Brian, although this isn't always possible. Function, class and keyword argument names should be explicit rather than abbreviated and consistent across Brian. See Romain's paper On the design of script languages for neural simulators for a discussion.

We use the PEP-8 coding conventions for our code. This in particular includes the following conventions:

- Use 4 spaces instead of tabs per indentation level
- Use spaces after commas and around the following binary operators: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).
- Do *not* use spaces around the equals sign in keyword arguments or when specifying default values. Neither put spaces immediately inside parentheses, brackets or braces, immediately before the open parenthesis that starts the argument list of a function call, or immediately before the open parenthesis that starts an indexing or slicing.
- Avoid using a backslash for continuing lines whenever possible, instead use Python's implicit line joining inside parentheses, brackets and braces.
- The core code should only contain ASCII characters, no encoding has to be declared
- imports should be on different lines (e.g. do not use `import sys, os`) and should be grouped in the following order, using blank lines between each group:
  1. standard library imports
  2. third-party library imports (e.g. numpy, scipy, sympy, ...)

3. brian imports

- Use absolute imports for everything outside of "your" package, e.g. if you are working in *brian2.equations*, import functions from the stringtools modules via `from brian2.utils.stringtools import ...`. Use the full path when importing, e.g. do `from brian2.units.fundamentalunits import seconds` instead of `from brian2 import seconds`.
- Use "new-style" relative imports for everything in "your" package, e.g. in `brian2.codegen.functions.py` import the *Function* class as `from .specifiers import Function`.
- Do not use wildcard imports (`from brian2 import *`), instead import only the identifiers you need, e.g. `from brian2 import NeuronGroup, Synapses`. For packages like numpy that are used a lot, use `import numpy as np`. But note that the user should still be able to do something like `from brian2 import *` (and this style can also be freely used in examples and tests, for example). Modules always have to use the `__all__` mechanism to specify what is being made available with a wildcard import. As an exception from this rule, the main `brian2/__init__.py` may use wildcard imports.

Python 2 vs. Python 3

Brian is written in Python 2 but runs on Python 3 using the 2to3 conversion tool (which is automatically applied if Brian is installed using the standard `python setup.py install` mechanism). To make this possible without too much effort, Brian no longer supports Python 2.5 and can therefore make use of a couple of forward-compatible (but backward-incompatible) idioms introduced in Python 2.6. The Porting to Python 3 book is available online and has a lot of information on these topics. Here are some things to keep in mind when developing Brian:

- If you are working with integers and using division, consider using `//` for flooring division (default behaviour for `/` in python 2) and switch the behaviour of `/` to floating point division by using `from __future__ import division`.
- If importing modules from the standard library (which have changed quite a bit from Python 2 to Python 3), only use simple import statements like `import itertools` instead of `from itertools import izip` – *2to3* is otherwise unable to make the correct conversion.
- If you are using the `print` statement (which should only occur in tests, in particular doctests – always use the Logging framework if you want to present messages to the user otherwise), try "cheating" and use the functional style in Python 2, i.e. write `print('some text')` instead of `print 'some text'`. More complicated print statements should be avoided, e.g instead of `print >>sys.stderr, 'Error message` use `sys.stderr.write('Error message\n')` (or, again, use logging).
- Exception stacktraces look a bit different in Python 2 and 3: For non-standard exceptions, Python 2 only prints the Exception class name (e.g. `DimensionMismatchError`) whereas Python 3 prints the name including the module name (e.g. `brian2.units.fundamentalunits.DimensionMismatchError`). This will make doctests fail that match the exception message. In this case, write the doctest in the style of Python 2 but add the doctest directive `#doctest: +IGNORE_EXCEPTION_DETAIL` to the statement leading to the exception. This unfortunately has the side effect of also ignoring the text of the exception, but it will still fail for an incorrect exception type.
- If you write code reading and writing strings to files, make sure you make the distinction between bytes and unicode (see "separate binary data and strings") In general, strings within Brian are unicode strings and only converted to bytes when reading from or writing to a file (or something like a network stream, for example).
- If you are sorting lists or dictionaries, have a look at "when sorting, use key instead of cmp"
- Make sure to define a `__hash__` function for objects that define an `__eq__` function (and to define it consistently). Python 3 is more strict about this, an object with `__eq__` but without `__hash__` is unhashable.

7.1.3 Representing Brian objects

`__repr__` and `__str__`

Every class should specify or inherit useful `__repr__` and `__str__` methods. The `__repr__` method should give the "official" representation of the object; if possible, this should be a valid Python expression, ideally allowing for `eval(repr(x)) == x`. The `__str__` method on the other hand, gives an "informal" representation of the object. This can be anything that is helpful but does not have to be Python code. For example:

```
>>> import numpy as np
>>> ar = np.array([1, 2, 3]) * mV
>>> print(ar)  # uses __str__
[ 1.  2.  3.] mV
>>> ar  # uses __repr__
array([ 1.,  2.,  3.]) * mvolt
```

If the representation returned by `__repr__` is not Python code, it should be enclosed in `<...>`, e.g. a *Synapses* representation might be `<Synapses object with 64 synapses>`.

If you don't want to make the distinction between `__repr__` and `__str__`, simply define only a `__repr__` function, it will be used instead of `__str__` automatically (no need to write `__str__ = __repr__`). Finally, if you include the class name in the representation (which you should in most cases), use `self.__class__.__name__` instead of spelling out the name explicitly – this way it will automatically work correctly for subclasses. It will also prevent you from forgetting to update the class name in the representation if you decide to rename the class.

LaTeX representations with sympy

Brian objects dealing with mathematical expressions and equations often internally use sympy. Sympy's `latex` function does a nice job of converting expressions into LaTeX code, using fractions, root symbols, etc. as well as converting greek variable names into corresponding symbols and handling sub- and superscripts. For the conversion of variable names to work, they should use an underscore for subscripts and two underscores for superscripts:

```
>>> from sympy import latex, Symbol
>>> tau_1__e = Symbol('tau_1__e')
>>> print latex(tau_1__e)
\tau^{e}_{1}
```

Sympy's printer supports formatting arbitrary objects, all they have to do is to implement a `_latex` method (no trailing underscore). For most Brian objects, this is unnecessary as they will never be formatted with sympy's LaTeX printer. For some core objects, in particular the units, is is useful, however, as it can be reused in LaTeX representations for ipython (see below). Note that the `_latex` method should not return `$` or `\begin{equation}` (sympy's method includes a `mode` argument that wraps the output automatically).

Representations for ipython

"Old" ipython console

In particular for representations involing arrays or lists, it can be useful to break up the representation into chunks, or indent parts of the representation. This is supported by the ipython console's "pretty printer". To make this work for a class, add a `_repr_pretty_(self, p, cycle)` (note the *single* underscores) method. You can find more information in the ipython documentation .

"New" ipython console (qtconsole and notebook)

The new ipython consoles, the qtconsole and the ipython notebook support a much richer set of representations for objects. As Brian deals a lot with mathematical objects, in particular the LaTeX and to a lesser extent the HTML formatting capabilities of the ipython notebook are interesting. To support LaTeX representation, implement a `_repr_latex_` method returning the LaTeX code (*including* `$`, `\begin{equation}` or similar). If the object already has a `_latex` method (see *LaTeX representations with sympy* above), this can be as simple as:

```
def _repr_latex_(self):
    return sympy.latex(self, mode='inline')  # wraps the expression in $ .. $
```

The LaTeX rendering only supports a single mathematical block. For complex objects, e.g. *NeuronGroup* it might be useful to have a richer representation. This can be achieved by returning HTML code from `_repr_html_` – this HTML code is processed by MathJax so it can include literal LaTeX code that will be transformed before it is rendered as HTML. An object containing two equations could therefore be represented with a method like this:

```
def _repr_html_(self):
    return '''
    <h3> Equation 1 </h3>
    {eq_1}
    <h3> Equation 2 </h3>
    {eq_2}'''.format(eq_1=sympy.latex(self.eq_1, mode='equation'),
                     eq_2=sympy.latex(self.eq_2, mode='equation'))
```

7.1.4 Defensive programming

One idea for Brian 2 is to make it so that it's more likely that errors are raised rather than silently causing weird bugs. Some ideas in this line:

Synapses.source should be stored internally as a weakref Synapses._source, and Synapses.source should be a computed attribute that dereferences this weakref. Like this, if the source object isn't kept by the user, Synapses won't store a reference to it, and so won't stop it from being deallocated.

We should write an automated test that takes a piece of correct code like:

```
NeuronGroup(N, eqs, reset='V>Vt')
```

and tries replacing all arguments by nonsense arguments, it should always raise an error in this case (forcing us to write code to validate the inputs). For example, you could create a new NonsenseObject class, and do this:

```
nonsense = NonsenseObject()
NeuronGroup(nonsense, eqs, reset='V>Vt')
NeuronGroup(N, nonsense, reset='V>Vt')
NeuronGroup(N, eqs, nonsense)
```

In general, the idea should be to make it hard for something incorrect to run without raising an error, preferably at the point where the user makes the error and not in some obscure way several lines later.

The preferred way to validate inputs is one that handles types in a Pythonic way. For example, instead of doing something like:

```
if not isinstance(arg, (float, int)):
    raise TypeError(...)
```

Do something like:

```
arg = float(arg)
```

(or use try/except to raise a more specific error). In contrast to the `isinstance` check it does not make any assumptions about the type except for its ability to be converted to a float.

This approach is particular useful for numpy arrays:

```
arr = np.asarray(arg)
```

(or `np.asanyarray` if you want to allow for array subclasses like arrays with units or masked arrays). This approach has also the nice advantage that it allows all "array-like" arguments, e.g. a list of numbers.

7.1.5 Documentation

It is very important to maintain documentation. We use the Sphinx documentation generator tools. The documentation is all hand written. Sphinx source files are stored in the `docs_sphinx` folder (currently: `dev/brian2/docs_sphinx`). The HTML files can be generated via the script `dev/tools/docs/build_html_brian2.py` and end up in the `docs` folder (currently: `dev/brian2/docs`).

Most of the documentation is stored directly in the Sphinx source text files, but reference documentation for important Brian classes and functions are kept in the documentation strings of those classes themselves. This is automatically pulled from these classes for the reference manual section of the documentation. The idea is to keep the definitive reference documentation near the code that it documents, serving as both a comment for the code itself, and to keep the documentation up to date with the code.

The reference documentation includes all classes, functions and other objects that are defined in the modules and only documents them in the module where they were defined. This makes it possible to document a class like *`Quantity`* only in *`brian2.units.fundamentalunits`* and not additionally in *`brian2.units`* and `brian2`. This mechanism relies on the `__module__` attribute, in some cases, in particular when wrapping a function with a decorator (e.g. *`check_units`*), this attribute has to be set manually:

```
foo.__module__ = __name__
```

Without this manual setting, the function might not be documented at all or in the wrong module.

In addition to the reference, all the examples in the examples folder are automatically included in the documentation.

Note that you can directly link to github issues using `#issuenumber`, e.g. writing `#33` links to a github issue about running benchmarks for Brian 2: #33. This feature should rarely be used in the main documentation, reserve its use for release notes and important known bugs.

Docstrings

Every module, class, method or function has to start with a docstring, unless it is a private or special method (i.e. starting with _ or __) *and* it is obvious what it does. For example, there is normally no need to document `__str__` with "Return a string representation.".

For the docstring format, we use the our own sphinx extension (in `brian2.utils.sphinxext`) based on numpydoc, allowing to write docstrings that are well readable both in sourcecode as well as in the rendered HTML. We generally follow the format used by numpy

When the docstring uses variable, class or function names, these should be enclosed in single backticks. Class and function/method names will be automatically linked to the corresponding documentation. For classes imported in the main brian2 package, you do not have to add the package name, e.g. writing `` `NeuronGroup` `` is enough. For other classes, you have to give the full path, e.g. `` `brian2.units.fundamentalunits.UnitRegistry` ``. If it is clear from the context where the class is (e.g. within the documentation of *`UnitRegistry`*), consider using the ~ abbreviation: `` `~brian2.units.fundamentalunits.UnitRegistry` `` displays only the class name: *`UnitRegistry`*. Note that you do not have to enclose the exception name in a "Raises" or "Warns" section, or the

class/method/function name in a "See Also" section in backticks, they will be automatically linked (putting backticks there will lead to incorrect display or an error message),

Inline source fragments should be enclosed in double backticks.

Class docstrings follow the same conventions as method docstrings and should document the `__init__` method, the `__init__` method itself does not need a docstring.

Documenting functions and methods

The docstring for a function/method should start with a one-line description of what the function does, without referring to the function name or the names of variables. Use a "command style" for this summary, e.g. "Return the result." instead of "Returns the result." If the signature of the function cannot be automatically extracted because of an decorator (e.g. *`check_units()`*), place a signature in the very first row of the docstring, before the one-line description.

For methods, do not document the `self` parameter, nor give information about the method being static or a class method (this information will be automatically added to the documentation).

Documenting classes

Class docstrings should use the same "Parameters" and "Returns" sections as method and function docstrings for documenting the `__init__` constructor. If a class docstring does not have any "Attributes" or "Methods" section, these sections will be automatically generated with all documented (i.e. having a docstring), public (i.e. not starting with _) attributes respectively methods of the class. Alternatively, you can provide these sections manually. This is useful for example in the *`Quantity`* class, which would otherwise include the documentation of many `ndarray` methods, or when you want to include documentation for functions like `__getitem__` which would otherwise not be documented. When specifying these sections, you only have to state the names of documented methods/attributes but you can also provide direct documentation. For example:

```
Attributes
----------
foo
bar
baz
    This is a description.
```

This can be used for example for class or instance attributes which do not have "classical" docstrings. However, you can also use a special syntax: When defining class attributes in the class body or instance attributes in `__init__` you can use the following variants (here shown for instance attributes):

```
def __init__(a, b, c):
    #: The docstring for the instance attribute a.
    #: Can also span multiple lines
    self.a = a

    self.b = b #: The docstring for self.b (only one line).

    self.c = c
    'The docstring for self.c, directly *after* its definition'
```

Long example of a function docstring

This is a very long docstring, showing all the possible sections. Most of the time no See Also, Notes or References section is needed:

```
def foo(var1, var2, long_var_name='hi') :
"""
A one-line summary that does not use variable names or the function name.

Several sentences providing an extended description. Refer to
variables using back-ticks, e.g. `var1`.

Parameters
----------
var1 : array_like
    Array_like means all those objects -- lists, nested lists, etc. --
    that can be converted to an array.  We can also refer to
    variables like `var1`.
var2 : int
    The type above can either refer to an actual Python type
    (e.g. ``int``), or describe the type of the variable in more
    detail, e.g. ``(N,) ndarray`` or ``array_like``.
Long_variable_name : {'hi', 'ho'}, optional
    Choices in brackets, default first when optional.

Returns
-------
describe : type
    Explanation
output : type
    Explanation
tuple : type
    Explanation
items : type
    even more explaining

Raises
------
BadException
    Because you shouldn't have done that.

See Also
--------
otherfunc : relationship (optional)
newfunc : Relationship (optional), which could be fairly long, in which
          case the line wraps here.
thirdfunc, fourthfunc, fifthfunc

Notes
-----
Notes about the implementation algorithm (if needed).

This can have multiple paragraphs.

You may include some math:

.. math:: X(e^{j\omega } ) = x(n)e^{ - j\omega n}

And even use a greek symbol like :math:`omega` inline.

References
----------
Cite the relevant literature, e.g. [1]_.  You may also cite these
```

```
references in the notes section above.

.. [1] O. McNoleg, "The integration of GIS, remote sensing,
   expert systems and adaptive co-kriging for environmental habitat
   modelling of the Highland Haggis using object-oriented, fuzzy-logic
   and neural-network techniques," Computers & Geosciences, vol. 22,
   pp. 585-588, 1996.

Examples
--------
These are written in doctest format, and should illustrate how to
use the function.

>>> a=[1,2,3]
>>> print [x + 3 for x in a]
[4, 5, 6]
>>> print "a\n\nb"
a
b

"""

pass
```

7.1.6 Logging

Logging in Brian is based on the `logging` module in Python's standard library. In Brian, all logging output is logged to a file (the file name is available in `brian2.utils.logger.TMP_LOG`). This log file will normally be deleted on exit, except if an uncaught exception occured or if *logging.delete_log_on_exit* is set to `False`. The default log level for the logging on the console is "warn".

Every brian module that needs logging should start with the following line, using the *get_logger()* function to get an instance of *BrianLogger*:

```
logger = get_logger(__name__)
```

In the code, logging can then be done via:

```
logger.debug('A debug message')
logger.info('An info message')
logger.warn('A warning message')
logger.error('An error message')
```

If a module logs similar messages in different places or if it might be useful to be able to suppress a subset of messages in a module, add an additional specifier to the logging command, specifying the class or function name, or a method name including the class name (do not include the module name, it will be automatically added as a prefix):

```
logger.debug('A debug message', 'CodeString')
logger.debug('A debug message', 'NeuronGroup.update')
logger.debug('A debug message', 'reinit')
```

If you want to log a message only once, e.g. in a function that is called repeatedly, set the optional `once` keyword to `True`:

```
logger.debug('Will only be shown once', once=True)
logger.debug('Will only be shown once', once=True)
```

The output of debugging looks like this in the log file:

```
2012-10-02 14:41:41,484 DEBUG    brian2.equations.equations.CodeString: A debug message
```

and like this on the console (if the log level is set to "debug"):

```
DEBUG    brian2.equations.equations.CodeString: A debug message
```

Log level recommendations

debug Low-level messages that are not of any interest to the normal user but useful for debugging. A typical example is the source code generated by the code generation module.

info Messages that are not necessary for the user, but possibly helpful in understanding the details of what is going on. An example would be displaying a message about which stateupdater has been chosen automatically after analyzing the equations, when no stateupdater has been specified explicitly.

warn Messages that alert the user to a potential mistake in the code, e.g. two possible solutions for an identifier in an equation. It can also be used to make the user aware that he/she is using an experimental feature, an unsupported compiler or similar. In this case, normally the `once=True` option should be used to raise this warning only once. As a rule of thumb, "common" scripts like the examples provided in the examples folder should normally not lead to any warnings.

error This log level is not used currently in Brian, an exception should be raised instead. It might be useful in "meta-code", running scripts and catching any errors that occur.

Showing/hiding log messages

The user can change the level of displayed log messages by using a static method of *BrianLogger*:

```
BrianLogger.log_level_info() # now also display info messages
```

It is also possible to suppress messages for certain sub-hierarchies by using *BrianLogger.suppress_hierarchy*:

```
# Suppress code generation messages on the console
BrianLogger.suppress_hierarchy('brian2.codegen')
# Suppress preference messages even in the log file
BrianLogger.suppress_hierarchy('brian2.core.preferences',
                               filter_log_file=True)
```

Similarly, messages ending in a certain name can be suppressed with *BrianLogger.suppress_name*:

```
# Suppress resolution conflict warnings
BrianLogger.suppress_name('resolution_conflict')
```

These functions should be used with care, as they suppresses messages independent of the level, i.e. even warning and error messages.

Testing log messages

It is possible to test whether code emits an expected log message using the *catch_logs* context manager. This is normally not necessary for debug and info messages, but should be part of the unit tests for warning messages (*catch_logs* by default only catches warning and error messages):

```
with catch_logs() as logs:
    # code that is expected to trigger a warning
    # ...
    assert len(logs) == 1
    # logs contains tuples of (log level, name, message)
    assert logs[0][0] == 'WARNING' and logs[0][1].endswith('warning_type')
```

7.1.7 Testing

Brian uses the nose package for its testing framework. To check the code coverage of the test suite, we use coverage.py.

Running the test suite

The nosetests tool automatically finds tests in the code. When brian2 is in your Python path or when you are in the main brian2 directory, you can start the test suite with:

```
$ nosetests brian2 --with-doctest
```

This should show no errors or failures but possibly a number of skipped tests. The recommended way however is to import brian2 and call the test function, which gives you convenient control over which tests are run:

```
>>> import brian2
>>> brian2.test()
```

By default, this runs the test suite for all available (runtime) code generation targets. If you only want to test a specific target, provide it as an argument:

```
>>> brian2.test('numpy')
```

If you want to test several targets, use a list of targets:

```
>>> brian2.test(['weave', 'cython'])
```

In addition to the tests specific to a code generation target, the test suite will also run a set of independent tests (e.g. parsing of equations, unit system, utility functions, etc.). To exclude these tests, set the `test_codegen_independent` argument to `False`. Not all available tests are run by default, tests that take a long time are excluded. To include these, set `long_tests` to `True`.

To run the C++ standalone tests, you have to set the `test_standalone` argument to the name of a standalone device. If you provide an empty argument for the runtime code generation targets, you will only run the standalone tests:

```
>>> brian2.test([], test_standalone='cpp_standalone')
```

Checking the code coverage

To check the code coverage under Linux (with coverage and nosetests in your path) and generate a report, use the following commands (this assumes the source code of Brian with the file `.coveragerc` in the directory /path/to/brian):

```
$ coverage run --rcfile=/path/to/brian/.coveragerc $(which nosetests) --with-doctest brian2
$ coverage report
```

Using `coverage html` you can also generate a HTML report which will end up in the directory `htmlcov`.

Writing tests

Generally speaking, we aim for a 100% code coverage by the test suite. Less coverage means that some code paths are never executed so there's no way of knowing whether a code change broke something in that path.

Unit tests

The most basic tests are unit tests, tests that test one kind of functionality or feature. To write a new unit test, add a function called `test_...` to one of the `test_...` files in the `brian2.tests` package. Test files should roughly correspond to packages, test functions should roughly correspond to tests for one function/method/feature. In the test functions, use assertions that will raise an `AssertionError` when they are violated, e.g.:

```
G = NeuronGroup(42, model='dv/dt = -v / (10*ms) : 1')
assert len(G) == 42
```

When comparing arrays, use the `array_equal()` function from `numpy.testing.utils` which takes care of comparing types, shapes and content and gives a nicer error message in case the assertion fails. Never make tests depend on external factors like random numbers – tests should always give the same result when run on the same codebase. You should not only test the expected outcome for the correct use of functions and classes but also that errors are raised when expected. For that you can use the `assert_raises` function (also in `numpy.testing.utils`) which takes an Exception type and a callable as arguments:

```
assert_raises(DimensionMismatchError, lambda: 3*volt + 5*second)
```

Note that you cannot simply write `3*volt + 5*second` in the above example, this would raise an exception before calling assert_raises. Using a callable like the simple lambda expression above makes it possible for `assert_raises` to catch the error and compare it against the expected type. You can also check whether expected warnings are raised, see the documentation of the logging mechanism for details

For simple functions, doctests (see below) are a great alternative to writing classical unit tests.

By default, all tests are executed for all selected code generation targets (see *Running the test suite* above). This is not useful for all tests, some basic tests that for example test equation syntax or the use of physical units do not depend on code generation and need therefore not to be repeated. To execute such tests only once, they can be annotated with a `codegen-independent` attribute, using the `attr` decorator:

```
from nose.plugins.attrib import attr
from brian2 import NeuronGroup

@attr('codegen-independent')
def test_simple():
    # Test that the length of a NeuronGroup is correct
    group = NeuronGroup(5, '')
    assert len(group) == 5
```

Tests that are not "codegen-independent" are by default only executed for the runtimes device, i.e. not for the `cpp_standalone` device, for example. However, many of those tests follow a common pattern that is compatible with standalone devices as well: they set up a network, run it, and check the state of the network afterwards. Such tests can be marked as `standalone-compatible`, using the `attr` decorator in the same way as for `codegen-independent` tests. Since standalone devices usually have an internal state where they store information about arrays, array assignments, etc., they need to be reinitialized after such a test. For that use the `with_setup` decorator and provide the *`restore_device`* function as the `teardown` argument:

```
from nose import with_setup
from nose.plugins.attrib import attr
from numpy.testing.utils import assert_equal
from brian2 import *
```

```
from brian2.devices.device import restore_device

@attr('standalone-compatible')
@with_setup(teardown=restore_initial_state)
def test_simple_run():
    # Check that parameter values of a neuron don't change after a run
    group = NeuronGroup(5, 'v : volt')
    group.v = 'i*mV'
    run(1*ms)
    assert_equal(group.v[:], np.arange(5)*mV)
```

As a rule of thumb:

- If a test does not have a *run* call, mark it as `codegen-independent`
- If a test has only a single *run* and only reads state variable values after the run, mark it as `standalone-compatible` and register the *restore_device* teardown function

Tests can also be written specifically for a standalone device (they then have to include the *set_device* and *build* calls explicitly). In this case tests have to be annotated with the name of the device (e.g. `'cpp_standalone'`) and with `'standalone-only'` to exclude this test from the runtime tests. Also, the device should be restored in the end:

```
from nose import with_setup
from nose.plugins.attrib import attr
from brian2 import *
from brian2.devices.device import restore_device

@attr('cpp_standalone', 'standalone-only')
@with_setup(teardown=restore_initial_state)
def test_cpp_standalone():
    set_device('cpp_standalone')
    # set up simulation
    # run simulation
    device.build(...)
    # check simulation results
```

Doctests

Doctests are executable documentation. In the `Examples` block of a class or function documentation, simply write code copied from an interactive Python session (to do this from ipython, use `%doctestmode`), e.g.:

```
>>> expr = 'a*_b+c5+8+f(A)'
>>> print word_substitute(expr, {'a':'banana', 'f':'func'})
banana*_b+c5+8+func(A)
```

During testing, the actual output will be compared to the expected output and an error will be raised if they don't match. Note that this comparison is strict, e.g. trailing whitespace is not ignored. There are various ways of working around some problems that arise because of this expected exactness (e.g. the stacktrace of a raised exception will never be identical because it contains file names), see the doctest documentation for details.

Doctests can (and should) not only be used in docstrings, but also in the hand-written documentation, making sure that the examples actually work. To turn a code example into a doc test, use the `.. doctest::` directive, see Equations for examples written as doctests. For all doctests, everything that is available after `from brian2 import *` can be used directly. For everything else, add import statements to the doctest code or – if you do not want the import statements to appear in the document – add them in a `.. testsetup::` block. See the documentation for Sphinx's doctest extension for more details.

Doctests are a great way of testing things as they not only make sure that the code does what it is supposed to do but also that the documentation is up to date!

Test attributes

As explained above, the test suite can be run with different subsets of the available tests. For this, tests have to be annotated with the `attr` decorator available from `nose.plugins.attrib`. Currently, the following attributes are understood:

- **standalone**: A C++ standalone test (not run by default when calling `brian2.test()`)
- **codegen-independent**: A test that does not use any code generation (run by default)
- **long**: A test that takes a long time to run (not run by default)

Attributes can be simply given as a string argument to the `attr` decorator:

```
from nose.plugins.attrib import attr

@attr('standalone')
test_for_standalone():
    pass  # ...
```

Correctness tests

[These do not exist yet for brian2]. Unit tests test a specific function or feature in isolation. In addition, we want to have tests where a complex piece of code (e.g. a complete simulation) is tested. Even if it is sometimes impossible to really check whether the result is correct (e.g. in the case of the spiking activity of a complex network), a useful check is also whether the result is *consistent*. For example, the spiking activity should be the same when using code generation for Python or C++. Or, a network could be pickled before running and then the result of the run could be compared to a second run that starts from the unpickled network.

7.2 Units

7.2.1 Casting rules

In Brian 1, a distinction is made between scalars and numpy arrays (including scalar arrays): Scalars could be multiplied with a unit, resulting in a Quantity object whereas the multiplication of an array with a unit resulted in a (unitless) array. Accordingly, scalars where considered as dimensionless quantities for the purpose of unit checking (e.g.. 1 + 1 * mV raised an error) whereas arrays where not (e.g. array(1) + 1 * mV resulted in 1.001 without any errors). Brian 2 no longer makes this distinction and treats both scalars and arrays as dimensionless for unit checking and make all operations involving quantities return a quantity.:

```
>> 1 + 1 * second
DimensionMismatchError: Addition, dimensions were (s) (1)

>> np.array([1]) + 1 * second
DimensionMismatchError: Addition, dimensions were (s) (1)

>> 1 * second  + 1 * second
2.0 * second

>> np.array([1]) * second  + 1 * second
array([ 2.]) * second
```

As one exception from this rule, a scalar or array `0` is considered as having "any unit", i.e. `0 + 1 * second` will result in `1 * second` without a dimension mismatch error and `0 == 0 * mV` will evaluate to `True`. This seems reasonable from a mathematical viewpoint and makes some sources of error disappear. For example, the Python builtin `sum` (not numpy's version) adds the value of the optional argument `start`, which defaults to 0, to its main argument. Without this exception, `sum([1 * mV, 2 * mV])` would therefore raise an error.

The above rules also apply to all comparisons (e.g. == or <) with one further exception: `inf` and `-inf` also have "any unit", therefore an expression like `v <= inf` will never raise an exception (and always return `True`).

7.2.2 Functions and units

ndarray methods

All methods that make sense on quantities should work, i.e. they check for the correct units of their arguments and return quantities with units were appropriate. Most of the methods are overwritten using thin function wrappers:

`wrap_function_keep_dimension`: Strips away the units before giving the array to the method of `ndarray`, then reattaches the unit to the result (examples: sum, mean, max)

`wrap_function_change_dimension`: Changes the dimensions in a simple way that is independent of function arguments, the shape of the array, etc. (examples: sqrt, var, power)

`wrap_function_dimensionless`: Raises an error if the method is called on a quantity with dimensions (i.e. it works on dimensionless quantities).

List of methods

`all, any, argmax, argmax, argsort, clip, compress, conj, conjugate, copy, cumsum, diagonal, dot, dump, dumps, fill, flatten, getfield, item, itemset, max, mean, min, newbyteorder, nonzero, prod, ptp, put, ravel, repeat, reshape, round, searchsorted, setasflat, setfield, setflags, sort, squeeze, std, sum, take, tolist, trace, transpose, var, view`

Notes

- Methods directly working on the internal data buffer (`setfield`, `getfield`, `newbyteorder`) ignore the dimensions of the quantity.
- The type of a quantity cannot be int, therefore `astype` does not quite work when trying to convert the array into integers.
- `choose` is only defined for integer arrays and therefore does not work
- `tostring` and `tofile` only return/save the pure array data without the unit (but you can use `dump` or `dumps` to pickle a quantity array)
- `resize` does not work: `ValueError: cannot resize this array: it does not own its data`
- `cumprod` would result in different dimensions for different elements and is therefore forbidden
- `item` returns a pure Python float by definition
- `itemset` does not check for units

Numpy ufuncs

All of the standard numpy ufuncs (functions that operate element-wise on numpy arrays) are supported, meaning that they check for correct units and return appropriate arrays. These functions are often called implicitly, for example when using operators like < or `**`.

Math operations: `add, subtract, multiply, divide, logaddexp, logaddexp2, true_divide, floor_divide, negative, power, remainder, mod, fmod, absolute, rint, sign, conj, conjugate, exp, exp2, log, log2, log10, expm1, log1p, sqrt, square, reciprocal, ones_like`

Trigonometric functions: `sin, cos, tan, arcsin, arccos, arctan, arctan2, hypot, sinh, cosh, tanh, arcsinh, arccosh, arctanh, deg2rad, rad2deg`

Bitwise functions: `bitwise_and, bitwise_or, bitwise_xor, invert, left_shift, right_shift`

Comparison functions: `greater, greater_equal, less, less_equal, not_equal, equal, logical_and, logical_or, logical_xor, logical_not, maximum, minimum`

Floating functions: `isreal, iscomplex, isfinite, isinf, isnan, floor, ceil, trunc, fmod`

Not taken care of yet: `signbit, copysign, nextafter, modf, ldexp, frexp`

Notes

- Everything involving `log` or `exp`, as well as trigonometric functions only works on dimensionless array (for `arctan2` and `hypot` this is questionable, though)
- Unit arrays can only be raised to a scalar power, not to an array of exponents as this would lead to differing dimensions across entries. For simplicity, this is enforced even for dimensionless quantities.
- Bitwise functions never works on quantities (numpy will by itself throw a `TypeError` because they are floats not integers).
- All comparisons only work for matching dimensions (with the exception of always allowing comparisons to 0) and return a pure boolean array.
- All logical functions treat quantities as boolean values in the same way as floats are treated as boolean: Any non-zero value is True.

Numpy functions

Many numpy functions are functional versions of ndarray methods (e.g. `mean`, `sum`, `clip`). They therefore work automatically when called on quantities, as numpy propagates the call to the respective method.

There are some functions in numpy that do not propagate their call to the corresponding method (because they use np.asarray instead of np.asanyarray, which might actually be a bug in numpy): `trace`, `diagonal`, `ravel`, `dot`. For these, wrapped functions in `unitsafefunctions.py` are provided.

Wrapped numpy functions in unitsafefunctions.py

These functions are thin wrappers around the numpy functions to correctly check for units and return quantities when appropriate:

`log, exp, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh, arctanh, diagonal, ravel, trace, dot`

numpy functions that work unchanged

This includes all functional counterparts of the methods mentioned above (with the exceptions mentioned above). Some other functions also work correctly, as they are only using functions/methods that work with quantities:

- `linspace, diff, digitize` [1]
- `trim_zeros, fliplr, flipud, roll, rot90, shuffle`

[1] But does not care about the units of its input.

- `corrcoeff` [1]

numpy functions that return a pure numpy array instead of quantities

- `arange`
- `cov`
- `random.permutation`
- `histogram, histogram2d`
- `cross, inner, outer`
- `where`

numpy functions that do something wrong

- `insert, delete` (return a quantity array but without units)
- `correlate` (returns a quantity with wrong units)
- `histogramdd` (raises a `DimensionMismatchError`)

User-defined functions and units

For performance and simplicity reasons, code within the Brian core does not use Quantity objects but unitless numpy arrays instead. See Adding support for new functions for details on how to make use user-defined functions with Brian's unit system.

7.3 Equations and namespaces

7.3.1 Equation parsing

Parsing is done via pyparsing, for now find the grammar at the top of the *brian2.equations.equations* file.

7.3.2 Variables

Each Brian object that saves state variables (e.g. *NeuronGroup*, *Synapses*, *StateMonitor*) has a `variables` attribute, a dictionary mapping variable names to `Variable` objects (in fact a `Variables` object, not a simple dictionary). `Variable` objects contain information *about* the variable (name, dtype, units) as well as access to the variable's value via a `get_value` method. Some will also allow setting the values via a corresponding `set_value` method. These objects can therefore act as proxies to the variables' "contents".

`Variable` objects provide the "abstract namespace" corresponding to a chunk of "abstract code", they are all that is needed to check for syntactic correctness, unit consistency, etc.

7.3.3 Namespaces

The `namespace` attribute of a group can contain information about the external (variable or function) names used in the equations. It specifies a group-specific namespace used for resolving names in that group. At run time, this namespace is combined with a "run namespace". This namespace is either explicitly provided to the *Network.run()* method, or the implicit namespace consisting of the locals and globals around the point where the run function is called.

Internally, this is realized via the `before_run` function. At the start of a run, *`Network.before_run()`* calls *`BrianObject.before_run()`* of every object in the network with a namespace argument and a level. If the namespace argument is given (even if it is an empty dictionary), it will be used together with any group-specific namespaces for resolving names. If it is not specified or `None`, the given level will be used to go up in the call frame and determine the respective locals and globals.

7.4 Variables and indices

7.4.1 Introduction

To be able to generate the proper code out of abstract code statements, the code generation process has to have access to information about the variables (their type, size, etc.) as well as to the indices that should be used for indexing arrays (e.g. a state variable of a *`NeuronGroup`* will be indexed differently in the *`NeuronGroup`* state updater and in synaptic propagation code). Most of this information is stored in the `variables` attribute of a *`Group`* (this includes *`NeuronGroup`*, *`Synapses`*, *`PoissonGroup`* and everything else that has state variables). The `variables` attribute can be accessed as a (read-only) dictionary, mapping variable names to `Variable` objects storing the information about the respective variable. However, it is not a simple dictionary but an instance of the `Variables` class. Let's have a look at its content for a simple example:

```
>>> tau = 10*ms
>>> G = NeuronGroup(10, 'dv/dt = -v / tau : volt')
>>> for name, var in G.variables.items():
...     print('%r : %s' % (name, var))
...
'_spikespace' : <ArrayVariable(unit=Unit(1),  dtype=<type 'numpy.int32'>, scalar=False, constant=Fals
'i' : <ArrayVariable(unit=Unit(1),  dtype=<type 'numpy.int32'>, scalar=False, constant=True, is_bool=
'N' : <Constant(unit=Unit(1),  dtype=<type 'numpy.int64'>, scalar=True, constant=True, is_bool=False,
't' : AttributeVariable(unit=second, obj=<Clock defaultclock: t = 0.0 s, dt = 0.1 ms>, attribute='t_
'v' : <ArrayVariable(unit=volt,  dtype=<type 'numpy.float64'>, scalar=False, constant=False, is_bool=
'dt' : AttributeVariable(unit=second, obj=<Clock defaultclock: t = 0.0 s, dt = 0.1 ms>, attribute='dt
```

The state variable `v` we specified for the *`NeuronGroup`* is represented as an `ArrayVariable`, all the other variables were added automatically. By convention, internal names for variables that should not be directly accessed by the user start with an underscore, in the above example the only variable of this kind is `'_spikespace'`, the internal datastructure used to store the spikes that occured in the current time step. There's another array `i`, the neuronal indices (simply an array of integers from 0 to 9), that is used for string expressions involving neuronal indices. The constant `N` represents the total number of neurons, `t` the current time of the clock and `dt` the its timestep. For the latter two variables, `AttributeVariable` is used as these variables can change between runs (`dt`) or during a run (`t`). Using this variable type means that at the beginning of the run (if `constant == True`) or at every time step (if `constant == False`), `getattr(obj, attribute)` is executed and the resulting value is given to the *`CodeObject`* doing the computation.

The information stored in the `Variable` objects is used to do various checks on the level of the abstract code, i.e. before any programming language code is generated. Here are some examples of errors that are caught this way:

```
>>> G.v = 3*ms  # G.variables['v'].unit is volt
Traceback (most recent call last):
...
DimensionMismatchError: Incorrect units for setting v, dimensions were (s) (m^2 kg s^-3 A^-1)
>>> G.N = 5  # G.variables['N'] is read-only
Traceback (most recent call last):
...
TypeError: Variable N is read-only
>>> G2 = NeuronGroup(10, 'dv/dt = -v / tau : volt', threshold='v')  #G2.variables['v'].is_bool is Fa
Traceback (most recent call last):
```

```
...
TypeError: Threshold condition "v" is not a boolean expression
```

7.4.2 Creating variables

Each variable that should be accessible as a state variable and/or should be available for use in abstract code has to be created as a `Variable`. For this, first a `Variables` container with a reference to the group has to be created, individual variables can then be added using the various `add_...` methods:

```
self.variables = Variables(self)
self.variables.add_array('an_array', unit=volt, size=100)
self.variables.add_constant('N', unit=Unit(1), value=self._N, dtype=np.int32)
self.variables.add_clock_variables(self.clock)
```

As an additional argument, array variables can be specified with a specific *index* (see *Indices* below).

7.4.3 References

For each variable, only one `Variable` object exists even if it is used in different contexts. Let's consider the following example:

```
G = NeuronGroup(5, 'dv/dt = -v / tau : volt')
subG = G[2:]
S = Synapses(G, G, pre='v+=1*mV', connect=True)
```

All allow an access to the state variable `v` (note the different shapes, these arise from the different indices used, see below):

```
>>> G.v
<neurongroup.v: array([ 0.,  0.,  0.,  0.,  0.]) * volt>
>>> subG.v
<neurongroup_subgroup.v: array([ 0.,  0.,  0.]) * volt>
>>> S.v
<synapses.v: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
    0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]) * volt>
```

In all of these cases, the `Variables` object stores references to the same `ArrayVariable` object:

```
>>> id(G.variables['v'])
108610960
>>> id(subG.variables['v'])
108610960
>>> id(S.variables['v'])
108610960
```

Such a reference can be added using `Variables.add_reference`, note that the name used for the reference is not necessarily the same as in the original group, e.g. in the above example `S.variables` also stores references to `v` under the names `v_pre` and `v_post`.

7.4.4 Indices

In subgroups and especially in synapses, the transformation of abstract code into executable code is not straightforward because it can involve variables from different contexts. Here is a simple example:

```
G = NeuronGroup(5, 'dv/dt = -v / tau : volt')
S = Synapses(G, G, 'w : volt', pre='v+=w')
```

The seemingly trivial operation `v+=w` involves the variable `v` of the *NeuronGroup* and the variable `w` of the *Synapses* object which have to be indexed in the appropriate way. Since this statement is executed in the context of `S`, the variable indices stored there are relevant:

```
>>> S.variables.indices['w']
'_idx'
>>> S.variables.indices['v']
'_postsynaptic_idx'
```

The index `_idx` has a special meaning and always refers to the "natural" index for a group (e.g. all neurons for a *NeuronGroup*, all synapses for a *Synapses* object, etc.). All other indices have to refer to existing arrays:

```
>>> S.variables['_postsynaptic_idx']
<DynamicArrayVariable(unit=Unit(1),  dtype=<type 'numpy.int32'>, scalar=False, constant=False, is_boo
```

In this case, `_postsynaptic_idx` refers to a dynamic array that stores the postsynaptic targets for each synapse (since it is an array itself, it also has an index. It is defined for each synapse so its index is `_idx` – in fact there is currently no support for an additional level of indirection in Brian: a variable representing an index has to have `_idx` as its own index). Using this index information, the following C++ code (slightly simplified) is generated:

```
for(int _spiking_synapse_idx=0;
    _spiking_synapse_idx<_num_spiking_synapses;
    _spiking_synapse_idx++)
{
    const int _idx = _spiking_synapses[_spiking_synapse_idx];
    const int _postsynaptic_idx = _ptr_array_synapses__synaptic_post[_idx];
    const double w = _ptr_array_synapses_w[_idx];
    double v = _ptr_array_neurongroup_v[_postsynaptic_idx];
    v += w;
    _ptr_array_neurongroup_v[_postsynaptic_idx] = v;
}
```

In this case, the "natural" index `_idx` iterates over all the synapses that received a spike (this is defined in the template) and `_postsynaptic_idx` refers to the postsynaptic targets for these synapses. The variables `w` and `v` are then pulled out of their respective arrays with these indices so that the statement `v += w;` does the right thing.

7.4.5 Getting and setting state variables

When a state variable is accessed (e.g. using `G.v`), the group does not return a reference to the underlying array itself but instead to a `VariableView` object. This is because a state variable can be accessed in different contexts and indexing it with a number/array (e.g. `obj.v[0]`) or a string (e.g. `obj.v['i>3']`) can refer to different values in the underlying array depending on whether the object is the *NeuronGroup*, a *Subgroup* or a *Synapses* object.

The `__setitem__` and `__getitem__` methods in `VariableView` delegate to `VariableView.set_item` and `VariableView.get_item` respectively (which can also be called directly under special circumstances). They analyze the arguments (is the index a number, a slice or a string? Is the target value an array or a string expression?) and delegate the actual retrieval/setting of the values to a method of *Group*:

- Getting with a numerical (or slice) index (e.g. `G.v[0]`): `Group.get_with_index_array`
- Getting with a string index (e.g. `G.v['i>3']`): `Group.get_with_expression`
- Setting with a numerical (or slice) index and a numerical target value (e.g. `G.v[5:] = -70*mV`): `Group.set_with_index_array`

- Setting with a numerical (or slice) index and a string expression value (e.g. `G.v[5:] = (-70+i)*mV`): `Group.set_with_expression`
- Setting with a string index and a string expression value (e.g. `G.v['i>5'] = (-70+i)*mV`): `Group.set_with_expression_conditional`

These methods are annotated with the `device_override` decorator and can therefore be implemented in a different way in certain devices. The standalone device, for example, overrides the all the getting functions and the setting with index arrays. Note that for standalone devices, the "setter" methods do not actually set the values but only note them down for later code generation.

7.4.6 Additional variables and indices

The variables stored in the `variables` attribute of a `Group` can be used everywhere (e.g. in the state updater, in the threshold, the reset, etc.). Objects that depend on these variables, e.g. the `Thresholder` of a *NeuronGroup* add additional variables, in particular `AuxiliaryVariables` that are automatically added to the abstract code: a threshold condition `v > 1` is converted into the statement `_cond = v > 1`; to specify the meaning of the variable `_cond` for the code generation stage (in particular, C++ code generation needs to know the data type) an `AuxiliaryVariable` object is created.

In some rare cases, a specific `variable_indices` dictionary is provided that overrides the indices for variables stored in the `variables` attribute. This is necessary for synapse creation because the meaning of the variables changes in this context: an expression `v>0` does not refer to the `v` variable of all the *connected* postsynaptic variables, as it does under other circumstances in the context of a *Synapses* object, but to the `v` variable of all *possible* targets.

7.5 Preferences system

Each preference looks like `codegen.c.compiler`, i.e. dotted names. Each preference has to be registered and validated. The idea is that registering all preferences ensures that misspellings of a preference value by a user causes an error, e.g. if they wrote `codgen.c.compiler` it would raise an error. Validation means that the value is checked for validity, so `codegen.c.compiler = 'gcc'` would be allowed, but `codegen.c.compiler = 'hcc'` would cause an error.

An additional requirement is that the preferences system allows for extension modules to define their own preferences, including extending the existing core brian preferences. For example, an extension might want to define `extension.*` but it might also want to define a new language for codegen, e.g. `codegen.lisp.*`. However, extensions cannot add preferences to an existing category.

7.5.1 Accessing and setting preferences

Preferences can be accessed and set either keyword-based or attribute-based. To set/get the value for the preference example mentioned before, the following are equivalent:

```
prefs['codegen.c.compiler'] = 'gcc'
prefs.codegen.c.compiler = 'gcc'

if prefs['codegen.c.compiler'] == 'gcc':
    ...
if prefs.codegen.c.compiler == 'gcc':
    ...
```

Using the attribute-based form can be particulary useful for interactive work, e.g. in ipython, as it offers autocompletion and documentation. In ipython, `prefs.codegen.c?` would display a docstring with all the preferences available in the `codegen.c` category.

7.5.2 Preference files

Preferences are stored in a hierarchy of files, with the following order (each step overrides the values in the previous step but no error is raised if one is missing):

- The global defaults are stored in the installation directory.
- The user default are stored in `~/.brian/preferences` (which works on Windows as well as Linux).
- The file `brian_preferences` in the current directory.

7.5.3 Registration

Registration of preferences is performed by a call to *BrianGlobalPreferences.register_preferences*, e.g.:

```
register_preferences(
    'codegen.c',
    'Code generation preferences for the C language',
    'compiler'= BrianPreference(
        validator=is_compiler,
        docs='...',
        default='gcc'),
     ...
    )
```

The first argument `'codegen.c'` is the base name, and every preference of the form `codegen.c.*` has to be registered by this function (preferences in subcategories such as `codegen.c.somethingelse.*` have to be specified separately). In other words, by calling *register_preferences*, a module takes ownership of all the preferences with one particular base name. The second argument is a descriptive text explaining what this category is about. The preferences themselves are provided as keyword arguments, each set to a *BrianPreference* object.

7.5.4 Validation functions

A validation function takes a value for the preference and returns `True` (if the value is a valid value) or `False`. If no validation function is specified, a default validator is used that compares the value against the default value: Both should belong to the same class (e.g. int or str) and, in the case of a *Quantity* have the same unit.

7.5.5 Validation

Setting the value of a preference with a registered base name instantly triggers validation. Trying to set an unregistered preference using keyword or attribute access raises an error. The only exception from this rule is when the preferences are read from configuration files (see below). Since this happens before the user has the chance to import extensions that potentially define new preferences, this uses a special function (`_set_preference`). In this case,for base names that are not yet registered, validation occurs when the base name is registered. If, at the time that *Network.run()* is called, there are unregistered preferences set, a *PreferenceError* is raised.

7.5.6 File format

The preference files are of the following form:

```
a.b.c = 1
# Comment line
[a]
b.d = 2
[a.b]
b.e = 3
```

This would set preferences `a.b.c=1`, `a.b.d=2` and `a.b.e=3`.

7.5.7 Built-in preferences

Brian itself defines the following preferences:

codegen

Code generation preferences

`codegen.loop_invariant_optimisations = True`

Whether to pull out scalar expressions out of the statements, so that they are only evaluated once instead of once for every neuron/synapse/... Can be switched off, e.g. because it complicates the code (and the same optimisation is already performed by the compiler) or because the code generation target does not deal well with it. Defaults to `True`.

`codegen.string_expression_target = 'numpy'`

Default target for the evaluation of string expressions (e.g. when indexing state variables). Should normally not be changed from the default numpy target, because the overhead of compiling code is not worth the speed gain for simple expressions.

Accepts the same arguments as *codegen.target*, except for `'auto'`

`codegen.target = 'auto'`

Default target for code generation.

Can be a string, in which case it should be one of:

- `'auto'` the default, automatically chose the best code generation target available.
- `'weave'` uses `scipy.weave` to generate and compile C++ code, should work anywhere where `gcc` is installed and available at the command line.
- `'cython'`, uses the Cython package to generate C++ code. Needs a working installation of Cython and a C++ compiler.
- `'numpy'` works on all platforms and doesn't need a C compiler but is often less efficient.

Or it can be a `CodeObject` class.

codegen.cpp

C++ compilation preferences

`codegen.cpp.compiler = ''`

Compiler to use (uses default if empty)

Should be gcc or msvc.

`codegen.cpp.define_macros = []`

List of macros to define; each macro is defined using a 2-tuple, where 'value' is either the string to define it to or None to define it without a particular value (equivalent of "#define FOO" in source or -DFOO on Unix C compiler command line).

`codegen.cpp.extra_compile_args = None`

Extra arguments to pass to compiler (if None, use either `extra_compile_args_gcc` or `extra_compile_args_msvs`).

`codegen.cpp.extra_compile_args_gcc = ['-w', '-O3']`

Extra compile arguments to pass to GCC compiler

`codegen.cpp.extra_compile_args_msvc = ['/Ox', '/EHsc', '/w']`

Extra compile arguments to pass to MSVC compiler

`codegen.cpp.extra_link_args = []`

Any extra platform- and compiler-specific information to use when linking object files together.

`codegen.cpp.headers = []`

A list of strings specifying header files to use when compiling the code. The list might look like ["<vector>","'my_header"']. Note that the header strings need to be in a form than can be pasted at the end of a #include statement in the C++ code.

`codegen.cpp.include_dirs = []`

Include directories to use. Note that `$prefix/include` will be appended to the end automatically, where `$prefix` is Python's site-specific directory prefix as returned by `sys.prefix`.

`codegen.cpp.libraries = []`

List of library names (not filenames or paths) to link against.

`codegen.cpp.library_dirs = []`

List of directories to search for C/C++ libraries at link time.

`codegen.cpp.msvc_architecture = ''`

MSVC architecture name (or use system architectue by default).

Could take values such as x86, amd64, etc.

`codegen.cpp.msvc_vars_location = ''`

Location of the MSVC command line tool (or search for best by default).

`codegen.cpp.runtime_library_dirs = []`

List of directories to search for C/C++ libraries at run time.

codegen.generators

Codegen generator preferences (see subcategories for individual languages)

codegen.generators.cpp

C++ codegen preferences

`codegen.generators.cpp.flush_denormals = False`

Adds code to flush denormals to zero.

The code is gcc and architecture specific, so may not compile on all platforms. The code, for reference is:

```
    #define CSR_FLUSH_TO_ZERO         (1 << 15)
    unsigned csr = __builtin_ia32_stmxcsr();
    csr |= CSR_FLUSH_TO_ZERO;
    __builtin_ia32_ldmxcsr(csr);
```

Found at http://stackoverflow.com/questions/2487653/avoiding-denormal-values-in-c.

`codegen.generators.cpp.restrict_keyword = '__restrict'`

The keyword used for the given compiler to declare pointers as restricted.

This keyword is different on different compilers, the default works for gcc and MSVS.

codegen.runtime

Runtime codegen preferences (see subcategories for individual targets)

codegen.runtime.cython

Cython runtime codegen preferences

`codegen.runtime.cython.multiprocess_safe = True`

Whether to use a lock file to prevent simultaneous write access to cython .pyx and .so files.

codegen.runtime.numpy

Numpy runtime codegen preferences

`codegen.runtime.numpy.discard_units = False`

Whether to change the namespace of user-specifed functions to remove units.

core

Core Brian preferences

`core.default_float_dtype = float64`

Default dtype for all arrays of scalars (state variables, weights, etc.).

`core.default_integer_dtype = int32`

Default dtype for all arrays of integer scalars.

`core.outdated_dependency_error = True`

Whether to raise an error for outdated dependencies (`True`) or just a warning (`False`).

core.network

Network preferences

`core.network.default_schedule = ['start', 'groups', 'thresholds', 'synapses', 'resets', 'end']`

Default schedule used for networks that don't specify a schedule.

devices

Device preferences

devices.cpp_standalone

C++ standalone preferences

`devices.cpp_standalone.openmp_threads` = `0`

The number of threads to use if OpenMP is turned on. By default, this value is set to 0 and the C++ code is generated without any reference to OpenMP. If greater than 0, then the corresponding number of threads are used to launch the simulation.

logging

Logging system preferences

`logging.console_log_level` = `'WARNING'`

What log level to use for the log written to the console.

Has to be one of CRITICAL, ERROR, WARNING, INFO or DEBUG.

`logging.delete_log_on_exit` = `True`

Whether to delete the log and script file on exit.

If set to `True` (the default), log files (and the copy of the main script) will be deleted after the brian process has exited, unless an uncaught exception occured. If set to `False`, all log files will be kept.

`logging.file_log` = `True`

Whether to log to a file or not.

If set to `True` (the default), logging information will be written to a file. The log level can be set via the *logging.file_log_level* preference.

`logging.file_log_level` = `'DEBUG'`

What log level to use for the log written to the log file.

In case file logging is activated (see *logging.file_log*), which log level should be used for logging. Has to be one of CRITICAL, ERROR, WARNING, INFO or DEBUG.

`logging.save_script` = `True`

Whether to save a copy of the script that is run.

If set to `True` (the default), a copy of the currently run script is saved to a temporary location. It is deleted after a successful run (unless *logging.delete_log_on_exit* is `False`) but is kept after an uncaught exception occured. This can be helpful for debugging, in particular when several simulations are running in parallel.

`logging.std_redirection` = `True`

Whether or not to redirect stdout/stderr to null at certain places.

This silences a lot of annoying compiler output, but will also hide error messages making it harder to debug problems. You can always temporarily switch it off when debugging. In any case, the output is saved to a file and if an error occurs the name of this file will be printed.

7.6 Adding support for new functions

For a description of Brian's function system from the user point of view, see Functions.

The default functions available in Brian are stored in the `DEFAULT_FUNCTIONS` dictionary. New *`Function`* objects can be added to this dictionary to make them available to all Brian code, independent of its namespace.

To add a new implementation for a code generation target, a `FunctionImplementation` can be added to the *`Function.implementations`* dictionary. The key for this dictionary has to be either a *`CodeGenerator`* class object, or a *`CodeObject`* class object. The *`CodeGenerator`* of a *`CodeObject`* (e.g. *`CPPCodeGenerator`* for *`WeaveCodeObject`*) is used as a fallback if no implementation specific to the *`CodeObject`* class exists.

If a function is already provided for the target language (e.g. it is part of a library imported by default), using the same name, all that is needed is to add an empty `FunctionImplementation` object to mark the function as implemented. For example, `exp` is a standard function in C++:

```
DEFAULT_FUNCTIONS['exp'].implementations[CPPCodeGenerator] = FunctionImplementation()
```

Some functions are implemented but have a different name in the target language. In this case, the `FunctionImplementation` object only has to specify the new name:

```
DEFAULT_FUNCTIONS['arcsin'].implementations[CPPCodeGenerator] = FunctionImplementation('asin')
```

Finally, the function might not exist in the target language at all, in this case the code for the function has to be provided, the exact form of this code is language-specific. In the case of C++, it's a dictionary of code blocks:

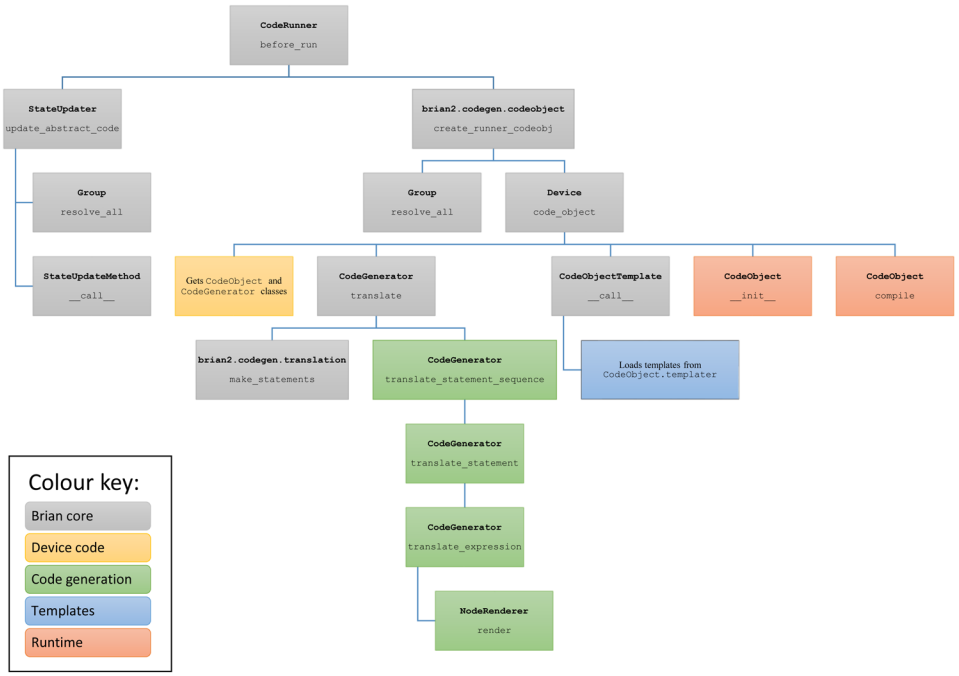
```
clip_code = {'support_code': '''
        double _clip(const float value, const float a_min, const float a_max)
        {
                if (value < a_min)
                    return a_min;
                if (value > a_max)
                    return a_max;
                return value;
        }
        '''}
DEFAULT_FUNCTIONS['clip'].implementations[CPPCodeGenerator] = FunctionImplementation('_clip',
                                                                                      code=clip_code)
```

7.7 Code generation

The generation of a code snippet is done by a *`CodeGenerator`* class. The templates are stored in the `CodeObject.templater` attribute, which is typically implemented as a subdirectory of templates. The compilation and running of code is done by a *`CodeObject`*. See the sections below for each of these.

7.7.1 Code path

The following gives an outline of the key steps that happen for the code generation associated to a *`NeuronGroup`* `StateUpdater`. The items in grey are Brian core functions and methods and do not need to be implemented to create a new code generation target or device. The parts in yellow are used when creating a new device. The parts in green relate to generating code snippets from abstract code blocks. The parts in blue relate to creating new templates which these snippets are inserted into. The parts in red relate to creating new runtime behaviour (compiling and running generated code).



In brief, what happens can be summarised as follows. *Network.run()* will call *BrianObject.before_run()* on each of the objects in the network. Objects such as StateUpdater, which is a subclass of *CodeRunner* use this spot to generate and compile their code. The process for doing this is to first create the abstract code block, done in the StateUpdater.update_abstract_code method. Then, a *CodeObject* is created with this code block. In doing so, Brian will call out to the currently active Device to get the *CodeObject* and *CodeGenerator* classes associated to the device, and this hierarchy of calls gives several hooks which can be changed to implement new targets.

7.7.2 Code generation

To implement a new language, or variant of an existing language, derive a class from *CodeGenerator*. Good examples to look at are the *NumpyCodeGenerator*, *CPPCodeGenerator* and *CythonCodeGenerator* classes in the brian2.codegen.generators package. Each *CodeGenerator* has a class_name attribute which is a string used by the user to refer to this code generator (for example, when defining function implementations).

The derived *CodeGenerator* class should implement the methods marked as NotImplemented in the base *CodeGenerator* class. *CodeGenerator* also has several handy utility methods to make it easier to write these, see the existing examples to get an idea of how these work.

7.7.3 Syntax translation

One aspect of writing a new language is that sometimes you need to translate from Python syntax into the syntax of another language. You are free to do this however you like, but we recommend using a NodeRenderer class which allows you to iterate over the abstract syntax tree of an expression. See examples in brian2.parsing.rendering.

7.7.4 Templates

In addition to snippet generation, you need to create templates for the new language. See the templates directories in brian2.codegen.runtime.* for examples of these. They are written in the Jinja2 templating system. The

location of these templates is set as the `CodeObject.templater` attribute. Examples such as `CPPCodeObject` show how this is done.

7.7.5 Code objects

To allow the final code block to be compiled and run, derive a class from *`CodeObject`*. This class should implement the placeholder methods defined in the base class. The class should also have attributes `templater` (which should be a `Templater` object pointing to the directory where the templates are stored) `generator_class` (which should be the *`CodeGenerator`* class), and `class_name` (which should be a string the user can use to refer to this code generation target.

7.7.6 Default functions

You will typically want to implement the default functions such as the trigonometric, exponential and `rand` functions. We usually put these implementations either in the same module as the *`CodeGenerator`* class or the *`CodeObject`* class depending on whether they are language-specific or runtime target specific. See those modules for examples of implementing these functions.

7.7.7 Code guide

- `brian2.codegen`: everything related to code generation
- `brian2.codegen.generators`: snippet generation, including the *`CodeGenerator`* classes and default function implementations.
- `brian2.codegen.runtime`: templates, compilation and running of code, including *`CodeObject`* and default function implementations.
- `brian2.core.functions`, `brian2.core.variables`: these define the values that variable names can have.
- `brian2.parsing`: tools for parsing expressions, etc.
- `brian2.parsing.rendering`: AST tools for rendering expressions in Python into different languages.
- `brian2.utils`: various tools for string manipulation, file management, etc.

7.7.8 Additional information

For some additional (older, but still accurate) notes on code generation:

Older notes on code generation

The following is an outline of how the Brian 2 code generation system works, with indicators as to which packages to look at and which bits of code to read for a clearer understanding.

We illustrate the global process with an example, the creation and running of a single *`NeuronGroup`* object:

- Parse the equations, add refractoriness to them: this isn't really part of code generation.
- Allocate memory for the state variables.
- Create `Thresholder`, `Resetter` and `StateUpdater` objects.
  - Determine all the variable and function names used in the respective abstract code blocks and templates

  - Determine the abstract namespace, i.e. determine a `Variable` or *`Function`* object for each name.
  - Create a *`CodeObject`* based on the abstract code, template and abstract namespace. This will generate code in the target language and the namespace in which the code will be executed.
- At runtime, each object calls *`CodeObject.__call__()`* to execute the code.

Stages of code generation

Equations to abstract code In the case of *`Equations`*, the set of equations are combined with a numerical integration method to generate an *abstract code block* (see below) which represents the integration code for a single time step.

An example of this would be converting the following equations:

```
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)
```

into the following abstract code using the *`exponential_euler`* method (which is selected automatically):

```
not_refractory = 1*((t - lastspike) > 0.005000)
_BA_v = -v0
_v = -_BA_v + (_BA_v + v)*exp(-dt*not_refractory/tau)
v = _v
```

The code for this stage can be seen in `NeuronGroup.__init__()`, `StateUpdater.__init__`, and `StateUpdater.update_abstract_code` (in `brian2.groups.neurongroup`), and the *`StateUpdateMethod`* classes defined in the `brian2.stateupdaters` package.

For more details, see State update.

Abstract code 'Abstract code' is just a multi-line string representing a block of code which should be executed for each item (e.g. each neuron, each synapse). Each item is independent of the others in abstract code. This allows us to later generate code either for vectorised languages (like numpy in Python) or using loops (e.g. in C++).

Abstract code is parsed according to Python syntax, with certain language constructs excluded. For example, there cannot be any conditional or looping statements at the moment, although support for this is in principle possible and may be added later. Essentially, all that is allowed at the moment is a sequence of arithmetical `a = b*c` style statements.

Abstract code is provided directly by the user for threshold and reset statements in *`NeuronGroup`* and for pre/post spiking events in *`Synapses`*.

Abstract code to snippet We convert abstract code into a 'snippet', which is a small segment of code which is syntactically correct in the target language, although it may not be runnable on its own (that's handled by insertion into a 'template' later). This is handled by the *`CodeGenerator`* object in `brian2.codegen.generators`. In the case of converting into python/numpy code this typically doesn't involve any changes to the code at all because the original code is in Python syntax. For conversion to C++, we have to do some syntactic transformations (e.g. `a**b` is converted to `pow(a, b)`), and add declarations for certain variables (e.g. converting `x=y*z` into `const double x = y*z;`).

An example of a snippet in C++ for the equations above:

```
const double v0 = _ptr_array_neurongroup_v0[_neuron_idx];
const double lastspike = _ptr_array_neurongroup_lastspike[_neuron_idx];
bool not_refractory = _ptr_array_neurongroup_not_refractory[_neuron_idx];
double v = _ptr_array_neurongroup_v[_neuron_idx];
not_refractory = 1 * (t - lastspike > 0.0050000000000000001);
const double _BA_v = -(v0);
const double _v = -(_BA_v) + (_BA_v + v) * exp(-(dt) * not_refractory / tau);
v = _v;
_ptr_array_neurongroup_not_refractory[_neuron_idx] = not_refractory;
_ptr_array_neurongroup_v[_neuron_idx] = v;
```

The code path that includes snippet generation will be discussed in more detail below, since it involves the concepts of namespaces and variables which we haven't covered yet.

Snippet to code block The final stage in the generation of a runnable code block is the insertion of a snippet into a template. These use the Jinja2 template specification language. This is handled in `brian2.codegen.templates`.

An example of a template for Python thresholding:

```
# USES_VARIABLES { not_refractory, lastspike, t }
{% for line in code_lines %}
{{line}}
{% endfor %}
_return_values, = _cond.nonzero()
# Set the neuron to refractory
not_refractory[_return_values] = False
lastspike[_return_values] = t
```

and the output code from the example equations above:

```
# USES_VARIABLES { not_refractory, lastspike, t }
v = _array_neurongroup_v
_cond = v > 10 * mV
_return_values, = _cond.nonzero()
# Set the neuron to refractory
not_refractory[_return_values] = False
lastspike[_return_values] = t
```

Code block to executing code A code block represents runnable code. Brian operates in two different regimes, either in runtime or standalone mode. In runtime mode, memory allocation and overall simulation control is handled by Python and numpy, and code objects operate on this memory when called directly by Brian. This is the typical way that Brian is used, and it allows for a rapid development cycle. However, we also support a standalone mode in which an entire project workspace is generated for a target language or device by Brian, which can then be compiled and run independently of Brian. Each mode has different templates, and does different things with the outputted code blocks. For runtime mode, in Python/numpy code is executed by simply calling the `exec` statement on the code block in a given namespace. For C++/weave code, the `scipy.weave.inline` function is used. In standalone mode, the templates will typically each be saved into different files.

Key concepts

Namespaces In general, a namespace is simply a mapping/dict from names to values. In Brian we use the term 'namespace' in two ways: the high level "abstract namespace" maps names to objects based on the `Variables` or *`Function`* class. In the above example, `v` maps to an `ArrayVariable` object, `tau` to a `Constant` object, etc. This namespace has all the information that is needed for checking the consistency of units, to determine which variables are boolean or scalar, etc. During the *`CodeObject`* creation, this abstract namespace is converted into

the final namespace in which the code will be executed. In this namespace, `v` maps to the numpy array storing the state variable values (without units) and `tau` maps to a concrete value (again, without units). See Equations and namespaces for more details.

Variable `Variable` objects contain information about the variable they correspond to, including details like the data type, whether it is a single value or an array, etc.

See `brian2.core.variables` and, e.g. `Group._create_variables`, `NeuronGroup._create_variables()`.

Templates Templates are stored in Jinja2 format. They come in one of two forms, either they are a single template if code generation only needs to output a single block of code, or they define multiple Jinja macros, each of which is a separate code block. The *`CodeObject`* should define what type of template it wants, and the names of the macros to define. For examples, see the templates in the directories in `brian2/codegen/runtime`. See `brian2.codegen.templates` for more details.

Code guide

This section includes a guide to the various relevant packages and subpackages involved in the code generation process.

`codegen` Stores the majority of all code generation related code.

- **`codegen.functions`** Code related to including functions - built-in and user-defined - in generated code.
- **`codegen.generators`** Each *`CodeGenerator`* is defined in a module here.
- **`codegen.runtime`** Each runtime *`CodeObject`* and its templates are defined in a package here.

`core`

- **`core.variables`** The `Variable` types are defined here.

`equations` Everything related to *`Equations`*.

`groups` All *`Group`* related stuff is in here. The *`Group.resolve()`* methods are responsible for determining the abstract namespace.

`parsing` Various tools using Python's `ast` module to parse user-specified code. Includes syntax translation to various languages in `parsing.rendering`.

`stateupdaters` Everything related to generating abstract code blocks from integration methods is here.

7.8 Devices

This document describes how to implement a new `Device` for Brian. This is a somewhat complicated process, and you should first be familiar with devices from the user point of view (Devices) as well as the code generation system (Code generation).

We wrote Brian's devices system to allow for two major use cases, although it can potentially be extended beyond this. The two use cases are:

1. Runtime mode. In this mode, everything is managed by Python, including memory management (using numpy by default) and running the simulation. Actual computational work can be carried out in several different ways, including numpy, weave or Cython.
2. Standalone mode. In this mode, running a Brian script leads to generating an entire source code project tree which can be compiled and run independently of Brian or Python.

Runtime mode is handled by `RuntimeDevice` and is already implemented, so here I will mainly discuss standalone devices. A good way to understand these devices is to look at the implementation of `CPPStandaloneDevice` (the only one implemented in the core of Brian). In many cases, the simplest way to implement a new standalone device would be to derive a class from `CPPStandaloneDevice` and overwrite just a few methods.

7.8.1 Memory management

Memory is managed primarily via the `Device.add_array`, `Device.get_value` and `Device.set_value` methods. When a new array is created, the `add_array` method is called, and when trying to access this memory the other two are called. The `RuntimeDevice` uses numpy to manage the memory and returns the underlying arrays in these methods. The `CPPStandaloneDevice` just stores a dictionary of array names but doesn't allocate any memory. This information is later used to generate code that will allocate the memory, etc.

7.8.2 Code objects

As in the case of runtime code generation, computational work is done by a collection of *`CodeObject`* s. In `CPPtandaloneDevice`, each code object is converted into a pair of `.cpp` and `.h` files, and this is probably a fairly typical way to do it. For this device, it just uses the same code generation routines as for the runtime C++ device weave.

7.8.3 Building

The method `Device.build` is used to generate the project. This can be implemented any way you like, although looking at `CPPStandaloneDevice.build` is probably a good way to get an idea of how to do it.

7.8.4 Device override methods

Several functions and methods in Brian are decorated with the `device_override` decorator. This mechanism allows a standalone device to override the behaviour of any of these functions by implementing a method with the name provided to `device_override`. For example, the `CPPStandaloneDevice` uses this to override *`Network.run()`* as `CPPStandaloneDevice.network_run`.

7.8.5 Other methods

There are some other methods to implement, including initialising arrays, creating spike queues for synaptic propagation. Take a look at the source code for these.

7.9 Multi-threading with OpenMP

The following is an outline of how to make C++ standalone templates compatible with OpenMP, and therefore make them work in a multi-threaded environment. This should be considered as an extension to Code generation, that has to be read first. The C++ standalone mode of Brian is compatible with OpenMP, and therefore simulations can be launched by users with one or with multiple threads. Therefore, when adding new templates, the developers need to make sure that those templates are properly handling the situation if launched with OpenMP.

7.9.1 Key concepts

All the simulations performed with the C++ standalone mode can be launched with multi-threading, and make use of multiple cores on the same machine. Basically, all the Brian operations that can easily be performed in parallel, such as computing the equations for *NeuronGroup*, *Synapses*, and so on can and should be split among several threads. The network construction, so far, is still performed only by one single thread, and all created objects are shared by all the threads. This is only during the *run()* function that all the objects are executed within a global parallel environment, meaning that all operations will be performed on all the threads, if multi-threading is activated.

7.9.2 Use of `#pragma` flags

In OpenMP, all the parallelism is handled thanks to extra comments, added in the main C++ code, under the form:

```
#pragma omp ...
```

But to avoid any dependencies in the code that is generated by Brian when OpenMP is not activated, we are using functions that will only add those comments, during code generation, when such a multi-threading mode is turned on. By default, nothing will be inserted.

Translations of the `#pragma` commands

All the translations from `openmp_pragma()` calls in the C++ templates are handled in the file `devices/cpp_standalone/codeobject.py` In this function, you can see that all calls with various string inputs will generate #pragma statements inserted into the C++ templates during code generation. For example:

```
{{ openmp_pragma('static') }}
```

will be transformed, during code generation, into:

```
#pragma omp for schedule(static)
```

You can find the list of all the translations in the core of the `openmp_pragma()` function, and if some extra translations are needed, they should be added here.

Execution of the OpenMP code

In this section, we are explaining the main ideas behind the OpenMP mode of Brian, and how the simulation is executed in such a parallel context. As can be seen in `devices/cpp_standalone/templates/main.cpp`, the appropriate number of threads, defined by the user, is fixed at the beginning of the main function in the C++ code with:

```
{{ openmp_pragma('set_num_threads') }}
```

equivalent to (thanks to the `openmp_pragam()` function defined above): nothing if OpenMP is turned off (default), and to:

```
omp_set_dynamic(0);
omp_set_num_threads(nb_threads);
```

otherwise. When OpenMP creates a parallel context, this is the number of threads that will be used. As said, network creation is performed without any calls to OpenMP, on one single thread. The parallelism is only starting in the file `devices/cpp_standalone/templates/network.cpp` with the creation of a parallel context in the *run()* function, as can be seen with:

```
{{ openmp_pragma('parallel') }}
{
    while(clock->running())
    {
      .....
    }
}
```

just before the loop over each time steps of the simulation. Therefore, from now on and until the end of the run() call, all operations are handled in parallel, and will, by default, be executed on all the nodes. This is more efficient, creating the parallel context only once, reducing the overhead. Interestingly, as you can see in this *run()* function, most of the clock operation are only handled by one node, thanks to:

```
{{ openmp_pragma('single') }}
{
    ....
}
```

This command is adding a #pragma flag such that only one node will update the clocks. Remembering that objects are shared by all threads, so we should not have multiple access. This flag is adding an implicit synchronization barrier such that all threads will be sync after such a block of operation. If such a synchronization is not needed, then you can use to `single-nowait` flag.

7.9.3 How to make your template compatible with OpenMP

Use of `IS_OPENMP_COMPATIBLE` flag

If an existing template is not compatible with OpenMP....

Design of a non parallel template

If the template added can not be parallelized, such as for example a *SpikeMonitor* object, then most of its operations should be encapsulated within a `single` flag, such as:

```
{{ openmp_pragma('single-nowait') }}
{
    ....
}
```

By doing so, this will make sure that even when OpenMP is turned on, operations will be safely performed only by one node. This is the safest option, and this *should* be present each time you have a node that will manipulate data structures such as vector, performing operations such as `push_back()`, affecting the data structure. Those operations should not be performed in parallel, leading to inconsistencies or segmentation faults.

Design of a parallel template

To design a parallel template, such as for example `devices/cpp_standalone/templates/common_group.cpp`, you can see that as soon as you have loops that can safely be split across nodes, you just need to add an openmp command in front of those loops:

```
{{openmp_pragma('static')}}
for(int _idx=0; _idx<N; _idx++)
{
```

```
    ...
}
```

By doing so, OpenMP will take care of splitting the indices and each thread will loop only on a subset of indices, sharing the load. By default, the scheduling use for splitting the indices is static, meaning that each node will get the same number of indices: this is the faster scheduling in OpenMP, and it makes sense for *NeuronGroup* or *Synapses* because operations are the same for all indices. By having a look at examples of templates such as `devices/cpp_standalone/templates/statemonitor.cpp`, you can see that you can merge portions of code executed by only one node and portions executed in parallel. In this template, for example, only one node is recording the time and extending the size of the arrays to store the recorded values:

```
{{ openmp_pragma('single') }}
{{_dynamic_t}}.push_back(_clock_t);

// Resize the dynamic arrays
{{ openmp_pragma('single') }}
{{_recorded}}.resize(_new_size, _num_indices);
```

But then, values are written in the arrays by all the nodes:

```
{{ openmp_pragma('static') }}
for (int _i = 0; _i < _num_indices; _i++)
{
    ....
}
```

Initialization of the arrays

Even if we said that all network creation was performed outside of the main parallel context, created in the run() loop of network.cpp, there are still some other parallel contexts that are created and destroyed while initializing the arrays. This can be seen in `devices/cpp_standalone/templates/objects.cpp`, especially in the function _init_arrays(). Because those calls are made outside a parallel context, we need to create one, and that's why there is a call to:

```
{{ openmp_pragma('parallel-static') }}
```

that is transformed into:

```
#pragma omp parallel for schedule(static)
```

This comment will create on the fly an OpenMP parallel context and destroy it just after the loop. This adds a little overhead, but those init calls are not numerous compared to the simulation.

A similar idea can be found in `devices/cpp_standalone/templates/group_variable_set.cpp`: because this template is called outside the main `run()` loop, during network creation, we need to create a parallel context to perform OpenMP operations. This is why we are using there:

```
{{ openmp_pragma('parallel-static') }}
```

instead of simply:

```
{{ openmp_pragma('static') }}
```

7.9.4 Synaptic propagation in parallel

General ideas

With OpenMP, synaptic propagation is also multi-threaded. Therefore, we have to modify the `SynapticPathway` objects, handling spike propagation. As can be seen in `devices/cpp_standalone/templates/synapses_classes.cpp`, such an object, created during run time, will be able to get the number of threads decided by the user:

```
_nb_threads = {{ openmp_pragma('get_num_threads') }};
```

By doing so, a `SynapticPathway`, instead of handling only one `SpikeQueue`, will be divided into `_nb_threads` `SpikeQueues`, each of them handling a subset of the total number of connections. Because all the calls to `SynapticPathway` object are performed within the `run()` loop of `devices/cpp_standalone/templates/network.cpp`, we have to assume that they are performed in a parallel context. This is why all the function of the `SynapticPathway` object are taking care of the node number:

```
void push(int *spikes, unsigned int nspikes)
{
    queue[{{ openmp_pragma('get_thread_num') }}]->push(spikes, nspikes);
}
```

Such a method for the `SynapticPathway` will make sure that when spikes are propagated, all the threads will propagate them to their connections. By default, again, if OpenMP is turned off, the queue vector has size 1.

Preparation of the `SynapticPathway`

Here we are explaining the implementation of the `prepare()` method for `SynapticPathway`. The preparation of the SynapticPathway is performed outside the *`run()`* loop, therefore outside of a parallel context. If we want each thread to prepare its own subset of connections, we need to create a temporary parallel context:

```
{{ openmp_pragma('parallel') }}
{
    unsigned int length;
    if ({{ openmp_pragma('get_thread_num') }} == _nb_threads - 1)
        length = n_synapses - (unsigned int) {{ openmp_pragma('get_thread_num') }}*n_synapses/_nb_thr
    else
        length = (unsigned int) n_synapses/_nb_threads;

    unsigned int padding  = {{ openmp_pragma('get_thread_num') }}*(n_synapses/_nb_threads);

    queue[{{ openmp_pragma('get_thread_num') }}]->openmp_padding = padding;
    queue[{{ openmp_pragma('get_thread_num') }}]->prepare(&real_delays[padding], &sources[padding],
}
```

Then, basically, each threads is getting an equal number of synapses (except the last one, that will get the remaining ones, if the number is not a multiple of `n_threads`), and the queues are receiving a padding integer telling them what part of the synapses belongs to each queue. After that, the parallel context is destroyed, and network creation can continue. Note that this could have been done without a parallel context, in a sequential manner, but this is just speeding up everything.

Selection of the spikes

Here we are explaining the implementation of the `peek()` method for `SynapticPathway`. This is an example of concurrent access to data structures that are not well handled in parallel, such as `std::vector`. When `peek()` is

called, we need to return a vector of all the neuron spiking at that particular time. Therefore, we need to ask every queue of the `SynapticPathway` what are the id of the spiking neurons, and concatenate them. Because those ids are stored in vectors with various shapes, we need to loop over nodes to perform this concatenate, in a sequential manner:

```
{{ openmp_pragma('static-ordered') }}
for(int _thread=0; _thread < {{ openmp_pragma('get_num_threads') }}; _thread++)
{
    {{ openmp_pragma('ordered') }}
    {
        if (_thread == 0)
            all_peek.clear();
        all_peek.insert(all_peek.end(), queue[_thread]->peek()->begin(), queue[_thread]->peek()->end
    }
}
```

The loop, with the keyword 'static-ordered', is therefore performed such that node 0 enters it first, then node 1, and so on. Only one node at a time is executing the block statement. This is needed because vector manipulations can not be performed in a multi-threaded manner. At the end of the loop, `all_peek` is now a vector where all sub queues have written the id of spiking cells, and therefore this is the list of all spiking cells within the `SynapticPathway`.

7.9.5 Compilation of the code

One extra file needs to be modified, in order for OpenMP implementation to work. This is the makefile `devices/cpp_standalone/templates/makefile`. As one can simply see, the CFLAGS are dynamically modified during code generation thanks to:

```
{{ openmp_pragma('compilation') }}
```

If OpenMP is activated, this will add the following dependencies:

```
-fopenmp
```

such that if OpenMP is turned off, nothing, in the generated code, does depend on it.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

[R13] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[R14] Wikipedia, “Inverse hyperbolic function”, http://en.wikipedia.org/wiki/Arccosh

[R15] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[R16] Wikipedia, “Inverse hyperbolic function”, http://en.wikipedia.org/wiki/Arcsinh

[R17] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[R18] Wikipedia, “Inverse hyperbolic function”, http://en.wikipedia.org/wiki/Arctanh

[R19] Wikipedia, “Exponential function”, http://en.wikipedia.org/wiki/Exponential_function

[R20] M. Abramovitz and I. A. Stegun, “Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables,” Dover, 1964, p. 69, http://www.math.sfu.ca/~cbm/aands/page_69.htm

[R21] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. http://www.math.sfu.ca/~cbm/aands/

[R22] Wikipedia, “Logarithm”. http://en.wikipedia.org/wiki/Logarithm

[R23] M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972, pg. 83. http://www.math.sfu.ca/~cbm/aands/

[R24] Wikipedia, “Hyperbolic function”, http://en.wikipedia.org/wiki/Hyperbolic_function

m

n

o

p

s

u

m

n

o

p

s

u

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z