Brian Documentation

Release 1.1.0

Romain Brette, Dan Goodman

February 02, 2009

CONTENTS

The manual for Brian is not yet entirely complete, we are working on filling in the gaps signposted 'TODO'. See also the automatically generated API documentation and the reference sheet. You can also download a PDF version of the documentation here.

CHAPTER

ONE

INTRODUCTION

Brian is a clock driven simulator for spiking neural networks, written in the Python programming language.

The simulator is written almost entirely in Python. The idea is that it can be used at various levels of abstraction without the steep learning curve of software like Neuron, where you have to learn their own programming language to extend their models. As a language, Python is well suited to this task because it is easy to learn, well known and supported, and allows a great deal of flexibility in usage and in designing interfaces and abstraction mechanisms. As an interpreted language, and therefore slower than say C++, Python is not the obvious choice for writing a computationally demanding scientific application. However, the SciPy module for Python provides very efficient linear algebra routines, which means that vectorised code can be very fast.

Here's what the Python web site has to say about themselves:

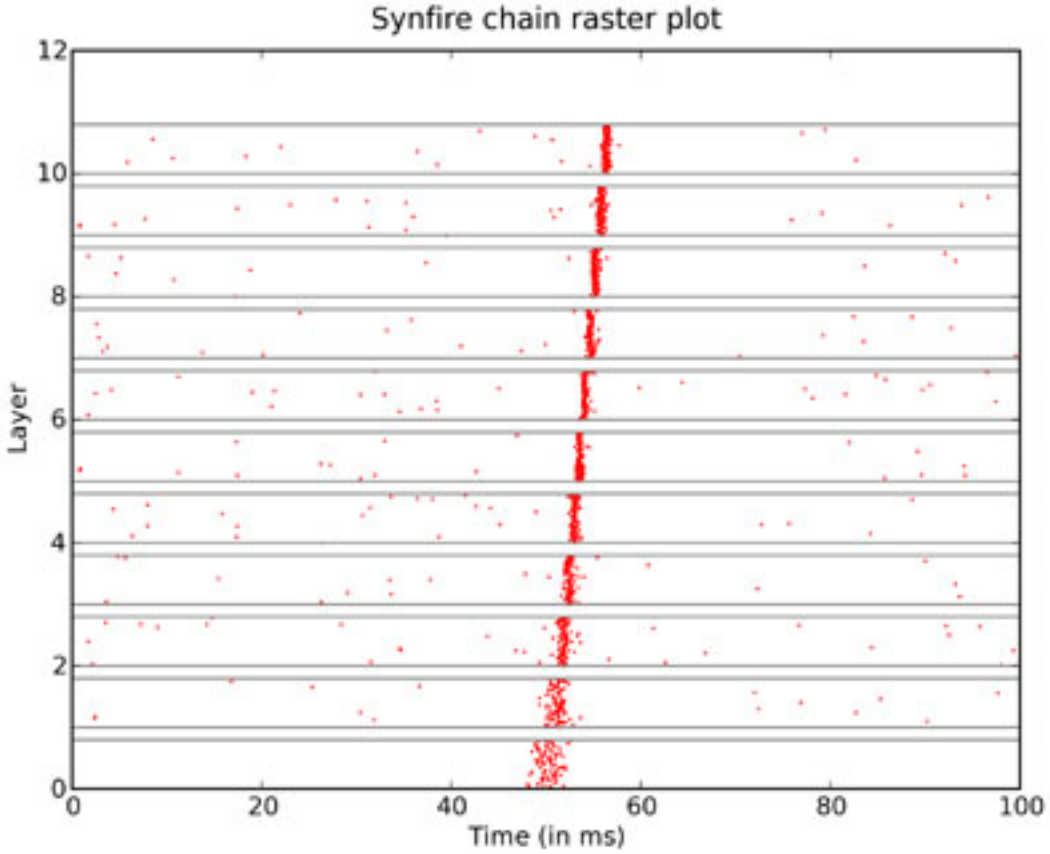
> Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.
>
> The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, http://www.python.org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

As an example of the ease of use and clarity of programs written in Brian, the following script defines and runs a randomly connected network of 4000 integrate and fire neurons with exponential currents:

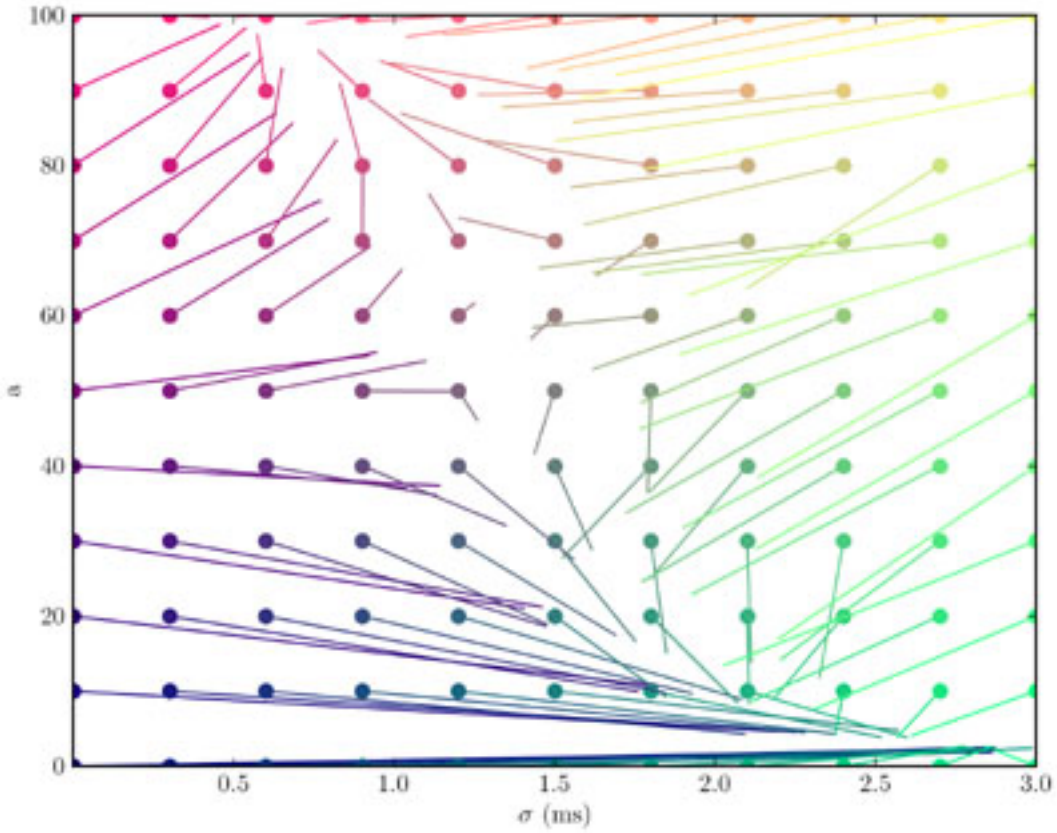
```
from brian import *
eqs='''
dv/dt = (ge+gi-(v+49*mV))/(20*ms) : volt
dge/dt = -ge/(5*ms) : volt
dgi/dt = -gi/(10*ms) : volt
'''
P=NeuronGroup(4000,model=eqs,threshold=-50*mV,reset=-60*mV)
P.v=-60*mV
Pe=P.subgroup(3200)
Pi=P.subgroup(800)
Ce=Connection(Pe,P,'ge',weight=1.62*mV,sparseness=0.02)
Ci=Connection(Pi,P,'gi',weight=-9*mV,sparseness=0.02)
M=SpikeMonitor(P)
run(1*second)
raster_plot(M)
show()
```

As an example of the output of Brian, the following two images reproduce figures from Diesmann et al. 1999 on

synfire chains. The first is a raster plot of a synfire chain showing the stabilisation of the chain.



The simulation of 1000 neurons in 10 layers, each all-to-all connected to the next, using integrate and fire neurons with synaptic noise for 100ms of simulated time took 1 second to run with a timestep of 0.1ms on a 2.4GHz Intel Xeon dual-core processor. The next image is of the state space, figure 3:



The figure computed 50 averages for each of 121 starting points over 100ms at a timestep of 0.1ms and took 201s to run on the same processor as above.

CHAPTER

TWO

INSTALLATION

If you already have a copy of Python 2.5, try the Quick installation below, otherwise take a look at Manual installation. Note: the Windows version of Numpy is currently not compatible with Python 2.6.

2.1 Quick installation

The easiest way to install Brian if you already have a version of Python 2.5 or 2.6 including the `easy_install` script is to simply run the following in a shell:

```
easy_install brian
```

This will download and install Brian and all its required packages (NumPy, SciPy, etc.).

2.2 Manual installation

Installing Brian requires the following components:

1. Python version 2.5.
2. NumPy and Scipy packages for Python: an efficient scientific library.
3. PyLab package for Python: a plotting library similar to Matlab (see the detailed installation instructions).
4. SymPy package for Python: a library for symbolic mathematics (not mandatory yet for Brian).
5. Brian itself (don't forget to download the extras.zip file, which includes examples, tutorials, and a complete copy of the documentation). Download the latest release: file ending `.win32.exe` for Windows, filenames ending `.tar.gz` or `.zip` for other operating systems. Brian is also a Python package and can be installed as explained below.

Fortunately, Python packages are very quick and easy to install, so the whole process shouldn't take very long.

We also recommend using the following for writing programs in Python (see details below):

1. Eclipse IDE with PyDev
2. IPython shell

Finally, if you want to use the (optional) automatic C++ code generation features of Brian, you should have the `gcc` compiler installed (on Cygwin if you are running on Windows).

Mac users: the Scipy Superpack for Intel OS X includes recent versions of Numpy, Scipy, Pylab and IPython.

Windows users: the Python(x,y) distribution includes all the packages (including Eclipse and IPython) above except Brian.

2.2.1 Installing Python packages

On Windows, Python packages (including Brian) are generally installed simply by running an .exe file. On other operating systems, you can download the source release (typically a compressed archive .tar.gz or .zip that you need to unzip) and then install the package by typing the following in your shell:

```
python setup.py install
```

2.2.2 Installing Eclipse

Eclipse is an Integrated Development Environment (IDE) for any programming language. PyDev is a plugin for Eclipse with features specifically for Python development. The combination of these two is excellent for Python development (it's what we use for writing Brian).

To install Eclipse, go to their web page and download any of the base language IDEs. It doesn't matter which one, but Python is not one of the base languages so you have to choose an alternative language. Probably the most useful is the C++ one or the Java one. The C++ one can be downloaded here.

Having downloaded and installed Eclipse, you should download and install the PyDev plugin from their web site. The best way to do this is directly from within the Eclipse IDE. Follow the instructions on the PyDev manual page.

2.2.3 Installing IPython

IPython is an interactive shell for Python. It has features for SciPy and PyLab built in, so it is a good choice for scientific work. Download from their page. If you are using Windows, you will also need to download PyReadline from the same page.

2.2.4 C++ compilers

The default for Brian is to use the `gcc` compiler which will be installed already on most unix or linux distributions. If you are using Windows, you can install cygwin (make sure to include the `gcc` package). Alternatively, some but not all versions of Microsoft Visual C++ should be compatible, but this is untested so far. See the documentation for the SciPy Weave package for more information on this.

2.3 Testing

You can test whether Brian has installed properly by running Python and typing the following two lines:

```
from brian import *
run_all_tests()
```

A series of tests should run and return 'ok' for each one. If not, and all of the packages other than Brian work OK, please let us know.

CHAPTER

THREE

GETTING STARTED

3.1 Tutorials

These tutorials cover some basic topics in writing Brian scripts in Python. The complete source code for the tutorials is available in the tutorials folder in the extras package.

3.1.1 Tutorials for Python and Scipy

Python

The first thing to do in learning how to use Brian is to have a basic grasp of the Python programming language. There are lots of good tutorials already out there. The best one is probably the official Python tutorial. There is also a course for biologists at the Pasteur Institute: Introduction to programming using Python.

NumPy, SciPy and Pylab

The first place to look is the SciPy documentation website. To start using Brian, you do not need to understand much about how NumPy and SciPy work, although understanding how their array structures work will be useful for more advanced uses of Brian.

The syntax of the Numpy and Pylab functions is very similar to Matlab. If you already know Matlab, you could read this tutorial: NumPy for Matlab users and this list of Matlab-Python translations (pdf version here). A tutorial is also available on the web site of Pylab.

3.1.2 Tutorial 1: Basic Concepts

In this tutorial, we introduce some of the basic concepts of a Brian simulation:

- Importing the Brian module into Python
- Using quantities with units
- Defining a neuron model by its differential equation
- Creating a group of neurons
- Running a network
- Looking at the output of the network
- Modifying the state variables of the network directly

- Defining the network structure by connecting neurons
- Doing a raster plot of the output
- Plotting the membrane potential of an individual neuron

The following Brian classes will be introduced:

- `NeuronGroup`
- `Connection`
- `SpikeMonitor`
- `StateMonitor`

We will build a Brian program that defines a randomly connected network of integrate and fire neurons and plot its output.

This tutorial assumes you know:

- The very basics of Python, the `import` keyword, variables, basic arithmetical expressions, calling functions, lists
- The simplest leaky integrate and fire neuron model

The best place to start learning Python is the official tutorial:

> http://docs.python.org/tut/

Tutorial contents

Tutorial 1a: The simplest Brian program

Importing the Brian module

The first thing to do in any Brian program is to load Brian and the names of its functions and classes. The standard way to do this is to use the Python `from ... import *` statement.

```
from brian import *
```

Integrate and Fire model

The neuron model we will use in this tutorial is the simplest possible leaky integrate and fire neuron, defined by the differential equation:

> tau dV/dt = -(V-El)

and with a threshold value Vt and reset value Vr.

Parameters

Brian has a system for defining physical quantities (quantities with a physical dimension such as time). The code below illustrates how to use this system, which (mostly) works just as you'd expect.

```
tau = 20*msecond        # membrane time constant
Vt  =-50*mvolt          # spike threshold
Vr  =-60*mvolt          # reset value
El  =-60*mvolt          # resting potential (same as the reset)
```

The built in standard units in Brian consist of all the fundamental SI units like second and metre, along with a selection of derived SI units such as volt, farad, coulomb. All names are lowercase following the SI standard. In addition, there are scaled versions of these units using the standard SI prefixes m=1/1000, K=1000, etc.

Neuron model and equations

The simplest way to define a neuron model in Brian is to write a list of the differential equations that define it. For the moment, we'll just give the simplest possible example, a single differential equation. You write it in the following form:

```
dx/dt = f(x) : unit
```

where `x` is the name of the variable, `f(x)` can be any valid Python expression, and `unit` is the physical units of the variable `x`. In our case we will write:

```
dV/dt = -(V-El)/tau : volt
```

to define the variable `V` with units `volt`.

To complete the specification of the model, we also define a threshold and reset value and create a group of 40 neurons with this model.

```
G = NeuronGroup(N=40, model='dV/dt = -(V-El)/tau : volt',
                threshold=Vt, reset=Vr)
```

The statement creates a new object 'G' which is an instance of the Brian class `NeuronGroup`, initialised with the values in the line above and 40 neurons. In Python, you can call a function or initialise a class using keyword arguments as well as ordered arguments, so if I defined a function `f(x,y)` I could call it as `f(1,2)` or as `f(y=2,x=1)` and get the same effect. See the Python tutorial for more information on this.

For the moment, we leave the neurons in this group unconnected to each other, each evolves separately from the others.

Simulation

Finally, we run the simulation for 1 second of simulated time. By default, the simulator uses a timestep dt = 0.1 ms.

```
run(1*second)
```

And that's it! To see some of the output of this network, go to the next part of the tutorial.

Exercise

The units system of Brian is useful for ensuring that everything is consistent, and that you don't make hard to find mistakes in your code by using the wrong units. Try changing the units of one of the parameters and see what happens.

Solution

You should see an error message with a Python traceback (telling you which functions were being called when the error happened), ending in a line something like:

```
Brian.units.DimensionMismatchError: The differential equations
are not homogeneous!, dimensions were (m^2 kg s^-3 A^-1)
(m^2 kg s^-4 A^-1)
```

Tutorial 1b: Counting spikes

In the previous part of the tutorial we looked at the following:

- Importing the Brian module into Python
- Using quantities with units
- Defining a neuron model by its differential equation
- Creating a group of neurons
- Running a network

In this part, we move on to looking at the output of the network.

The first part of the code is the same.

```
from brian import *

tau = 20*msecond        # membrane time constant
Vt  =-50*mvolt          # spike threshold
Vr  =-60*mvolt          # reset value
El  =-60*mvolt          # resting potential (same as the reset)

G = NeuronGroup(N=40, model='dV/dt = -(V-El)/tau : volt',
                threshold=Vt, reset=Vr)
```

Counting spikes

Now we would like to have some idea of what this network is doing. In Brian, we use monitors to keep track of the behaviour of the network during the simulation. The simplest monitor of all is the `SpikeMonitor`, which just records the spikes from a given `NeuronGroup`.

```
M = SpikeMonitor(G)
```

Results

Now we run the simulation as before:

```
run(1*second)
```

And finally, we print out how many spikes there were:

```
print M.nspikes
```

So what's going on? Why are there 40 spikes? Well, the answer is that the initial value of the membrane potential for every neuron is 0 mV, which is above the threshold potential of -50 mV and so there is an initial spike at t=0 and then it resets to -60 mV and stays there, below the threshold potential. In the next part of this tutorial, we'll make sure there are some more spikes to see.

Tutorial 1c: Making some activity

In the previous part of the tutorial we found that each neuron was producing only one spike. In this part, we alter the model so that some more spikes will be generated. What we'll do is alter the resting potential `El` so that it is above threshold, this will ensure that some spikes are generated. The first few lines remain the same:

```
from brian import *

tau = 20*msecond        # membrane time constant
Vt  =-50*mvolt          # spike threshold
Vr  =-60*mvolt          # reset value
```

But we change the resting potential to -49 mV, just above the spike threshold:

```
El  =-49*mvolt          # resting potential (same as the reset)
```

And then continue as before:

```
G = NeuronGroup(N=40, model='dV/dt = -(V-El)/tau : volt',
                threshold=Vt, reset=Vr)

M = SpikeMonitor(G)

run(1*second)

print M.nspikes
```

Running this program gives the output `840`. That's because every neuron starts at the same initial value and proceeds deterministically, so that each neuron fires at exactly the same time, in total 21 times during the 1s of the run.

In the next part, we'll introduce a random element into the behaviour of the network.

Exercises

1. Try varying the parameters and seeing how the number of spikes generated varies.

2. Solve the differential equation by hand and compute a formula for the number of spikes generated. Compare this with the program output and thereby partially verify it. (Hint: each neuron starts at above the threshold and so fires a spike immediately.)

Solution

Solving the differential equation gives:

V = El + (Vr-El) exp (-t/tau)

Setting V=Vt at time t gives:

t = tau log((Vr-El) / (Vt-El))

If the simulator runs for time T, and fires a spike immediately at the beginning of the run it will then generate n spikes, where:

n = [T/t] + 1

If you have m neurons all doing the same thing, you get nm spikes. This calculation with the parameters above gives:

t = 48.0 ms n = 21 nm = 840

As predicted.

Tutorial 1d: Introducing randomness

In the previous part of the tutorial, all the neurons start at the same values and proceed deterministically, so they all spike at exactly the same times. In this part, we introduce some randomness by initialising all the membrane potentials to uniform random values between the reset and threshold values.

We start as before:

```
from brian import *

tau = 20*msecond        # membrane time constant
Vt  =-50*mvolt          # spike threshold
Vr  =-60*mvolt          # reset value
El  =-49*mvolt          # resting potential (same as the reset)

G = NeuronGroup(N=40, model='dV/dt = -(V-El)/tau : volt',
                threshold=Vt, reset=Vr)

M = SpikeMonitor(G)
```

But before we run the simulation, we set the values of the membrane potentials directly. The notation `G.V` refers to the array of values for the variable `V` in group `G`. In our case, this is an array of length 40. We set its values by generating an array of random numbers using Brian's `rand` function. The syntax is `rand(size)` generates an array of length `size` consisting of uniformly distributed random numbers in the interval 0, 1.

```
G.V=Vr+rand(40)*(Vt-Vr)
```

And now we run the simulation as before.

```
run(1*second)

print M.nspikes
```

But this time we get a varying number of spikes each time we run it, roughly between 800 and 850 spikes. In the next part of this tutorial, we introduce a bit more interest into this network by connecting the neurons together.

Tutorial 1e: Connecting neurons

In the previous parts of this tutorial, the neurons are still all unconnected. We add in connections here. The model we use is that when neuron i is connected to neuron j and neuron i fires a spike, then the membrane potential of neuron j is instantaneously increased by a value `psp`. We start as before:

```
from brian import *

tau = 20*msecond        # membrane time constant
Vt  =-50*mvolt          # spike threshold
Vr  =-60*mvolt          # reset value
El  =-49*mvolt          # resting potential (same as the reset)
```

Now we include a new parameter, the PSP size:

```
psp = 0.5*mvolt         # postsynaptic potential size
```

And continue as before:

```
G = NeuronGroup(N=40, model='dV/dt = -(V-El)/tau : volt',
                threshold=Vt, reset=Vr)
```

Connections

We now proceed to connect these neurons. Firstly, we declare that there is a connection from neurons in `G` to neurons in `G`. For the moment, this is just something that is necessary to do, the reason for doing it this way will become clear in the next tutorial.

```
C = Connection(G,G)
```

Now the interesting part, we make these neurons be randomly connected with probability 0.1 and weight `psp`. Each neuron i in `G` will be connected to each neuron j in `G` with probability 0.1. The weight of the connection is the amount that is added to the membrane potential of the target neuron when the source neuron fires a spike.

```
C.connect_random(sparseness=0.1,weight=psp)
```

These two previous lines could be done in one line:

```
C = Connection(G,G,sparseness=0.1,weight=psp)
```

Now we continue as before:

```
M = SpikeMonitor(G)

G.V=Vr+rand(40)*(Vt-Vr)

run(1*second)

print M.nspikes
```

You can see that the number of spikes has jumped from around 800-850 to around 1000-1200. In the next part of the tutorial, we'll look at a way to plot the output of the network.

Exercise

Try varying the parameter `psp` and see what happens. How large can you make the number of spikes output by the network? Why?

Solution

The logically maximum number of firings is 400,000 = 40 * 1000 / 0.1, the number of neurons in the network * the time it runs for / the integration step size (you cannot have more than one spike per step).

In fact, the number of firings is bounded above by 200,000. The reason for this is that the network updates in the following way:

1. Integration step
2. Find neurons above threshold
3. Propagate spikes
4. Reset neurons which spiked

You can see then that if neuron i has spiked at time t, then it will not spike at time t+dt, even if it receives spikes from another neuron. Those spikes it receives will be added at step 3 at time t, then reset to `Vr` at step 4 of time t, then the thresholding function at time t+dt is applied at step 2, before it has received any subsequent inputs. So the most a neuron can spike is every other time step.

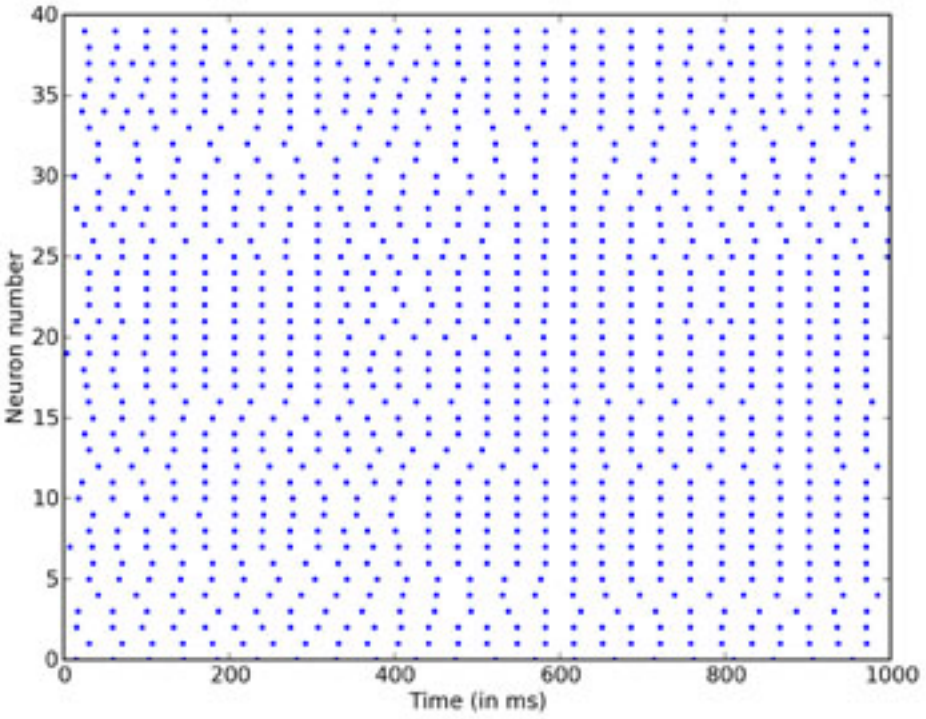
Tutorial 1f: Recording spikes

In the previous part of the tutorial, we defined a network with not entirely trivial behaviour, and printed the number of spikes. In this part, we'll record every spike that the network generates and display a raster plot of them. We start as before:

```
from brian import *

tau = 20*msecond        # membrane time constant
Vt  =-50*mvolt          # spike threshold
Vr  =-60*mvolt          # reset value
El  =-49*mvolt          # resting potential (same as the reset)
psp = 0.5*mvolt         # postsynaptic potential size

G = NeuronGroup(N=40, model='dV/dt = -(V-El)/tau : volt',
                threshold=Vt, reset=Vr)

C = Connection(G,G)
C.connect_random(sparseness=0.1,weight=psp)

M = SpikeMonitor(G)

G.V=Vr+rand(40)*(Vt-Vr)

run(1*second)

print M.nspikes
```

Having run the network, we simply use the `raster_plot()` function provided by Brian. After creating plots, we have to use the `show()` function to display them. This function is from the PyLab module that Brian uses for its built in plotting routines.

```
raster_plot()
show()
```



As you can see, despite having introduced some randomness into our network, the output is very regular indeed. In the next part we introduce one more way to plot the output of a network.

Tutorial 1g: Recording membrane potentials

In the previous part of this tutorial, we plotted a raster plot of the firing times of the network. In this tutorial, we introduce a way to record the value of the membrane potential for a neuron during the simulation, and plot it. We continue as before:

```
from brian import *

tau = 20*msecond        # membrane time constant
Vt  =-50*mvolt          # spike threshold
Vr  =-60*mvolt          # reset value
El  =-49*mvolt          # resting potential (same as the reset)
psp = 0.5*mvolt         # postsynaptic potential size

G = NeuronGroup(N=40, model='dV/dt = -(V-El)/tau : volt',
                threshold=Vt, reset=Vr)

C = Connection(G,G)
C.connect_random(sparseness=0.1,weight=psp)
```

This time we won't record the spikes.

Recording states

Now we introduce a second type of monitor, the `StateMonitor`. The first argument is the group to monitor, and the second is the state variable to monitor. The keyword `record` can be an integer, list or the value `True`. If it is an integer `i`, the monitor will record the state of the variable for neuron `i`. If it's a list of integers, it will record the states for each neuron in the list. If it's set to `True` it will record for all the neurons in the group.

```
M = StateMonitor(G,'V',record=0)
```

And then we continue as before:

```
G.V=Vr+rand(40)*(Vt-Vr)
```

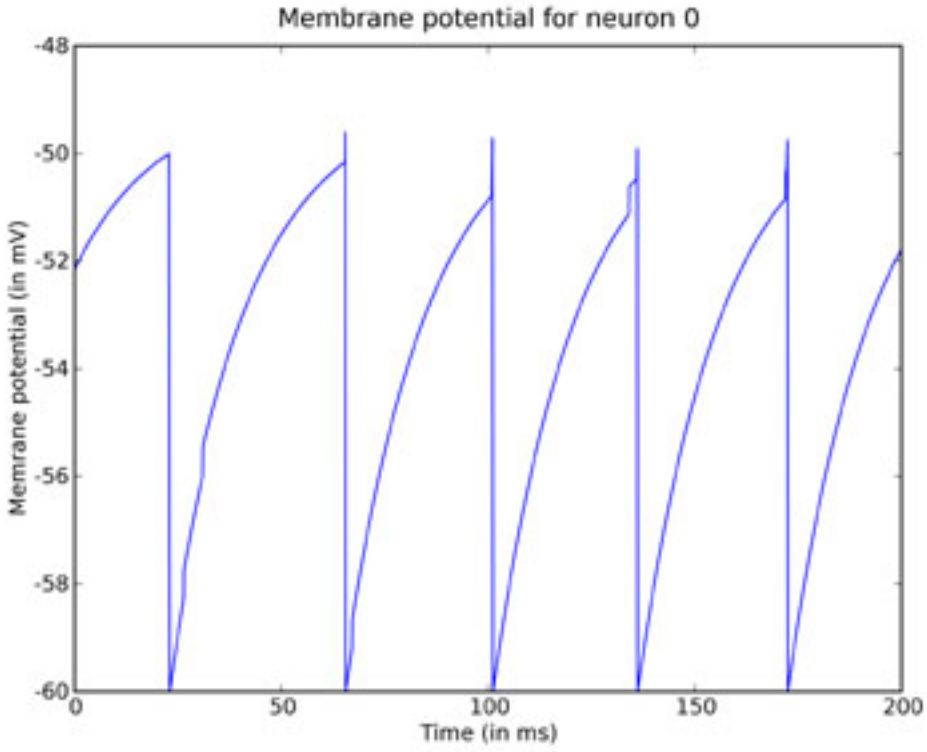
But this time we run it for a shorter time so we can look at the output in more detail:

```
run(200*msecond)
```

Having run the simulation, we plot the results using the `plot` command from PyLab which has the same syntax as the Matlab `plot` command, i.e. `plot(xvals,yvals,...)`. The `StateMonitor` monitors the times at which it monitored a value in the array `M.times`, and the values in the array `M[0]`. The notation `M[i]` means the array of values of the monitored state variable for neuron `i`.

In the following lines, we scale the times so that they're measured in ms and the values so that they're measured in mV. We also label the plot using PyLab's `xlabel`, `ylabel` and `title` functions, which again mimic the Matlab equivalents.

```
plot(M.times/ms,M[0]/mV)
xlabel('Time (in ms)')
ylabel('Membrane potential (in mV)')
title('Membrane potential for neuron 0')
show()
```



You can clearly see the leaky integration exponential decay toward the resting potential, as well as the jumps when a spike was received.

3.1.3 Tutorial 2: Connections

In this tutorial, we will cover in more detail the concept of a `Connection` in Brian.

Tutorial contents

Tutorial 2a: The concept of a Connection

The network

In this first part, we'll build a network consisting of three neurons. The first two neurons will be under direct control and have no equations defining them, they'll just produce spikes which will feed into the third neuron. This third neuron has two different state variables, called Va and Vb. The first two neurons will be connected to the third neuron, but a spike arriving at the third neuron will be treated differently according to whether it came from the first or second neuron (which you can consider as meaning that the first two neurons have different types of synapses on to the third neuron).

The program starts as follows.

```
from brian import *

tau_a =  1*ms
tau_b = 10*ms
Vt    = 10*mV
Vr    =  0*mV
```

Differential equations

This time, we will have multiple differential equations. We will use the `Equations` object, although you could equally pass the multi-line string defining the differential equations directly when initialising the `NeuronGroup` object (see the next part of the tutorial for an example of this).

```
eqs = Equations('''
      dVa/dt = -Va/tau_a : volt
      dVb/dt = -Vb/tau_b : volt
      ''')
```

So far, we have defined a model neuron with two state variables, `Va` and `Vb`, which both decay exponentially towards 0, but with different time constants `tau_a` and `tau_b`. This is just so that you can see the difference between them more clearly in the plot later on.

SpikeGeneratorGroup

Now we introduce the `SpikeGeneratorGroup` class. This is a group of neurons without a model, which just produces spikes at the times that you specify. You create a group like this by writing:

```
G = SpikeGeneratorGroup(N,spiketimes)
```

where `N` is the number of neurons in the group, and `spiketimes` is a list of pairs `(i,t)` indicating that neuron `i` should fire at time `t`. In fact, `spiketimes` can be any 'iterable container' or 'generator', but we don't cover that here (see the detailed documentation for `SpikeGeneratorGroup`).

In our case, we want to create a group with two neurons, the first of which (neuron 0) fires at times 1 ms and 4 ms, and the second of which (neuron 1) fires at times 2 ms and 3 ms. The list of `spiketimes` then is:

```
spiketimes = [(0,1*ms), (0,4*ms),
              (1,2*ms), (1,3*ms)]
```

and we create the group as follows:

```
G1 = SpikeGeneratorGroup(2,spiketimes)
```

Now we create a second group, with one neuron, according to the model we defined earlier.

```
G2 = NeuronGroup(N=1, model=eqs, threshold=Vt, reset=Vr)
```

Connections

In Brian, a `Connection` from one `NeuronGroup` to another is defined by writing:

```
C = Connection(G,H,state)
```

Here `G` is the source group, `H` is the target group, and `state` is the name of the target state variable. When a neuron `i` in `G` fires, Brian finds all the neurons `j` in `H` that `i` in `G` is connected to, and adds the amount `C[i,j]` to the specified state variable of neuron `j` in `H`. Here `C[i,j]` is the (i,j)th entry of the connection matrix of `C` (which is initially all zero).

To start with, we create two connections from the group of two directly controlled neurons to the group of one neuron with the differential equations. The first connection has the target state `Va` and the second has the target state `Vb`.

```
C1 = Connection(G1,G2,'Va')
C2 = Connection(G1,G2,'Vb')
```

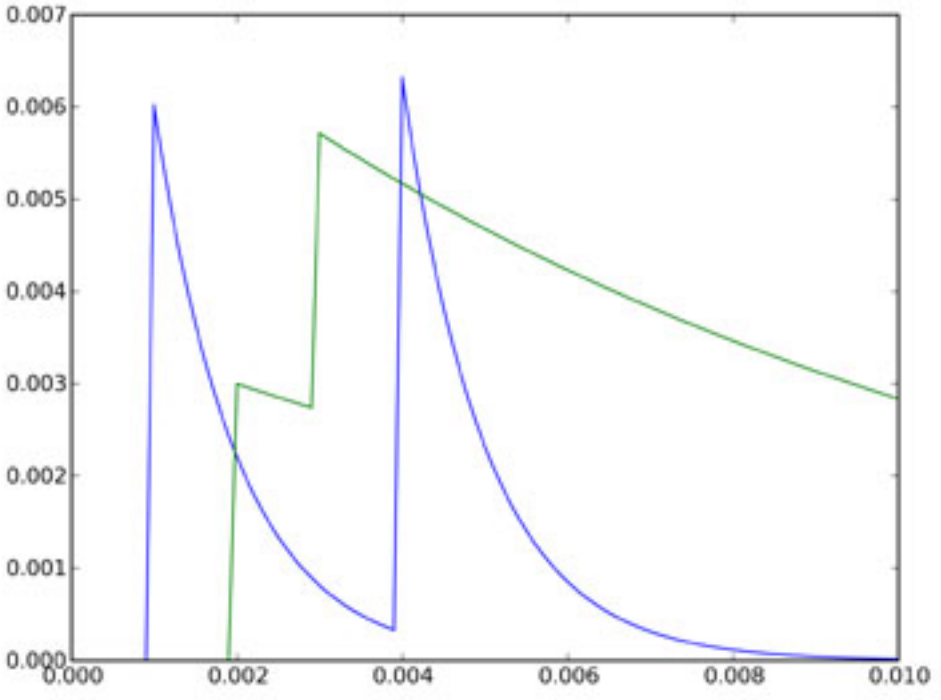
So far, this only declares our intention to connect neurons in group `G1` to neurons in group `G2`, because the connection matrix is initially all zeros. Now, with connection `C1` we connect neuron 0 in group `G1` to neuron 0 in group `G2`, with weight 3 mV. This means that when neuron 0 in group `G1` fires, the state variable `Va` of the neuron in group `G2` will be increased by 6 mV. Then we use connection `C2` to connection neuron 1 in group `G1` to neuron 0 in group `G2`, this time with weight 3 mV.

```
C1[0,0] = 6*mV
C2[1,0] = 3*mV
```

The net effect of this is that when neuron 0 of `G1` fires, `Va` for the neuron in `G2` will increase 6 mV, and when neuron 1 of `G1` fires, `Vb` for the neuron in `G2` will increase 3 mV.

Now we set up monitors to record the activity of the network, run it and plot it.

```
Ma = StateMonitor(G2,'Va',record=True)
Mb = StateMonitor(G2,'Vb',record=True)

run(10*ms)

plot(Ma.times,Ma[0])
plot(Mb.times,Mb[0])
show()
```



The two plots show the state variables `Va` and `Vb` for the single neuron in group `G2`. `Va` is shown in blue, and `Vb` in green. According to the differential equations, `Va` decays much faster than `Vb` (time constant 1 ms rather than 10 ms), but we have set it up (through the connection strengths) that an incoming spike from neuron 0 of `G1` causes a large increase of 6 mV to `Va`, whereas a spike from neuron 1 of `G1` causes a smaller increase of 3 mV to Vb. The value for `Va` then jumps at times 1 ms and 4 ms, when we defined neuron 0 of `G1` to fire, and decays almost back to rest in between. The value for `Vb` jumps at times 2 ms and 3 ms, and because the times are closer together and the time constant is longer, they add together.

In the next part of this tutorial, we'll see how to use this system to do something useful.

Exercises

1. Try playing with the parameters `tau_a`, `tau_b` and the connection strengths, `C1[0,0]` and `C2[0,1]`. Try changing the list of spike times.
2. In this part of the tutorial, the states `Va` and `Vb` are independent of one another. Try rewriting the differential equations so that they're not independent and play around with that.
3. Write a network with inhibitory and excitatory neurons. Hint: you only need one connection.
4. Write a network with inhibitory and excitatory neurons whose actions have different time constants (for example, excitatory neurons have a slower effect than inhibitory ones).

Solutions

1. Simple write `C[i,j]=-3*mV` to make the connection from neuron i to neuron j inhibitory.
2. See the next part of this tutorial.

Tutorial 2b: Excitatory and inhibitory currents

In this tutorial, we use multiple connections to solve a real problem, how to implement two types of synapses with excitatory and inhibitory currents with different time constants.

The scheme

The scheme we implement is the following diffential equations:

taum dV/dt = -V + ge - gi
taue dge/dt = -ge
taui dgi/dt = -gi

An excitatory neuron connects to state ge, and an inhibitory neuron connects to state gi. When an excitatory spike arrives, ge instantaneously increases, then decays exponentially. Consequently, V will initially but continuously rise and then fall. Solving these equations, if V(0)=0, ge(0)=g0 corresponding to an excitatory spike arriving at time 0, and gi(0)=0 then:

gi = 0
ge = g0 exp(-t/taue)
V = (exp(-t/taum) - exp(-t/taue)) taue g0 / (taum-taue)

We use a very short time constant for the excitatory currents, a longer one for the inhibitory currents, and an even longer one for the membrane potential.

```
from brian import *

taum = 20*ms
taue =  1*ms
taui = 10*ms
Vt   = 10*mV
Vr   =  0*mV

eqs = Equations('''
      dV/dt  = (-V+ge-gi)/taum : volt
      dge/dt = -ge/taue        : volt
      dgi/dt = -gi/taui        : volt
      ''')
```

Connections

As before, we'll have a group of two neurons under direct control, the first of which will be excitatory this time, and the second will be inhibitory. To demonstrate the effect, we'll have two excitatory spikes reasonably close together, followed by an inhibitory spike later on, and then shortly after that two excitatory spikes close together.

```
spiketimes = [(0,1*ms),(0,10*ms),
              (1,40*ms),
              (0,50*ms),(0,55*ms)]

G1 = SpikeGeneratorGroup(2,spiketimes)
G2 = NeuronGroup(N=1, model=eqs, threshold=Vt, reset=Vr)
```

```
C1 = Connection(G1,G2,'ge')
C2 = Connection(G1,G2,'gi')
```

The weights are the same - when we increase `ge` the effect on `V` is excitatory and when we increase `gi` the effect on `V` is inhibitory.

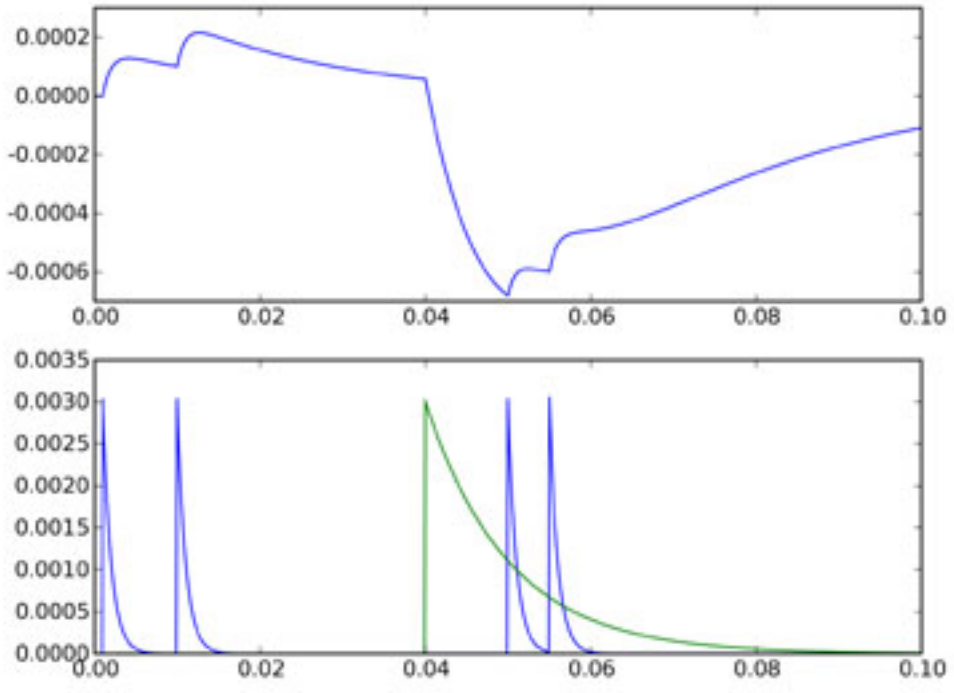
```
C1[0,0] = 3*mV
C2[1,0] = 3*mV
```

We set up monitors and run as normal.

```
Mv  = StateMonitor(G2,'V',record=True)
Mge = StateMonitor(G2,'ge',record=True)
Mgi = StateMonitor(G2,'gi',record=True)

run(100*ms)
```

This time we do something a little bit different when plotting it. We want a plot with two subplots, the top one will show `V`, and the bottom one will show both `ge` and `gi`. We use the `subplot` command from pylab which mimics the same command from Matlab.

```
figure()
subplot(211)
plot(Mv.times,Mv[0])
subplot(212)
plot(Mge.times,Mge[0])
plot(Mgi.times,Mgi[0])
show()
```



The top figure shows the voltage trace, and the bottom figure shows `ge` in blue and `gi` in green. You can see that although the inhibitory and excitatory weights are the same, the inhibitory current is much more powerful. This is because the effect of `ge` or `gi` on `V` is related to the integral of the differential equation for those variables, and `gi` decays much more slowly than `ge`. Thus the size of the negative deflection at 40 ms is much bigger than the excitatory ones, and even the double excitatory spike after the inhibitory one can't cancel it out.

In the next part of this tutorial, we set up our first serious network, with 4000 neurons, excitatory and inhibitory.

Exercises

1. Try changing the parameters and spike times to get a feel for how it works.
2. Try an equivalent implementation with the equation taum dV/dt = -V+ge+gi
3. Verify that the differential equation has been solved correctly.

Solutions

Solution for 2:

Simply use the line `C2[1,0] = -3*mV` to get the same effect.

Solution for 3:

First, set up the situation we described at the top for which we already know the solution of the differential equations, by changing the spike times as follows:

```
spiketimes = [(0,0*ms)]
```

Now we compute what the values ought to be as follows:

```
t = Mv.times
Vpredicted = (exp(-t/taum) - exp(-t/taue))*taue*(3*mV) / (taum-taue)
```

Now we can compute the difference between the predicted and actual values:

```
Vdiff = abs(Vpredicted - Mv[0])
```

This should be zero:

```
print max(Vdiff)
```

Sure enough, it's as close as you can expect on a computer. When I run this it gives me the value 1.3 aV, which is 1.3 * 10^-18 volts, i.e. effectively zero given the finite precision of the calculations involved.

Tutorial 2c: The CUBA network

In this part of the tutorial, we set up our first serious network that actually does something. It implements the CUBA network, Benchmark 2 from:

> Simulation of networks of spiking neurons: A review of tools and strategies (2006). 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

This is a network of 4000 neurons, of which 3200 excitatory, and 800 inhibitory, with exponential synaptic currents. The neurons are randomly connected with probability 0.02.

```
from brian import *

taum =  20*ms          # membrane time constant
taue =   5*ms          # excitatory synaptic time constant
taui =  10*ms          # inhibitory synaptic time constant
Vt   = -50*mV          # spike threshold
Vr   = -60*mV          # reset value
El   = -49*mV          # resting potential
we   = (60*0.27/10)*mV # excitatory synaptic weight
wi   =  (20*4.5/10)*mV # inhibitory synaptic weight

eqs = Equations('''
        dV/dt  = (ge-gi-(V-El))/taum : volt
        dge/dt = -ge/taue            : volt
        dgi/dt = -gi/taui            : volt
        ''')
```

So far, this has been pretty similar to the previous part, the only difference is we have a couple more parameters, and we've added a resting potential `El` into the equation for `V`.

Now we make lots of neurons:

```
G = NeuronGroup(4000, model=eqs, threshold=Vt, reset=Vr)
```

Next, we divide them into subgroups. The `subgroup()` method of a `NeuronGroup` returns a new `NeuronGroup` that can be used in exactly the same way as its parent group. At the moment, the subgrouping mechanism can only be used to create contiguous groups of neurons (so you can't have a subgroup consisting of neurons 0-100 and also 200-300 say). We designate the first 3200 neurons as `Ge` and the second 800 as `Gi`, these will be the excitatory and inhibitory neurons.

```
Ge = G.subgroup(3200) # Excitatory neurons
Gi = G.subgroup(800)  # Inhibitory neurons
```

Now we define the connections. As in the previous part of the tutorial, `ge` is the excitatory current and `gi` is the inhibitory one. `Ce` says that an excitatory neuron can synapse onto any neuron in `G`, be it excitatory or inhibitory. Similarly for inhibitory neurons. We also randomly connect `Ge` and `Gi` to the whole of `G` with probability 0.02 and the weights given in the list of parameters at the top.

```
Ce=Connection(Ge, G, 'ge', sparseness=0.02, weight=we)
Ci=Connection(Gi, G, 'gi', sparseness=0.02, weight=wi)
```

Set up some monitors as usual. The line `record=0` in the `StateMonitor` declarations indicates that we only want to record the activity of neuron 0. This saves time and memory.

```
M   = SpikeMonitor(G)
MV  = StateMonitor(G, 'V', record=0)
Mge = StateMonitor(G, 'ge', record=0)
Mgi = StateMonitor(G, 'gi', record=0)
```

And in order to start the network off in a somewhat more realistic state, we initialise the membrane potentials uniformly randomly between the reset and the threshold.

```
G.V = Vr + (Vt-Vr) * rand(len(G))
```

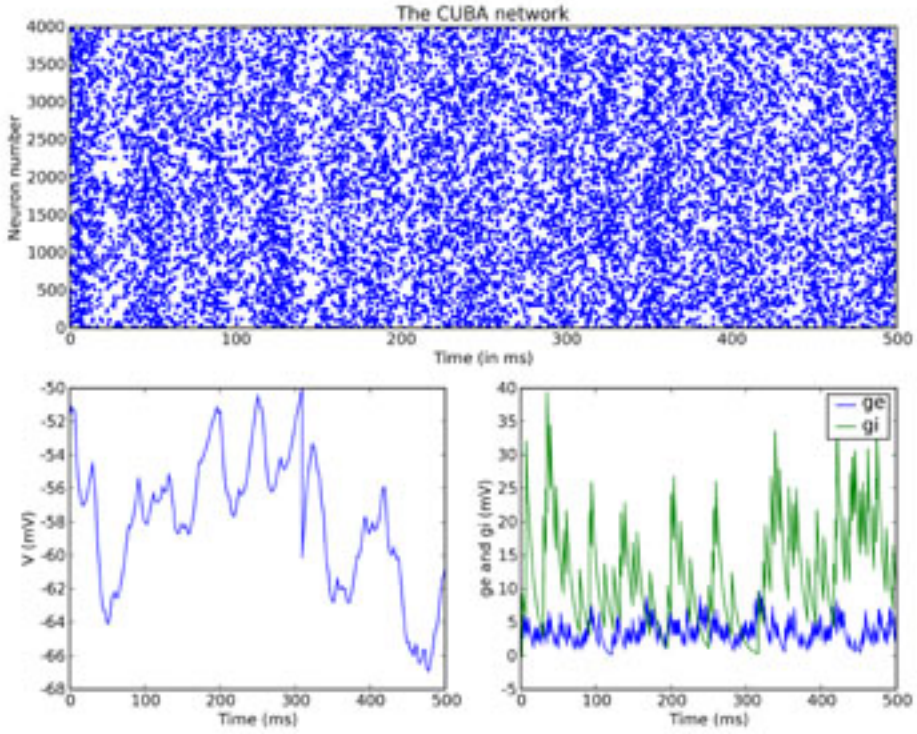
Now we run.

```
run(500*ms)
```

And finally we plot the results. Just for fun, we do a rather more complicated plot than we've been doing so far, with three subplots. The upper one is the raster plot of the whole network, and the lower two are the values of `V` (on the left) and `ge` and `gi` (on the right) for the neuron we recorded from. See the PyLab documentation for an explanation of the plotting functions, but note that the `raster_plot()` keyword `newfigure=False` instructs the (Brian) function `raster_plot()` not to create a new figure (so that it can be placed as a subplot of a larger figure).

```
subplot(211)
raster_plot(M, title='The CUBA network', newfigure=False)
subplot(223)
plot(MV.times/ms, MV[0]/mV)
xlabel('Time (ms)')
ylabel('V (mV)')
subplot(224)
plot(Mge.times/ms, Mge[0]/mV)
plot(Mgi.times/ms, Mgi[0]/mV)
xlabel('Time (ms)')
```

```
ylabel('ge and gi (mV)')
legend(('ge','gi'), 'upper right')
show()
```



CHAPTER

FOUR

USER MANUAL

The SciPy, NumPy and PyLab packages are documented on the following web sites:

- http://www.scipy.org/Getting_Started
- http://www.scipy.org/Documentation
- http://docs.scipy.org/
- http://matplotlib.sourceforge.net/matplotlib.pylab.html

Brian itself is documented in the following sections:

4.1 Units

4.1.1 Basics

Brian has a system for physical quantities with units built in, and most of the library functions require that variables have the right units. This restriction is useful in catching hard to find errors based on using incorrect units, and ensures that simulated models are physically meaningful. For example, running the following code causes an error:

```
>>> from brian import *
>>> c = Clock(t=0)

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    c = Clock(t=0)
  File "C:\Documents and Settings\goodman\Mes documents\Programming\Python simulator\Brian\units.py",
    raise DimensionMismatchError("Function " + f.__name__ + " variable " + k + " should have dimensio
DimensionMismatchError: Function __init__ variable t should have dimensions of s, dimensions were (1)
```

You can see that Brian raises a `DimensionMismatchError` exception, because the `Clock` object expects `t` to have units of time. The correct thing to write is:

```
>>> from brian import *
>>> c = Clock(t=0*second)
```

Similarly, attempting to do numerical operations with inconsistent units will raise an error:

```
>>> from brian import *
>>> 3*second+2*metre

Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    3*second+2*metre
  File "C:\Documents and Settings\goodman\Mes documents\Programming\Python simulator\Brian\units.py",
    if dim==self.dim:
DimensionMismatchError: Addition, dimensions were (s) (m)
```

4.1.2 Units defined in Brian

The following fundamental SI unit names are defined:

`metre`, `meter` (US spelling), `kilogram`, `second`, `amp`, `kelvin`, `mole`, `candle`

These derived SI unit names are also defined:

`radian`, `steradian`, `hertz`, `newton`, `pascal`, `joule`, `watt`, `coulomb`, `volt`, `farad`, `ohm`, `siemens`, `weber`, `tesla`, `henry`, `celsius`, `lumen`, `lux`, `becquerel`, `gray`, `sievert`, `katal`

In addition, you can form scaled versions of these units with any of the standard SI prefixes:

Factor	Name	Symbol	Factor	Name	Symbol
10^24	yotta	Y	10^-24	yocto	y
10^21	zetta	Z	10^-21	zepto	z
10^18	exa	E	10^-21	zepto	z
10^15	peta	P	10^-15	femto	f
10^12	tera	T	10^-12	pico	p
10^9	giga	G	10^-9	nano	n
10^6	mega	M	10^-6	micro	u (mu in SI)
10^3	kilo	k	10^-3	milli	m
10^2	hecto	h	10^-2	centi	c
10^1	deka	da	10^-1	deci	d

So for example, you could write `fnewton` for femto-newtons, `Mwatt` for megawatt, etc.

There are also units for 2nd and 3rd powers of each of the above units, for example `metre3 = metre**3`, `watt2 = watt*watt`, etc.

You can optionally use short names for some units derived from volts, amps, farads, siemens, seconds, hertz and metres: `mV`, `mA`, `uA`, `nA`, `pA`, `mF`, `uF`, `nF`, `mS`, `uS`, `ms`, `Hz`, `kHz`, `MHz`, `cm`, `cm2`, `cm3`, `mm`, `mm2`, `mm3`, `um`, `um2`, `um3`. Since these names are so short, there is a danger that they might clash with your own variables names, so watch out for that.

4.1.3 Arrays and units

Versions of Brian before 1.0 had a system for allowing arrays to have units, this has been removed for the 1.0 release because of stability problems - as new releases of NumPy, SciPy and PyLab came out it required changes to the units code. Now all arrays used by Brian are standard NumPy arrays and have no units.

4.1.4 Checking units

Units are automatically checked when arithmetic operations are performed, and when a neuron group is initialised (the consistency of the differential equations is checked). They can also be checked explictly when a user-defined function is called by using the decorator `@check_units`, which can be used as follows:

```
@check_units(I=amp,R=ohm,wibble=metre,result=volt)
def getvoltage(I,R,**k):
    return I*R
```

Remarks:

- not all arguments need to be checked
- keyword arguments may be checked
- the result can optionnally be checked
- no error is raised if the values are strings.

4.1.5 Disabling units

Unit checking can slow down the simulations. The units system can be disabled by inserting `import brian_no_units` as the *first line* of the script, e.g.:

```
import brian_no_units
from brian import *
# etc
```

Internally, physical quantities are floats with an additional units information. The float value is the value in the SI system. For example, `float(mV)` returns `0.001`. After importing `brian_no_units`, all units are converted to their float values. For example, `mV` is simply the number `0.001`. This may also be a solution when using external libraries which are not compatible with units (but see next section).

Unit checking can also be turned down locally when initializing a neuron group by passing the argument `check_units=False`. In that case, no error is raised if the differential equations are not homogeneous.

A good practice is to develop the script with units on, then switch them off once the script runs correctly.

4.1.6 Converting quantities

In many situations, physical quantities need to be expressed with given units. For example, one might want to plot a graph of the membrane potential in mV as a function of time in ms. The following code:

```
plot(t,V)
```

displays the trace with time in seconds and potential in volts. The simplest solution to have time in ms and potential in mV is to use units operations:

```
plot(t/ms,V/mV)
```

Here, t/ms is a unitless array containing the values of t in ms. The same trick may be applied to use external functions which do not work with units (convert the arguments to unitless quantities as above).

4.2 Models and neuron groups

4.2.1 Equations

`Equations` objects are initialised with a string as follows:

```
eqs=Equations('''
dx/dt=(y-x)/tau + a : volt    # differential equation
y=2*x : volt                  # equation
z=x                           # alias
a : volt/second               # parameter
''')
```

It is possible to pass a string instead of an `Equations` object when initialising a neuron group. In that case, the string is implicitly converted to an `Equations` object. There are 4 different types of equations:

- Differential equations: a differential equation, also defining the variable as a state variable in neuron groups.
- Equations: a non-differential equation, which is useful for defining complicated models. The variables are also accessible for reading in neuron groups, which is useful for monitoring. The graph of dependencies of all equations must have no cycle.
- Aliases: the two variables are equivalent. This is implemented as an equation, with write access in neuron groups.
- Parameters: these are constant variables, but their values can differ from one neuron to the next. They are implemented internally as differential equations with zero derivative.

Right hand sides must be valid Python expressions, possibly including comments and multiline characters (\).

The units of all variables except aliases must be specified. Note that in first line, the units *volt* are meant for x, not dx/dt. The consistency of all units is checked with the method `check_units()`, which is automatically called when initialising a neuron group (through the method `prepare()`).

When an `Equations` object is finalised (through the method `prepare()`, automatically called the `NeuronGroup` initialiser), the names of variables defined by non-differential equations are replaced by their (string) values, so that differential equations are self-consistent. In the process, names of external variables are also modified to avoid conflicts (by adding a prefix).

4.2.2 Neuron groups

The key idea for efficient simulations is to update synchronously the state variables of all identical neuron models. A neuron group is defined by the model equations, and optionally a threshold condition and a reset. For example for 100 neurons:

```
eqs=Equations('dv/dt=-v/tau : volt')
group=NeuronGroup(100,model=eqs,reset=0*mV,threshold=10*mV)
```

The `model` keyword also accepts strings (in that case it is converted to an `Equations` object), e.g.:

```
group=NeuronGroup(100,model='dv/dt=-v/tau : volt',reset=0*mV,threshold=10*mV)
```

The units of both the reset and threshold are checked for consistency with the equations. The code above defines a group of 100 integrate-and-fire neurons with threshold 10 mV and reset 0 mV. The second line defines an object named `group` which contains all the state variables, which can be accessed with the dot notation, i.e. `group.v` is a vector

with the values of variable `v` for all of the 100 neurons. It is an array with units as defined in the equations (here, volt). By default, all state variables are initialised at value 0. It can be initialised by the user as in the following example:

```
group.v=linspace(0*mV,10*mV,100)
```

Here the values of `v` for all the neurons are evenly spaced between 0 mV and 10 mV (`linspace` is a NumPy function). The method `group.rest()` may also be used to set the resting point of the equations, but convergence is not always guaranteed.

Important options

- `refractory`: a refractory period (default 0 ms), to be used in combination with the `reset` value.
- `implicit` (default `False`): if True, then an implicit method is used. This is useful for Hodgkin-Huxley equations, which are stiff.

Subgroups

Subgroups can be created with the slice operator:

```
subgroup1=group[0:50]
subgroup2=group[50:100]
```

Then `subgroup2.v[i]` equals `group.v[50+i]`. An alternative equivalent method is the following:

```
subgroup1=group.subgroup(50)
subgroup2=group.subgroup(50)
```

The parent group keeps track of the allocated subgroups. But note that the two methods are mutually exclusive, e.g. in the following example:

```
subgroup1=group[0:50]
subgroup2=group.subgroup(50)
```

both subgroups are actually identical.

Subgroups are useful when creating connections or monitoring the state variables or spikes. The best practice is to define groups as large as possible, then divide them in subgroups if necessary. Indeed, the larger the groups are, the faster the simulation runs. For example, for a network with a feedforward architecture, one should first define one group holding all the neurons in the network, then define the layers as subgroups of this big group.

Details

For details, see the reference documentation for `NeuronGroup`.

4.2.3 Reset

More complex resets can be defined. The value of the `reset` keyword can be:

- a quantity (`0*mV`)
- a string

- a function
- a `Reset` object, which can be used for resetting a specific state variable or for resetting a state variable to the value of another variable.

Reset as Python code

The simplest way to customise the reset is to define it as a Python statement, e.g.:

```
eqs='''
dv/dt=-v/tau : volt
dw/dt=-w/tau : volt
'''
group=NeuronGroup(100,model=eqs,reset="v=0*mV;w+=3*mV",threshold=10*mV)
```

The string must be a valid Python statement (possibly a multiline string). It can contain variables from the neuron group, units and any variable defined in the namespace (e.g. tau), as for equations. Be aware that if a variable in the namespace has the same name as a neuron group variable, then it masks the neuron variable. The way it works is that the code is evaluated with each neuron variable `v` replaced by `v[spikes]`, where `spikes` is the array of indexes of the neurons that just spiked.

Functional reset

To define a specific reset, the generic method is define a function as follows:

```
def myreset(P,spikes):
  P.v[spikes]=rand(len(spikes))*5*mV
group=NeuronGroup(100,model=eqs,reset=myreset,threshold=10*mV)
```

or faster:

```
def myreset(P,spikes):
  P.v_[spikes]=rand(len(spikes))*5*mV
```

Every time step, the user-defined function is called with arguments `P`, the neuron group, and `spikes`, the list of indexes of the neurons that just spiked. The function above resets the neurons that just spiked to a random value.

Resetting another variable

It is possible to specify the reset variable explicitly:

```
group=NeuronGroup(100,model=eqs,reset=Reset(0*mV,state='w'),threshold=10*mV)
```

Here the variable `w` is reset.

Resetting to the value of another variable

The value of the reset can be given by another state variable:

```
group=NeuronGroup(100,model=eqs,reset=VariableReset(0*mV,state='v',resetvaluestate='w'),threshold=10*
```

Here the value of the variable `w` is used to reset the variable `v`.

4.2.4 Threshold

As for the reset, the threshold can be customised.

Threshold as Python expression

The simplest way to customise the threshold is to define it as a Python expression, e.g.:

```
eqs='''
dv/dt=-v/tau : volt
dw/dt=(v-w)/tau : volt
'''
group=NeuronGroup(100,model=eqs,reset=0*mV,threshold="v>=w")
```

The string must be an expression which evaluates to a boolean. It can contain variables from the neuron group, units and any variable defined in the namespace (e.g. tau), as for equations. Be aware that if a variable in the namespace has the same name as a neuron group variable, then it masks the neuron variable. The way it works is that the expression is evaluated with the neuron variables replaced by their vector values (values for all neurons), so that the expression returns a boolean vector.

Functional threshold

The generic method to define a custom threshold condition is to pass a function of the state variables which returns a boolean (true if the threshold condition is met), for example:

```
eqs='''
dv/dt=-v/tau : volt
dw/dt=(v-w)/tau : volt
'''
group=NeuronGroup(100,model=eqs,reset=0*mV,threshold=lambda v,w:v>=w)
```

Here we used an anonymous function (`lambda` keyword) but of course a named function can also be used. In this example, spikes are generated when v is greater than w. Note that the arguments of the function must be the state variables with the same order as in the `Equations` string.

Thresholding another variable

It is possible to specify the threshold variable explicitly:

```
group=NeuronGroup(100,model=eqs,reset=0*mV,threshold=Threshold(0*mV,state='w'))
```

Here the variable `w` is checked.

Using another variable as the threshold value

The same model as in the functional threshold example can be defined as follows:

```
group=NeuronGroup(100,model=eqs,reset=0*mV,threshold=\
      VariableThreshold(state='v',threshold_state='w'))
```

Empirical threshold

For Hodgkin-Huxley models, one needs to determine the threshold empirically. Here the *threshold* should really be understood rather as the onset of the spikes (used to propagate the spikes to the other neurons), since there is no explicit reset. There is a `Threshold` subclass for this purpose:

```
group=NeuronGroup(100,model=eqs,threshold=EmpiricalThreshold(threshold=-20*mV,refractory=3*ms))
```

Spikes are triggered when the membrane potential reaches the value -20 mV, but only if it has not spiked in the last 3 ms (otherwise there would be spikes every time step during the action potential). The `state` keyword may be used to specify the state variable which should be checked for the threshold condition.

Poisson threshold

It is possible to generate spikes with a given probability rather than when a threshold condition is met, by using the class `PoissonThreshold`, as in the following example:

```
group=NeuronGroup(100,model='x : Hz',threshold=PoissonThreshold(state='x'))
x=linspace(0*Hz,10*Hz,100)
```

Here spikes are generated as Poisson processes with rates given by the variable x (the `state` keyword is optional: default = first variable defined). Note that x can change over time (inhomogeneous Poisson processes). The units of variable x must be Hertz.

4.3 Connections

4.3.1 Building connections

First, one must define which neuron groups are connected and which state variable receives the spikes. The following instruction:

```
myconnection=Connection(group1,group2,'ge')
```

defines a connection from group `group1` to `group2`, acting on variable `ge`. When neurons from group `group1` spike, the variable `ge` of the target neurons in group `group2` are incremented. When the connection object is initialised, the list of connections is empty. It can be created in several ways. First, explicitly:

```
myconnection[2,5]=3*nS
```

This instruction connects neuron 2 from `group1` to neuron 5 from `group2` with synaptic weight 3 nS. Units should match the units of the variable defined at initialisation time (`ge`).

The matrix of synaptic weights can be defined directly with the method `Connection.connect()`:

```
W=rand(len(group1),len(group2))*nS
myconnection.connect(group1,group2,W)
```

Here a matrix with random elements is used to define the synaptic weights from `group1` to `group2`. It is possible to build the matrix by block by using subgroups, e.g.:

```
W=rand(20,30)*nS
myconnection.connect(group1[0:20],group2[10:40],W=W)
```

There are several handy functions available to set the synaptic weights: `connect_full()`, `connect_random()` and `connect_one_to_one()`. The first one is used to set uniform weights for all pairs of neurons in the (sub)groups:

```
myconnection.connect_full(group1[0:20],group2[10:40],weight=5*nS)
```

The second one is used to set uniform weights for random pairs of neurons in the (sub)groups:

```
myconnection.connect_random(group1[0:20],group2[10:40],sparseness=0.02,weight=5*nS)
```

Here the third argument (0.02) is the probability that a synaptic connection exists between two neurons. The number of presynaptic neurons can be made constant by setting the keyword `fixed=True` (probability * number of neurons in `group1`). Finally, the method `connect_one_to_one()` connects neuron i from the first group to neuron i from the second group:

```
myconnection.connect_one_to_one(group1,group2,weight=3*nS)
```

Both groups must have the same number of neurons.

If you are connecting the whole groups, you can omit the first two arguments, e.g.:

```
myconnection.connect_full(weight=5*nS)
```

connects `group1` to `group2` with weights 5 nS.

Building connections with connectivity functions

There is a simple and efficient way to build heterogeneous connections, by passing functions instead of constants to the methods `connect_full()` and `connect_random()`. The function must return the synaptic weight for a given pair of neuron (i,j). For example:

```
myconnection.connect_full(group1,group2,weight=lambda i,j:(1+cos(i-j))*2*nS)
```

where i (j) indexes neurons in `group1` (`group2`). This is the same as doing by hand:

```
for i in range(len(group1)):
  for j in range(len(group2)):
    myconnection[i,j]=(1+cos(i-j))*2*nS
```

but it is much faster because the construction is vectorised, i.e., the function is called for every i with j being the entire row of target indexes. Thus, the implementation is closer to:

```
for i in range(len(group1)):
    myconnection[i,j]=(1+cos(i-arange(len(group2)))*2*nS
```

The method `connect_random()` also accepts functional arguments for the weights (not the connection probability yet). For that method, it is possible to pass a function with no argument, as in the following example:

```
myconnection.connect_random(group1,group2,0.1,weight=lambda:rand()*nS)
```

Here each synaptic weight is random (between 0 and 1 nS).

4.3.2 Delays

Transmission delays can be introduced with the keyword `delay`, passed at initialisation time. There are two types of delays, homogeneous (all synapses have the same delay) and heterogeneous (all synapses can have different delays). The former is more computationally efficient than the latter. An example of homogeneous delays:

```
myconnection=Connection(group1,group2,'ge',delay=3*ms)
```

If you have limited heterogeneity, you can use several Connection objects, e.g.:

```
myconnection_fast=Connection(group1,group2,'ge',delay=1*ms)
myconnection_slow=Connection(group1,group2,'ge',delay=5*ms)
```

For a highly heterogeneous set of delays, initialise the connection with `delay=True`, set a maximum delay with for example `max_delay=5*ms` and then use the `delay` keyword in the `connect_random()` and `connect_full()` methods:

```
myconnection=Connection(group1,group2,'ge',delay=True,max_delay=5*ms)
myconnection.connect_full(group1,group2,weight=3*nS,delay=(0*ms,5*ms))
```

The code above initialises the delays uniformly randomly between 0ms and 5ms. You can also set `delay` to be a function of no variables, where it will be called once for each synapse, or of two variables `i, j` where it will be called once for each row, as in the case of the weights in the section above.

Alternatively, you can set the delays as follows:

```
myconnection.delay[i,j] = 3*ms
```

See the reference documentation for `Connection` and `DelayConnection` for more details.

4.3.3 Connection structure

The underlying data structure used to store the synaptic connections is by default a sparse matrix. If the connections are dense, it is more efficient to use a dense matrix, which can be set at initialisation time:

```
myconnection=Connection(group1,group2,'ge',structure='dense')
```

The sparse matrix structure is fixed during a run, new synapses cannot be added or deleted, but there is a dynamic sparse matrix structure. It is less computationally efficient, but allows runtime adding and deleting of synaptic connections. Use the `structure='dynamic'` keyword. For more details see the reference documentation for `Connection`.

4.3.4 Modulation

The synaptic weights can be modulated by a state variable of the presynaptic neurons with the keyword `modulation`:

```
myconnection=Connection(group1,group2,'ge',modulation='u')
```

When a spike is produced by a presynaptic neuron (`group1`), the variable ge of each postsynaptic neuron (`group2`) is incremented by the synaptic weight multiplied by the value of the variable u of the presynaptic neuron. This is useful to implement short-term plasticity.

4.3.5 Direct connection

In some cases, it is useful to connect a group directly to another one, in a one-to-one fashion. The most efficient way to implement it is with the class `IdentityConnection`:

```
myconnection=IdentityConnection(group1,group2,'ge',weight=1*nS)
```

With this structure, the synaptic weights are homogeneous (it is not possible to define them independently). When neuron i from `group1` spikes, the variable ge of neuron i from `group2` is increased by 1 nS. A typical application is when defining inputs to a network.

4.3.6 Simple connections

If your connection just connects one group to another in a simple way, you can initialise the weights and delays at the time you initialise the `Connection` object by using the `weight`, `sparseness` and `delay` keywords. For example:

```
myconnection = Connection(group1, group2, 'ge', weight=1*nS, sparseness=0.1,
                          delay=(0*ms, 5*ms), max_delay=5*ms)
```

This would be equivalent to:

```
myconnection = Connection(group1, group2, 'ge', delay=True, max_delay=5*ms)
myconnection.connect_random(group1, group2, weight=1*nS, delay=(0*ms, 5*ms))
```

If the `sparseness` value is omitted or set to value 1, full connectivity is assumed, otherwise random connectivity.

NOTE: in this case the `delay` keyword used without the `weight` keyword has no effect.

4.4 Spike-timing-dependent plasticity

Synaptic weights can be modified by spiking activity. Weight modifications at a given synapse depend on the relative timing between presynaptic and postsynaptic spikes. Down to the biophysical level, there is a number of synaptic variables which are continuously evolving according to some differential equations, and those variables can be modified by presynaptic and postsynaptic spikes. In spike-timing-dependent plasticity (STDP) rules, the synaptic weight changes at the times of presynaptic and postsynaptic spikes only, as a function of the other synaptic variables. In Brian, an STDP rule can be specified by defining an `STDP` object, as in the following example:

```
eqs_stdp='''
dA_pre/dt=-A_pre/tau_pre : 1
dA_post/dt=-A_post/tau_post : 1
'''
stdp=STDP(myconnection,eqs=eqs_stdp,pre='A_pre+=dA_pre;w+=A_post',
          post='A_post+=dA_post;w+=A_pre',wmax=gmax)
```

The `STDP` object acts on the `Connection` object `myconnection`. Equations of the synaptic variables are given in a string (argument `eqs`) as for defining neuron models. When a presynaptic (postsynaptic) spike is received, the code `pre` (`post`) is executed, where the special identifier `w` stands for the synaptic weight (from the specified connection matrix). Optionally, an upper limit can be specified for the synaptic weights (`wmax`).

The example above defines an exponential STDP rule with hard bounds and all-to-all pair interactions.

4.4.1 Current limitations

- The differential equations must be linear.
- Presynaptic and postsynaptic variables must not interact, that is, a variable cannot be modified by both presynaptic and postsynaptic spikes. However, synaptic weight modifications can depend on all variables.
- STDP currently works only with homogeneous delays, not heterogeneous ones.

Exponential STDP

In many applications, the STDP function is piecewise exponential. In that case, one can use the `ExponentialSTDP` class:

```
stdp=ExponentialSTDP(synapses,taup,taum,Ap,Am,wmax=gmax,interactions='all',update='additive')
```

Here the synaptic weight modification function is:

```
f(s) = Ap*exp(-s/taup) if s >0
       Am*exp(s/taum) if s <0
```

where s is the time of the postsynaptic spike minus the time of the presynaptic spike. The modification is generally relative to the maximum weight `wmax` (see below). The `interactions` keyword determines how pairs of pre/post synaptic spikes interact: `all` if contributions from all pairs are added, `nearest` for only nearest neighbour interactions, `nearest_pre` if only the nearest presynaptic spike and all postsynaptic spikes are taken into account and `nearest_post` for the symmetrical situation. The weight update can be `additive`, i.e., w=w+wmax*f(s), or `multiplicative`: w=w+w*f(s) for depression (usually s<0) and w=w+(wmax-w)*f(s) for potentiation (usually s>0). It can also be `mixed`: multiplicative for depression, additive for potentiation.

Delays

By default, transmission delays are assumed to be axonal, i.e., synapses are located on the soma: if the delay of the connection C is d, then presynaptic spikes act after a delay d while postsynaptic spikes act immediately. This behaviour can be overriden with the keywords `delay_pre` and `delay_post`, in both classes `STDP` and `ExponentialSTDP`.

4.5 Short-term plasticity

Brian implements the short-term plasticity model described in: Markram et al (1998). Differential signaling via the same axon of neocortical pyramidal neurons, PNAS 95(9):5323-8. Synaptic dynamics is described by two variables x and u, which follows the following differential equations:

```
dx/dt=(1-x)/taud  (depression)
du/dt=(U-u)/tauf  (facilitation)
```

where taud, tauf are time constants and U is a parameter in 0..1. Each a presynaptic spike triggers modifications of the variables:

```
u->u+U*(1-u)
x->x*(1-u)
```

Note that the update order is important. Synaptic weights are modulated by the product u*x (in 0..1), which is taken before updating the variables. This model describes both depression and facilitation.

To introduce short-term plasticity into an existing connection C, use the class `STP`:

```
mystp=STP(C,taud=100*ms,tauf=5*ms,U=.6)
```

4.6 Recording

The activity of the network can be recorded by defining *monitors*.

4.6.1 Recording spikes

To record the spikes from a given group, define a `SpikeMonitor` object:

```
M=SpikeMonitor(group)
```

At the end of the simulation, the spike times are stored in the variable `spikes` as a list of pairs (i,t) where neuron i fired at time t. For example, the following code extracts the list of spike times for neuron 3:

```
spikes3=[t for i,t in M.spikes if i==3]
```

but this operation can be done directly as follows:

```
spikes3=M[3]
```

The total number of spikes is `M.nspikes`.

Custom monitoring

To process the spikes in a specific way, one can pass a function at initialisation of the `SpikeMonitor` object:

```
def f(spikes):
  print spikes

M=SpikeMonitor(group,function=f)
```

The function `f` is called every time step with the argument `spikes` being the list of indexes of neurons that just spiked.

4.6.2 Recording state variables

State variables can be recorded continuously by defining a `StateMonitor` object, as follows:

```
M=StateMonitor(group,'v')
```

Here the state variables `v` of the defined group are monitored. By default, only the statistics are recorded. The list of time averages for all neurons is `M.mean`; the standard deviations are stored in `M.std` and the variances in `M.var`. Note that these are averages over time, not over the neurons.

To record the values of the state variables over the whole simulation, use the keyword `record`:

```
M1=StateMonitor(group,'v',record=True)
M2=StateMonitor(group,'v',record=[3,5,9])
```

The first monitor records the value of `v` for all neurons while the second one records `v` for neurons 3, 5 and 9 only. The list of times is stored in `M1.times` and the lists of values are stored in `M1[i]`, where i the index of the neuron. By default, the values of the state variables are recorded every timestep, but one may record every n timesteps by setting the keyword `timestep`:

```
M=StateMonitor(group,'v',record=True,timestep=n)
```

Recording spike triggered state values

You can record the value of a state variable at each spike using `StateSpikeMonitor`:

```
M = StateSpikeMonitor(group, 'V')
```

The `spikes` attribute of `M` consists of a series of tuples `(i,t,V)` where `V` is the value at the time of the spike.

Recording multiple state variables

You can either use multiple `StateMonitor` objects or use the `MultiStateMonitor` object:

```
M = MultiStateMonitor(group, record=True)
...
run(...)
...
plot(M['V'].times, M['V'][0])
figure()
for name, m in M.iteritems():
    plot(m.times, m[0], label=name)
legend()
show()
```

4.6.3 Counting spikes

To count the total number of spikes produced by a group, use a `PopulationSpikeCounter` object:

```
M=PopulationSpikeCounter(group)
```

Then the number of spikes after the simulation is `M.nspikes`. If you need to count the spikes separately for each neuron, use a `SpikeCounter` object:

```
M=SpikeCounter(group)
```

Then `M[i]` is the number of spikes produced by neuron i.

4.6.4 Recording population rates

The population rate can be monitored with a `PopulationRateMonitor` object:

```
M=PopulationRateMonitor(group)
```

After the simulation, `M.times` contains the list of recording times and `M.rate` is the list of rate values (where the rate is meant in the spatial sense: average rate over the whole group at some given time). The bin size is set with the `bin` keyword (in seconds):

```
M=PopulationRateMonitor(group,bin=1*ms)
```

Here the averages are calculated over 1 ms time windows. Alternatively, one can use the `smooth_rate()` method to smooth the rates:

```
rates=M.smooth_rate(width=1*ms,filter='gaussian')
```

The rates are convolved with a linear filter, which is either a Gaussian function (`gaussian`, default) or a box function ('flat').

4.7 Inputs

Some specific types of neuron groups are available to provide inputs to a network.

4.7.1 Poisson inputs

Poisson spike trains can be generated as follows:

```
group=PoissonGroup(100,rates=10*Hz)
```

Here 100 neurons are defined, which emit spikes independently according to Poisson processes with rates 10 Hz. To have different rates across the group, initialise with an array of rates:

```
group=PoissonGroup(100,rates=linspace(0*Hz,10*Hz,100))
```

Inhomogeneous Poisson processes can be defined by passing a function of time that returns the rates:

```
group=PoissonGroup(100,rates=lambda t:(1+cos(t))*10*Hz)
```

or:

```
r0=linspace(0*Hz,10*Hz,100)
group=PoissonGroup(100,rates=lambda t:(1+cos(t))*r0)
```

4.7.2 Correlated inputs

Generation of correlated spike trains is partially implemented, using algorithms from the the following paper: Brette, R. (2009) Generation of correlated spike trains, Neural Computation 21(1): 188-215. Currently, only the method with Cox processes (or doubly stochastic processes, first method in the paper) is fully implemented.

Doubly stochastic processes

To generate correlated spike trains with identical rates and homogeneous exponential correlations, use the class `HomogeneousCorrelatedSpikeTrains`:

```
group=HomogeneousCorrelatedSpikeTrains(100,r=10*Hz,c=0.1,tauc=10*ms)
```

where `r` is the rate, `c` is the total correlation strength and `tauc` is the correlation time constant. The cross-covariance functions are *(c*r/tauc)*exp(-|s|/tauc)*. To generate correlated spike trains with arbitrary rates r(i) and cross-covariance functions *c(i,j)*exp(-|s|/tauc)*, use the class `CorrelatedSpikeTrains`:

```
group=CorrelatedSpikeTrains(rates,C,tauc)
```

where `rates` is the vector of rates r(i), `C` is the correlation matrix (which must be symmetrical) and `tauc` is the correlation time constant. Note that distortions are introduced with strong correlations and short correlation time constants. For short time constants, the mixture method is more appropriate (see the paper above). The two classes `HomogeneousCorrelatedSpikeTrains` and `CorrelatedSpikeTrains` define neuron groups, which can be directly used with `Connection` objects.

Mixture method

The mixture method to generate correlated spike trains is only partially implemented and the interface may change in future releases. Currently, one can use the function `mixture_process()` to generate spike trains:

```
spiketrains=mixture_process(nu,P,tauc,t)
```

where `nu` is the vector of rates of the source spike trains, `P` is the mixture matrix (entries between 0 and 1), `tauc` is the correlation time constant, `t` is the duration. It returns a list of (neuron_number,spike_time), which can be passed to `SpikeGeneratorGroup`. This method is appropriate for short time constants and is explained in the paper mentioned above.

4.7.3 Input spike trains

A set of spike trains can be explicitly defined as list of pairs (i,t) (meaning neuron i fires at time t), which used to initialise a `SpikeGeneratorGroup`:

```
spiketimes=[(0,1*ms), (1,2*ms)]
input=SpikeGeneratorGroup(5,spiketimes)
```

The neuron 0 fires at time 1 ms and neuron 1 fires at time 2 ms (there are 5 neurons, but 3 of them never spike). One may also pass a generator instead of a list (in that case the pairs should be ordered in time).

Spike times may also be provided separately for each neuron, using the `MultipleSpikeGeneratorGroup` class:

```
S0=[1*ms, 2*ms]
S1=[3*ms]
S2=[1*ms, 3*ms, 5*ms]
input=MultipleSpikeGeneratorGroup([S0,S1,S2])
```

The object is initialised with a list of spike containers, one for each neuron. Each container can be a sorted list of spike times or any iterable object returning the spike times (ordered in time).

Gaussian spike packets

There is a subclass of `SpikeGeneratorGroup` for generating spikes with a Gaussian distribution:

```
input=PulsePacket(t=10*ms,n=10,sigma=3*ms)
```

Here 10 spikes are produced, with spike times distributed according a Gaussian distribution with mean 10 ms and standard deviation 3 ms.

4.7.4 Direct input

Inputs may also be defined by accessing directly the state variables of a neuron group. The standard way to do this is to insert parameters in the equations:

```
eqs='''
dv/dt=(I-x)/tau : volt
I : volt
'''
group=NeuronGroup(100,model=eqs,reset=0*mV,threshold=15*mV)
group.I=linspace(0*mV,20*mV,100)
```

Here the value of the parameter I for each neuron is provided at initialisation time (evenly distributed between 0 mV and 20 mV). It is possible to change the value of I every timestep by using a user-defined operation (see next section).

4.8 User-defined operations

In addition to neuron models, the user can provide functions that are to be called every timestep during the simulation, using the decorator `network_operation()`:

```
@network_operation
def myoperation():
    do_something_every_timestep()
```

The operation may be called at regular intervals by defining a clock:

```
myclock=Clock(dt=1*ms)

@network_operation(myclock)
def myoperation():
    do_something_every_ms()
```

4.9 Analysis and plotting

Most plotting should be done with the PyLab commands, all of which are loaded when you import Brian. See:

> http://matplotlib.sourceforge.net/matplotlib.pylab.html

for help on PyLab. The scientific library Scipy is also automatically imported by the instruction `from brian import *`.

The most useful plotting instruction is the Pylab function `plot`. A typical use with Brian is:

```
plot(t/ms,vm/mV)
```

where t is a vector of times with units ms and vm is a vector of voltage values with units mV. To display the figures on the screen, the function `show()` must be called once (this should be the last line of your script), except when using IPython with the Pylab mode (`ipython -pylab`).

Brian currently defines just two plotting functions of its own, `raster_plot()` and `hist_plot()`.

4.9.1 Raster plots

Spike trains recorded by a `SpikeMonitor` can be displayed as raster plots:

```
S=SpikeMonitor(group)
...
raster_plot(S)
```

Usual options of the `plot` command can also be passed to `raster_plot()`. One may also pass several spike monitors as arguments.

4.9.2 Statistics

Here are a few functions to analyse first and second order statistical properties of spike trains, defined as ordered lists of spike times:

- Firing rate: `firing_rate(spikes)` where `spikes` is a spike train (list of spike times).
- Coefficient of variation: `CV(spikes)`.
- Cross-correlogram: `correlogram(T1,T2,width=20*ms,bin=1*ms,T=None)` returns the cross-correlogram of spike trains T1 and T2 with lag in [-width,width] and given bin size. T is the total duration (optional) and should be greater than the duration of T1 and T2. The result the rate of coincidences in each bin, returned as an array.
- Autocorrelogram: `autocorrelogram(T0,width=20*ms,bin=1*ms,T=None)` is the same as `correlogram(T0,T0,width=20*ms,bin=1*ms,T=None)`.
- Cross-correlation function: `CCF(T1,T2,width=20*ms,bin=1*ms,T=None)` returns the cross-correlation function of T1 and T2, which is the same as the cross-correlogram divided by the bin size (which makes the result independent of the bin size).
- Autocorrelation function: `ACF(T0,width=20*ms,bin=1*ms,T=None)`, same as `CCF(T0,T0,width=20*ms,bin=1*ms,T=None)`.
- Cross-covariance function: `CCVF(T1,T2,width=20*ms,bin=1*ms,T=None)` is the cross-correlation function of T1 and T2 minus for the cross-correlation of independent spike trains with the same rates (product of rates).
- Auto-covariance function: `ACVF(T0,width=20*ms,bin=1*ms,T=None)` is the same as `CCVF(T0,T0,width=20*ms,bin=1*ms,T=None)`.
- Total correlation coefficient: `total_correlation(T1,T2,width=20*ms,T=None)` is the integral of the cross-covariance function divided by the rate of T1, typically (but not always) between 0 and 1.

4.10 Clocks

Brian is a clock-based simulator: operations are done synchronously at each tick of a clock.

Many Brian objects store a clock object, passed in the initialiser with the optional keyword `clock`. For example, to simulate a neuron group with time step dt=1 ms:

```
myclock=Clock(dt=1*ms)
group=NeuronGroup(100,model='dx/dt=1*mV/ms : volt',clock=myclock)
```

If no clock is specified, the program uses the global default clock. When Brian is initially imported, this is the object `defaultclock`, and it has a default time step of 0.1 ms. In a simple script, you can override this by writing (for example):

```
defaultclock.dt = 1*ms
```

You may wish to use multiple clocks in your program. In this case, for each object which requires one, you have to pass a copy of its `Clock` object. The network run function automatically handles objects with different clocks, updating them all at the appropriate time according to their time steps (value of `dt`).

Multiple clocks can be useful, for example, for defining a simulation that runs with a very small `dt`, but with some computationally expensive operation running at a lower frequency. In the following example, the model is simulated with dt=0.01 ms and the variable x is recorded every ms:

```
simulation_clock=Clock(dt=0.01*ms)
record_clock=Clock(dt=1*ms)
group=NeuronGroup(100,model='dx/dt=-x/tau : volt',clock=simulation_clock)
M=StateMonitor(group,'x',record='True',clock=record_clock)
```

The current time of a clock is stored in the attribute `t` (`simulation_clock.t`) and the timestep is stored in the attribute `dt`.

4.11 Simulation control

4.11.1 The update schedule

When a simulation is run, the operations are done in the following order by default:

1. Update every `NeuronGroup`, this typically performs an integration time step for the differential equations defining the neuron model.
2. Check the threshold condition and propagate the spikes to the target neurons.
3. Reset all neurons that spiked.
4. Call all user-defined operations and state monitors.

The user-defined operations and state monitors can be placed at other places in this schedule, by using the keyword `when`. The values can be `start`, `before_groups`, `after_groups`, `middle`, `before_connections`, `after_connections`, `before_resets`, `after_resets` or `end` (default: end). For example, to call a function `f` at the beginning of every timestep:

```
@network_operation(when='start')
def f():
  do_something()
```

or to record the value of a state variable just before the resets:

```
M=StateMonitor(group,'x',record=True,when='before_resets')
```

4.11.2 Basic simulation control

The simulation is run simply as follows:

```
run(1000*ms)
```

where 1000 ms is the duration of the run. It can be stopped during the simulation with the instruction `stop()`, and the network can be reinitialised with the instruction `reinit()`.

When the `run()` function is called, Brian looks for all relevant objects in the namespace (groups, connections, monitors, user operations), and runs them. In complex scripts, the user might want to run only selected objects. In that case, a `Network` object needs to be created.

Users of `ipython` may also want to make use of the `clear()` function which removes all Brian objects and deletes their data. This is useful because `ipython` keeps persistent references to these objects which stops memory from being freed.

4.11.3 The Network class

A `Network` object holds a collection of objets that can be run, i.e., objects with class `NeuronGroup`, `Connection`, `SpikeMonitor`, `StateMonitor` (or subclasses) or any user-defined operation with the decorator `network_operation()`. Thoses objects can then be simulated. Example:

```
G = NeuronGroup(...)
C = Connection(...)
net = Network(G,C)
net.run(1*second)
```

You can also pass lists of objects. The simulation can be controlled with the methods `stop` and `reinit`.

4.11.4 The `MagicNetwork` object

When `run()`, `reinit()` and `stop()` are called, they act on the "magic network" (the network consisting of all relevant objects such as groups, connections, monitors and user operations). This "magic network" can be explicitly constructed using the `MagicNetwork` object:

```
G = NeuronGroup(...)
C = Connection(...)
net = MagicNetwork()
net.run(1*second)
```

4.12 More on equations

The `Equations` class is a central part of Brian, since models are generally specified with an `Equations` object. Here we explain advanced aspects of this class.

4.12.1 External variables

Equations may contain external variables. When an `Equations` object is initialised, a dictionary is built with the values of all external variables. These values are taken from the namespace where the `Equations` object was defined. It is possible to go one or several levels up in the namespaces by specifying the keyword `level` (default=0). The value of these parameters can in general be changed during the simulation and the modifications are taken into account, except in two situations: when the equations are frozen (see below) or when the integration is exact (linear equations). In those cases, the values of the parameters are the ones at initialisation time.

Alternatively, the string defining the equations can be evaluated within a given namespace by providing keywords at initialisation time, e.g.:

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

In that case, the values of all external variables are taken from the specified dictionary (given by the keyword arguments), even if variables with the same name exist in the namespace where the string was defined. The two methods for passing the values of external variables are mutually exclusive, that is, either all external variables are explicitly specified with keywords (if not, they are left unspecified even if there are variables with the same names in the namespace where the string was defined), or all values are taken from the calling namespace.

More can be done with keyword arguments. If the value is a string, then the name of the variable is replaced, e.g.:

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

changes the variable name x to Vm. This is useful for writing functions which return equations where the variable name is provided by the user.

Finally, if the value is `None` then the name of the variable is replaced by a unique name, e.g.:

```
eqs=Equations('dx/dt=-x/tau : volt',tau=10*ms,x=None)
```

This is useful to avoid conflicts in the names of hidden variables.

Issues

- There can be problems if a variable with the same name as the variable of a differential equation exists in the namespace where the `Equations` object was defined.

4.12.2 Combining equations

`Equations` can be combined using the sum operator. For example:

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

Note that some variables may be undefined when defining the first equation. No error is raised when variables are undefined and absent from the calling namespace. When two `Equations` objects are added, the consistency is checked. For example it is not possible to add two `Equations` objects which define the same variable.

4.12.3 Which variable is the membrane potential?

Several objects, such as `Threshold` or `Reset` objects can be initialised without specifying which variable is the membrane potential, in which case it is assumed that it is the first variable. Internally, the variables of an `Equations` object are reorderered so that the first one is most likely to be the membrane potential (using `Equations.get_Vm()`). The first variable is, with decreasing priority :

- v
- V
- vm
- Vm
- the first defined variable.

4.12.4 Numerical integration

The currently available integration methods are:

- Exact integration when the equations are linear.
- Euler integration (explicit, first order).
- Runge-Kutta integration (explicit, second order).
- Exponential Euler integration (implicit, first order).

The method is selected when a `NeuronGroup` is initialized. If the equations are linear, exact integration is automatically selected. Otherwise, Euler integration is selected by default, unless the keyword `implicit=True` is passed, which selects the exponential Euler method. A second-order method can be selected using the keyword `order=2` (explicit Runge-Kutta method, midpoint estimation). It is possible to override this behaviour with the `method` keyword when initialising a `NeuronGroup`. Possible values are `linear`, `nonlinear`, `Euler`, `RK`, `exponential_Euler`.

Exact integration

If the differential equations are linear, then the update phase X(t)->X(t+dt) can be calculated exactly with a matrix product. First, the equations are examined to determine whether they are linear with the method `islinear()` and the function `is_affine()` (this is currently done using dynamic typing). Second, the matrix M and the vector B such that dX/dt=M(X-B) are calculated with the function `get_linear_equations()` [1]. Third, the matrix A such that X(t+dt)=A*(X(t)-B)+B is calculated at initialisation of a specific state updater object, `LinearStateUpdater`, as A=expm(M*dt), where expm is the matrix exponential.

Important remark: since the update matrix and vector are precalculated, the values of all external variables in the equations are frozen at initialisation. If external variables are modified after initialisation, those modifications are *not* taken into account during the simulation.

[1] Note that this approach raises an issue when dX/dt=B. We currently (temporarily) solve this problem by adding a small diagonal matrix to M to make it invertible.

Inexact exact integration: If the equation cannot be put into the form dX/dt=M(X-B), for example if the equation is dX/dt=MX+A where M is not invertible, then the equations are not integrated exactly, but using a system equivalent to Euler integration but with dt 100 times smaller than specified. Updates are of the form X(t+dt)=A*X(t)+C where the matrix A and vector C are computed by applying Euler integration 100 times to the differential equations.

Euler integration

The Euler is a first order explicit integration method. It is the default one for nonlinear equations. It is simply implemented as X(t+dt)=X(t)+f(X)*dt.

Exponential Euler integration

The exponential Euler method is used for Hodgkin-Huxley type equations, are which stiff. Equations of that type are conditionally linear, that is, the differential equation for each variable is linear in that variable (i.e., linear if all other variables are considered constant). The idea is thus to solve the differential equation for each variable over one time step, assuming that all other variables are constant over that time step. The numerical scheme is still first order, but it is more stable than the forward Euler method. Each equation can be written as dx/dt=a*x+b, where a and b depend on the other variables and thus change after each time step. The values of a and b are obtained during the update phase by calculating a*x+b for x=0 and x=1 (note that these values are different for every neuron, thus we calculate vectors A and B). Then x(t+dt) is calculated in the same way as for the exact integration method above.

4.12.5 Stochastic differential equations

Noise is introduced in differential equations with the keyword `xi`, which means normalised gaussian noise (the derivative of the Brownian term). Currently, this is implemented simply by adding a normal random number to the variable at the end of the integration step (independently for each neuron). The unit of white noise is non-trivial, it is `second**(-.5)`. Thus, a typical stochastic equation reads:

```
dx/dt=-x/tau+sigma*xi/tau**.5
```

where `sigma` is in the same units as `x`. We note the following two facts:

- The noise term is independent between neurons. Thus, one cannot use this method to analyse the response to frozen noise (where all neurons receive the same input noise). One would need to use an external variable representing the input, updated by a user-defined operation.
- The noise term is independent between equations. This can however be solved by the following trick:

```
dx/dt=-x/tau+sigmax*u/tau**.5 : volt
dy/dt=-y/tau+sigmay*u/tau**.5 : volt
u=xi : second**(-.5)
```

4.12.6 Non-autonomous equations

The time variable `t` can be directly inserted into an equation string. It is replaced at run time by the current value of the time variable for the relevant neuron group, and also appears as a state variable of the neuron group.

4.12.7 Freezing

External variables can be frozen by passing the keyword `freeze=True` (default = `False`) at initialization of a `NeuronGroup` object. Then when the string defining the equations are compiled into Python functions (method `compile_functions()`), the external variables are replaced by their float values (units are discarded). This can result in a significant speed-up.

TODO: more on the implementation.

4.12.8 Compilation

State updates can be compiled into Python code objects by passing the keyword `compile=True` at initialization of a a `NeuronGroup`. Note that this is different from the method `compile_functions()`, which compiles the equation for every variable into a Python function (not the whole state update process).

When the `compile` keyword is set, the method `forward_euler_code()` or `exponential_euler_code()` is called. It generates a string containing the Python code for the update of all state variables (one time step), then compiles it into Python code object. That compiled object is then called at every time step. All external variables are frozen in the process (regardless of the value of the `freeze` keyword). This results in a significant speed-up (although the exponential Euler code is not quite optimised yet). Note that only Python code is generated, thus a C compiler is not required.

4.12.9 Working with equations

`Equations` object can also be used outside simulations. In the following, we suppose that an `Equations` object is defined as follows:

```
eqs=Equations('''
dx/dt=(y-x)/(10*ms) : volt
dy/dt=-z/(5*ms) : volt
z=2*(x+y) : volt
''')
```

Applying an equation

The value of z can be calculated using the `apply()` method:

```
z=eqs.apply('z',dict(x=3*mV,y=5*mV))
```

The second argument is a dictionary containing the values of all dependent variables (here the result is `8*mV`). The right-hand side of differential equations can also be calculated in the same way:

```
x=eqs.apply('x',dict(x=2*mV,y=3*mV))
y=eqs.apply('y',dict(x=2*mV,y=3*mV))
```

Note in the second case that only the values of the dynamic variables should be passed.

Calculating a fixed point

A fixed point of the equations can be calculated as follows:

```
fp=eqs.fixedpoint(x=2*mV,y=3*mV)
```

where the optional keywords give the initial point (zero if not provided). Internally, the function `optimize.fsolve` from the Scipy package is used to find a zero of the set of differential equations (thus, convergence is not guaranteed; in that case, the initial values are returned). A dictionary with the values of the dynamic variables at the fixed point is returned.

Issues

- If the equations were previously frozen, then the units disappear from the equations and unit consistency problems may arise.
- `Equations` objects need to be "prepared" before use, as follows:

  ```
  eqs.prepare()
  ```

  This is automatically called by the NeuronGroup initialiser.

For more detailed information, see the reference chapter.

CHAPTER

FIVE

THE LIBRARY

A number of standard models is defined in the library folder. To use library elements, use the following syntax:

```
from brian.library.module_name import *
```

For example, to import electrophysiology models:

```
from brian.library.electrophysiology import *
```

5.1 Library models

5.1.1 Membrane equations

Library models are defined using the `MembraneEquation` class. This is a subclass of `Equations` which is defined by a capacitance C and a sum of currents. The following instruction:

```
eqs=MembraneEquation(200*pF)
```

defines the equation C*dvm/dt=0*amp, with the membrane capacitance C=200 pF. The name of the membrane potential variable can be changed as follows:

```
eqs=MembraneEquation(200*pF,vm='V')
```

The main interest of this class is that one can use it to build models by adding currents to a membrane equation. The `Current` class is a subclass of `Equations` which defines a current to be added to a membrane equation. For example:

```
eqs=MembraneEquation(200*pF)+Current(I='(V0-vm)/R : amp',current_name='I')
```

defines the same equation as:

```
eqs=Equations('''
dvm/dt=I/(200*pF) : volt
I=(V0-vm)/R : amp
''')
```

The keyword `current_name` is optional if there is no ambiguity, i.e., if there is only one variable or only one variable with amp units. As for standard equations, `Current` objects can be initialised with a multiline string (several

equations). By default, the convention for the current direction is the one for injected current. For the ionic current convention, use the `IonicCurrent` class:

```
eqs=MembraneEquation(200*pF)+IonicCurrent(I='(vm-V0)/R : amp')
```

5.1.2 Compartmental modelling

Compartmental neuron models can be created by merging several `MembraneEquation` objects, with the `compartments` module. If `soma` and `dendrite` are two compartments defined as `MembraneEquation` objects, then a neuron with those 2 compartments can be created as follows:

```
neuron_eqs=Compartments({'soma':soma,'dendrite':dendrite})
neuron_eqs.connect('soma','dendrite',Ra)
neuron=NeuronGroup(1,model=neuron_eqs)
```

The `Compartments` object is initialised with a dictionary of `MembraneEquation` objects. The returned object `neuron_eqs` is also a `MembraneEquation` object, where the name of each compartment has been appended to variable names (with a leading underscore). For example, `neuron.vm_soma` refers to variable `vm` of the somatic compartment. The `connect` method adds a coupling current between the two named compartments, with the given resistance `Ra`.

5.1.3 Integrate-and-Fire models

A few standard Integrate-and-Fire models are implemented in the `IF` library module:

```
from brian.library.IF import *
```

All these functions return `Equations` objects (more precisely, `MembraneEquation` objects).

- Leaky integrate-and-fire model (`dvm/dt=(El-vm)/tau : volt`):

  ```
  eqs=leaky_IF(tau=10*ms,El=-70*mV)
  ```

- Perfect integrator (`dvm/dt=Im/tau : volt`):

  ```
  eqs=perfect_IF(tau=10*ms)
  ```

- Quadratic integrate-and-fire model (`C*dvm/dt=a*(vm-El)*(vm-VT) : volt`):

  ```
  eqs=quadratic_IF(C=200*pF,a=10*nS/mV,EL=-70*mV,VT=-50*mV)
  ```

- Exponential integrate-and-fire model (`C*dvm/dt=gL*(EL-vm)+gL*DeltaT*exp((vm-VT)/DeltaT) :volt`):

  ```
  eqs=exp_IF(C=200*pF,gL=10*nS,EL=-70*mV,VT=-55*mV,DeltaT=3*mV)
  ```

In general, it is possible to define a neuron group with different parameter values for each neuron, by passing strings at initialisation. For example, the following code defines leaky integrate-and-fire models with heterogeneous resting potential values:

```
eqs=leaky_IF(tau=10*ms,El='V0')+Equations('V0:volt')
group=NeuronGroup(100,model=eqs,reset=0*mV,threshold=15*mV)
```

5.1.4 Two-dimensional IF models

Integrate-and-fire models with two variables can display a very rich set of electrophysiological behaviours. In Brian, two such models have been implemented: Izhikevich model and Brette-Gerstner adaptive exponential integrate-and-fire model (also included in the `IF` module). The equations are obtained in the same way as for one-dimensional models:

```
eqs=Izhikevich(a=0.02/ms,b=0.2/ms)
eqs=Brette_Gerstner(C=281*pF,gL=30*nS,EL=-70.6*mV,VT=-50.4*mV,DeltaT=2*mV,tauw=144*ms,a=4*nS)
eqs=aEIF(C=281*pF,gL=30*nS,EL=-70.6*mV,VT=-50.4*mV,DeltaT=2*mV,tauw=144*ms,a=4*nS) # equivalent
```

and two state variables are defined: `vm` (membrane potential) and `w` (adaptation variable). The equivalent equations for Izhikevich model are:

```
dvm/dt=(0.04/ms/mV)*vm**2+(5/ms)*vm+140*mV/ms-w : volt/second
dw/dt=a*(b*vm-w)                                : volt/second
```

and for Brette-Gerstner model:

```
C*dvm/dt=gL*(EL-vm)+gL*DeltaT*exp((vm-VT)/DeltaT)-w :volt
dw/dt=(a*(vm-EL)-w)/tauw : amp
```

To simulate these models, one needs to specify a threshold value, and a good choice is `VT+4*DeltaT`. The reset is particular in these models since it is bidimensional: vm->Vr and w->w+b. A specific reset class has been implemented for this purpose: `AdaptiveReset`, initialised with Vr and b. Thus, a typical construction of a group of such models is:

```
eqs=Brette_Gerstner(C=281*pF,gL=30*nS,EL=-70.6*mV,VT=-50.4*mV,DeltaT=2*mV,tauw=144*ms,a=4*nS)
group=NeuronGroup(100,model=eqs,threshold=-43*mV,reset=AdaptiveReset(Vr=-70.6*mvolt,b=0.0805*nA))
```

5.1.5 Synapses

A few simple synaptic models are implemented in the module `synapses`:

```
from brian.library.synapses import *
```

All the following functions need to be passed the name of the variable upon which the received spikes will act, and the name of the variable representing the current or conductance. The simplest one is the exponential synapse:

```
eqs=exp_synapse(input='x',tau=10*ms,unit=amp,output='x_current')
```

It is equivalent to:

```
eqs=Equations('''
dx/dt=-x/tau : amp
x_out=x
''')
```

Here, `x` is the variable which receives the spikes and `x_current` is the variable to be inserted in the membrane equation (since it is a one-dimensional synaptic model, the variables are the same). If the output variable name is not defined, then it will be automatically generated by adding the suffix `_out` to the input name.

Two other types of synapses are implemented. The alpha synapse (`x(t)=alpha*(t/tau)*exp(1-t/tau)`, where `alpha` is a normalising factor) is defined with the same syntax by:

```
eqs=alpha_synapse(input='x',tau=10*ms,unit=amp)
```

and the bi-exponential synapse is defined by (`x(t)=(tau2/(tau2-tau1))*(exp(-t/tau1)-exp(-t/tau2))`, up to a normalising factor):

```
eqs=biexp_synapse(input='x',tau1=10*ms,tau2=5*ms,unit=amp)
```

For all types of synapses, the normalising factor is such that the maximum of x(t) is 1. These functions can be used as in the following example:

```
eqs=MembraneEquation(C=200*pF)+Current('I=gl*(El-vm)+ge*(Ee-vm):amp')
eqs+=alpha_synapse(input='ge_in',tau=10*ms,unit=siemens,output='ge')
```

where alpha conductances have been inserted in the membrane equation.

One can directly insert synaptic currents with the functions `exp_current`, `alpha_current` and `biexp_current`:

```
eqs=MembraneEquation(C=200*pF)+Current('I=gl*(El-vm):amp')+\
    alpha_current(input='ge',tau=10*ms)
```

(the units is amp by default), or synaptic conductances with the functions `exp_conductance`, `alpha_conductance` and `biexp_conductance`:

```
eqs=MembraneEquation(C=200*pF)+Current('I=gl*(El-vm):amp')+\
    alpha_conductance(input='ge',E=0*mV,tau=10*ms)
```

where `E` is the reversal potential.

5.1.6 Ionic currents

A few standard ionic currents have implemented in the module `ionic_currents`:

```
from brian.library.ionic_currents import *
```

When the current name is not specified, a unique name is generated automatically. Models can be constructed by adding currents to a `MembraneEquation`.

- Leak current (`gl*(El-vm)`):

  ```
  current=leak_current(gl=10*nS,El=-70*mV,current_name='I')
  ```

- Hodgkin-Huxley K+ current:

  ```
  current=K_current_HH(gmax,EK,current_name='IK'):
  ```

- Hodgkin-Huxley Na+ current:

  ```
  current=Na_current_HH(gmax,ENa,current_name='INa'):
  ```

5.2 Random processes

To import the random processes library:

```
from brian.library.random_processes import *
```

For the moment, only the Ornstein-Uhlenbeck process has been included. The function `OrnsteinUhlenbeck()` returns an `Equations` object. The following example defines a membrane equation with an Ornstein-Uhlenbeck current `I` (= coloured noise):

```
eqs=Equations('dv/dt=-v/tau+I/C : volt')
eqs+=OrnsteinUhlenbeck('I',mu=1*nA,sigma=2*nA,tau=10*ms)
```

where `mu` is the mean of the current, `sigma` is the standard deviation and `tau` is autocorrelation time constant.

5.3 Electrophysiology

The electrophysiology library contains a number of models of electrodes, amplifiers and recording protocols to simulate intracellular electrophysiological recordings. To import the electrophysiology library:

```
from brian.library.electrophysiology import *
```

There is a series of example scripts in the examples/electrophysiology folder.

5.3.1 Electrodes

Electrodes are defined as resistor/capacitor (RC) circuits, or multiple RC circuits in series. Define a simple RC electrode with resistance Re and capacitance Ce (possibly 0 pF) as follows:

```
el=electrode(Re,Ce)
```

The `electrode` function returns an `Equations` object containing the electrode model, where the electrode potential is `v_el` (the recording), the membrane potential is `vm`, the electrode current entering the membrane is `i_inj` and command current is `i_cmd`. These names can be overriden using the corresponding keywords. For example, a membrane equation with a .5 nA current injected through an electrode is defined as follows:

```
eqs=Equations('dv/dt=(-gl*v+i_inj)/Cm : volt')+electrode(50*Mohm,10*pF,vm='v',i_cmd=.5*nA)
```

Specify `i_cmd=None` if the electrode is only used to record (no current injection). More complex electrodes can be defined by passing lists of resistances and capacitances, e.g.:

```
el=electrode([50*Mohm,20*Mohm],[5*pF,3*pF])
```

5.3.2 Amplifiers

Current-clamp amplifier

A current-clamp amplifier injects a current through an intracellular electrode and records the membrane potential. Two standard circuits are included to compensate for the electrode voltage: bridge compensation and capacitance

neutralization (see e.g. the Axon guide). The following command:

```
amp=current_clamp(Re=80*Mohm,Ce=10*pF)
```

defines a current-clamp amplifier with an electrode modelled as a RC circuit. The function returns an `Equations` object, where the recording potential is `v_rec`, the membrane potential is `vm`, the electrode current entering the membrane is `i_inj` and command current is `i_cmd`. These names can be overriden using the corresponding keywords. For implementation reasons, the amplifier always includes an electrode. Optionally, bridge compensation, can be used with the `bridge` keyword and capacitance neutralization with the `capa_comp` keyword. For example, the following instruction defines a partially compensated recording:

```
amp=current_clamp(Re=80*Mohm,Ce=10*pF,bridge=78*Mohm,capa_comp=8*pF)
```

The capacitance neutralization is a feedback circuit, so that it becomes unstable if the feedback capacitance is larger than the actual capacitance of the electrode. The bridge compensation is an input-dependent voltage offset (`bridge*i_cmd`), and thus is always stable (unless an additional feedback, such as dynamic clamp, is provided). Note that the bridge and capacitance neutralization parameters can be variable names, e.g.:

```
amp=current_clamp(Re=80*Mohm,Ce=10*pF,bridge='Rbridge',capa_comp=8*pF)
```

and then the bridge compensation can be changed dynamically during the simulation.

Voltage-clamp amplifier

The library includes a single-electrode voltage-clamp amplifier, which clamps the potential at a given value and records the current going through the electrode. The following command:

```
amp=voltage_clamp(Re=20*Mohm)
```

defines a voltage-clamp amplifier with an electrode modelled as a pure resistance. The function returns an `Equations` object, where the recording current is `i_rec`, the membrane potential is `vm`, the electrode current entering the membrane is `i_inj` and command voltage is `v_cmd` (note that `i_rec` = - `i_inj`). These names can be overriden using the corresponding keywords. For implementation reasons, the amplifier always includes an electrode. Electrode capacitance is not included, meaning that the capacitance neutralization circuit is always set at the maximum value. The quality of the clamp is limited by the electrode or ''series" resistance, which can be compensated in a similar way as bridge compensation in current-clamp recordings. Series resistance compensation consists in adding a current-dependent voltage offset to the voltage command. Because of the feedback, that compensation needs to be slightly delayed (with a low-pass circuit). The following example defines a voltage-clamp amplifier with half-compensated series resistance and compensation delay 1 ms:

```
amp=voltage_clamp(Re=20*Mohm,Rs=10*Mohm,tau_u=1*ms)
```

The `tau_u` keyword is optional and defaults to 1 ms.

Acquisition board

An acquisition board samples a recording and sends a command (e.g. injected current) at regular times. It is defined as a `NeuronGroup`. Use:

```
board=AcquisitionBoard(P=neuron,V='V',I='I',clock)
```

where `P` = neuron group (possibly containing amplifier and electrode), `V` = potential variable name, `I` = current variable name, `clock` = acquisition clock. The recording variable is then stored in `board.record` and a command is sent with the instruction `board.command=I`.

Discontinuous current clamp

The discontinuous current clamp (DCC) consists in alternatively injecting current and measuring the potential, in order to measure the potential when the voltage across the electrode has vanished. The sampling clock is mainly determined by the electrode time constant (the sampling period should be two orders of magnitude larger than the electrode time constant). It is defined and used in the same way as an acquisition board (above):

```
board=DCC(P=neuron,V='V',I='I',frequency=2*kHz)
```

where `frequency` is the sampling frequency. The duty cycle is 1/3 (meaning current is injected during 1/3 of each sampling step).

Discontinuous voltage clamp

The discontinuous voltage clamp or single-electrode voltage clamp (SEVC) is an implementation of the voltage clamp using a feedback current with a DCC amplifier. It is defined as the DCC:

```
board=SEVC(P=neuron,V='V',I='I',frequency=2*kHz,gain=10*nS)
```

except that a gain parameter is included. The SEVC injects a negative feedback current I=gain*(Vcommand-V). The quality of the clamp improves with higher gains, but there is a maximum value above which the system is unstable, because of the finite temporal resolution. The recorded current is stored in `board.record` and the command voltage is sent with the instruction `board.command=-20*mV`. With this implementation of the SEVC, the membrane is never perfectly clamped. A better clamp is obtained by adding an integral controller with the keyword `gain2=10*nS/ms`. The additional current J(t) is governed by the differential equation dJ/dt=gain2*(Vcommand-V), so that it ensures perfect clamping in the stationary state. However, this controller does not improve the settling time of the clamp, but only the final voltage value.

5.3.3 Active Electrode Compensation (AEC)

The electrophysiology library includes the Active Electrode Compensation (AEC) technique described in Brette et al (2008), High-resolution intracellular recordings using a real-time computational model of the electrode, Neuron 59(3):379-91.

Offline AEC

Given a digital current-clamp recording of the (uncompensated) potential `v` (vector of values) and injected current `i`, the following instructions calculate the full kernel of the system and the electrode kernel:

```
K=full_kernel(v,i,ksize)
Ke=electrode_kernel_soma(K,start_tail)
```

`ksize` is the size of the full kernel (number of sampling steps; typical size is about 15 ms) and `start_tail` is the size of the electrode kernel (start point of the ''tail'' of the full kernel; typical size if about 4 ms). The electrode should be compensated for capacitance (capacitance neutralization) but not resistance (bridge compensation). The best choice for the input current is a series of independent random values, and the last `ksize` steps of `v` should be null (i.e., the injection should stop before the end). Here it was assumed that the recording was done at the soma; if it is done in

a thin process such as a dendrite or axon, the function `electrode_kernel_dendrite` should be used instead. The full kernel can also be obtained from a step current injection:

```
K=full_kernel_from_step(v,i,ksize)
Ke=electrode_kernel_soma(K,start_tail)
```

where `i` is a constant value in this case (note that this is not the best choice for real recordings).

Once the electrode kernel has been found, any recording can be compensated as follows:

```
vcomp=AEC_compensate(v,i,ke)
```

where `v` is the raw voltage recording, `i` is the injected current and `ke` is the electrode kernel.

Online AEC

For dynamic-clamp or voltage-clamp recordings, the electrode compensation must be done online. An AEC board is initialized in the same way as an acquisition board:

```
board=AEC(neuron,'V','I',clock)
```

where `clock` is the acquisition clock. The estimation phase typically looks like:

```
board.start_injection()
run(2*second)
board.start_injection()
run(100*ms)
board.estimate()
```

where white noise is injected for 2 seconds (default amplitude .5 nA). You can change the default amplitude and DC current as follows: `board.start_injection(amp=.5*nA,DC=1*nA)`. After estimation, the kernel is stored in `board.Ke`. The following options can be passed to the function `estimate`: `ksize` (default 150 sampling steps), `ktail` (default 50 sampling steps) and `dendritic` (default `False`, use `True` is the recording is a thin process, i.e., axon or dendrite). Online compensation is then switched on with `board.switch_on()` and off with `board.switch_off()`. For example, to inject a .5 nA current pulse for 200 ms, use the following instructions:

```
board.switch_on()
board.command=.5*nA
run(200*ms)
board.command=0*nA
run(150*ms)
board.switch_off()
```

During the simulation, the variable `board.record` stores the compensated potential.

Voltage-clamp with AEC

To be documented!

5.4 Extending the library

For names, we try to follow the standard Python convention, that is, `function_and_variable_names` and `ClassNames`.

CHAPTER

SIX

ADVANCED CONCEPTS

6.1 Parallel computing

ppfunction(*f*)

Convenience wrapper for writing functions to use with parallelpython

The parallel python module `pp` allows you to write code that runs simultaneously on multiple cores on a single machine, or multiple machines on a cluster. You can download the module at http://www.parallelpython.com/

One annoying feature of `pp` is that you cannot use data values that are in the global namespace as part of your code, only modules and functions. This means that you could not execute a function with the code `3*mV` for example, because `mV` is a data value in the `brian` namespace. Instead you would have to import the `brian` module and write `3*brian.mV` which is annoying in long chunks of code. This decorator simply generates a new version of your code with every name `x` from the `brian` namespace replaced by `brian.x`. As part of this process, the rewritten code has to be saved to a file (because of how `pp` works) and this file is named according to the scheme `basename_funcname_parallelpythonised.py`, where `basename` is the current module name and `funcname` is the name of the function being rewritten.

Example

```
@ppfunction
def testf():
    return 3*mV
```

6.2 How to write efficient Brian code

There are a few keys to writing fast and efficient Brian code. The first is to use Brian itself efficiently. The second is to write good vectorised code, which is using Python and NumPy efficiently. For more performance tips, see also *Compiled code*.

6.2.1 Brian specifics

You can switch off Brian's entire unit checking module by including the line:

```
import brian_no_units
```

before importing Brian itself. Good practice is to leave unit checking on most of the time when developing and debugging a model, but switching it off for long runs once the basic model is stable.

Another way to speed up code is to store references to arrays rather than extracting them from Brian objects each time you need them. For example, if you know the custom reset object in the code above is only ever applied to a group `custom_group` say, then you could do something like this:

```
def myreset(P, spikes):
        custom_group_V_[spikes] = 0*mV
        custom_group_Vt_[spikes] = 2*mV

custom_group = ...
custom_group_V_ = custom_group.V_
custom_group_Vt_ = custom_group.Vt_
```

In this case, the speed increase will be quite small, and probably not worth doing because it makes it less readable, but in more complicated examples where you repeatedly refer to `custom_group.V_` it could add up.

6.2.2 Vectorisation

Python is a fast language, but each line of Python code has an associated overhead attached to it. Sometimes you can get considerable increases in speed by writing a vectorised version of it. A good guide to this in general is the Performance Python page. Here we will do a single worked example in Brian.

Suppose you wanted to multiplicatively depress the connection strengths every time step by some amount, you might do something like this:

```
C = Connection(G1, G2, 'V', structure='dense')
...
@network_operation(when='end')
def depress_C():
        for i in range(len(G1)):
                for j in range(len(G2)):
                        C[i,j] = C[i,j]*depression_factor
```

This will work, but it will be very, very slow.

The first thing to note is that the Python expression `range(N)` actually constructs a list `[0,1,2,...,N-1]` each time it is called, which is not really necessary if you are only iterating over the list. Instead, use the `xrange` iterator which doesn't construct the list explicitly:

```
for i in xrange(len(G1)):
        for j in xrange(len(G2)):
                C[i,j] = C[i,j]*depression_factor
```

The next thing to note is that when you call C[i,j] you are doing an operation on the `Connection` object, not directly on the underlying matrix. Instead, do something like this:

```
C = Connection(G1, G2, 'V', structure='dense')
C_matrix = asarray(C.W)
...
@network_operation(when='end')
def depress_C():
        for i in xrange(len(G1)):
                for j in xrange(len(G2)):
                        C_matrix[i,j] *= depression_factor
```

What's going on here? First of all, `C.W` refers to the `ConnectionMatrix` object, which is a 2D NumPy array with some extra stuff - we don't need the extra stuff so we convert it to a straight NumPy array `asarray(C.W)`. We also

store a copy of this as the variable `C_matrix` so we don't need to do this every time step. The other thing we do is to use the `*=` operator instead of the `*` operator.

The most important step of all though is to vectorise the entire operation. You don't need to loop over `i` and `j` at all, you can manipulate the array object with a single NumPy expression:

```
C = Connection(G1, G2, 'V', structure='dense')
C_matrix = asarray(C.W)
...
@network_operation(when='end')
def depress_C():
        C_matrix *= depression_factor
```

This final version will probably be hundreds if not thousands of times faster than the original. It's usually possible to work out a way using NumPy expressions only to do whatever you want in a vectorised way, but in some very rare instances it might be necessary to have a loop. In this case, if this loop is slowing your code down, you might want to try writing that loop in inline C++ using the SciPy Weave package. See the documentation at that link for more details, but as an example we could rewrite the code above using inline C++ as follows:

```
from scipy import weave
...
C = Connection(G1, G2, 'V', structure='dense')
C_matrix = asarray(C.W)
...
@network_operation(when='end')
def depress_C():
        n = len(G1)
        m = len(G2)
        code = '''
                for(int i=0;i<n;i++)
                        for(int j=0;j<m;j++)
                                C_matrix(i,j) *= depression_factor
                '''
        weave.inline(code,
                ['C_matrix', 'n', 'm', 'depression_factor'],
                type_converters=weave.converters.blitz,
                compiler='gcc',
                extra_compile_args=['-O3'])
```

The first time you run this it will be slower because it compiles the C++ code and stores a copy, but the second time will be much faster as it just loads the saved copy. The way it works is that Weave converts the listed Python and NumPy variables (`C_matrix`, `n`, `m` and `depression_factor`) into C++ compatible data types. `n` and `m` are turned into `int``s, ``depression_factor` is turned into a `double`, and `C_matrix` is turned into a Weave `Array` class. The only thing you need to know about this is that elements of a Weave array are referenced with parentheses rather than brackets, i.e. `C_matrix(i,j)` rather than `C_matrix[i,j]`. In this example, I have used the `gcc` compiler and added the optimisation flag `-O3` for maximum optimisations. Again, in this case it's much simpler to just use the `C_matrix *= depression_factor` NumPy expression, but in some cases using inline C++ might be necessary, and as you can see above, it's very easy to do this with Weave, and the C++ code for a snippet like this is often almost as simple as the Python code would be.

6.3 Compiled code

Compiled C code can be used in several places in Brian to get speed improvements in cases where performance is the most important factor.

6.3.1 Weave

Weave is a SciPy module that allows the use of inlined C++ code. Brian by default doesn't use any C++ optimisations for maximum compatibility across platforms, but you can enable several optimised versions of Brian objects and functions by enabling weave compilation. See *Preferences* for more information.

See also *Vectorisation* for some information on writing your own inlined C++ code using Weave.

6.3.2 Circular arrays

For maximum compatibility, Brian works with pure Python only. However, as well as the optional weave optimisations, there is also an object used in the spike propagation code that can run with a pure C++ version for a considerable speedup (1.5-3x). You need a copy of the `gcc` compiler installed (either on linux or through cygwin on Windows) to build it.

Installation:

In a command prompt or shell window, go to the directory where Brian is installed. On Windows this will probably be `C:\Python25\lib\site-packages\brian`. Now go to the Brian/brian/utils/ccircular folder. If you're on Linux (and this may also work for Mac) run the command "python setup.py build_ext –inplace". If you're on windows you'll need to have cygwin with gcc installed, and then you run "setup.py build_ext –inplace -c mingw32" instead. You should see some compilation, possibly with some warnings but no errors.

6.3.3 Automatically generated C code

There is an experimental module for automatic generation of C code, see *Automatic C code generation for nonlinear state updaters* for details.

6.4 Projects with multiple files or functions

Brian works with the minimal hassle if the whole of your code is in a single Python module (`.py` file). This is fine when learning Brian or for quick projects, but for larger, more realistic projects with the source code separated into multiple files, there are some small issues you need to be aware of. These issues essentially revolve around the use of the ''magic" functions `run()`, etc. The way these functions work is to look for objects of the required type that have been instantiated (created) in the same ''execution frame" as the `run()` function. In a small script, that is normally just any objects that have been defined in that script. However, if you define objects in a different module, or in a function, then the magic functions won't be able to find them.

There are three main approaches then to splitting code over multiple files (or functions).

6.4.1 Use the `Network` object explicitly

The magic `run()` function works by creating a `Network` object automatically, and then running that network. Instead of doing this automatically, you can create your own `Network` object. Rather than writing something like:

```
group1 = ...
group2 = ...
C = Connection(group1,group2)
...
run(1*second)
```

You do this:

```
group1 = ...
group2 = ...
C = Connection(group1, group2)
...
net = Network(group1, group2, C)
net.run(1*second)
```

In other words, you explicitly say which objects are in your network. Note that any `NeuronGroup`, `Connection`, `Monitor` or function decorated with `network_operation()` should be included in the `Network`. See the documentation for `Network` for more details.

This is the preferred solution for almost all cases. You may want to use either of the following two solutions if you think your code may be used by someone else, or if you want to make it into an extension to Brian.

6.4.2 Use the `magic_return()` decorator or `magic_register()` function

The `magic_return()` decorator is used as follows:

```
@magic_return
def f():
        ...
        return obj
```

Any object returned by a function decorated by `magic_return()` will be considered to have been instantiated in the execution frame that called the function. In other words, the magic functions will find that object even though it was really instantiated in a different execution frame.

In more complicated scenarios, you may want to use the `magic_register()` function. For example:

```
def f():
        ...
        magic_register(obj1, obj2)
        return (obj1, obj2)
```

This does the same thing as `magic_return()` but can be used with multiple objects. Also, you can specify a `level` (see documentation on `magic_register()` for more details).

6.4.3 Use derived classes

Rather than writing a function which returns an object, you could instead write a derived class of the object type. So, suppose you wanted to have an object that emitted N equally spaced spikes, with an interval dt between them, you could use the `SpikeGeneratorGroup` class as follows:

```
@magic_return
def equally_spaced_spike_group(N, dt):
        spikes = [(0,i*dt) for i in range(N)]
        return SpikeGeneratorGroup(spikes)
```

Or alternatively, you could derive a class from `SpikeGeneratorGroup` as follows:

```
class EquallySpacedSpikeGroup(SpikeGeneratorGroup):
        def __init__(self, N, t):
                spikes = [(0,i*dt) for i in range(N)]
                SpikeGeneratorGroup.__init__(self, spikes)
```

You would use these objects in the following ways:

```
obj1 = equally_spaced_spike_group(100, 10*ms)
obj2 = EquallySpacedSpikeGroup(100, 10*ms)
```

For simple examples like the one above, there's no particular benefit to using derived classes, but using derived classes allows you to add methods to your derived class for example, which might be useful. For more experienced Python programmers, or those who are thinking about making their code into an extension for Brian, this is probably the preferred approach. Finally, it may be useful to note that there is a protocol for one object to 'contain' other objects. That is, suppose you want to have an object that can be treated as a simple `NeuronGroup` by the person using it, but actually instantiates several objects (perhaps internal `Connection` objects). These objects need to be added to the `Network` object in order for them to be run with the simulation, but the user shouldn't need to have to know about them. To this end, for any object added to a `Network`, if it has an attribute `contained_objects`, then any objects in that container will also be added to the network.

6.5 Parameters

Brian includes a simple tool for keeping track of parameters. If you only need something simple, then a dict or an empty class could be used. The point of the parameters class is that allows you to define a cascade of computed parameters that depend on the values of other parameters, so that changing one will automatically update the others. See the synfire chain example `examples/sfc.py` for a demonstration of how it can be used.

class `Parameters` (***kwds*)

A storage class for keeping track of parameters

Example usage:

```
p = Parameters(
    a = 5,
    b = 6,
    computed_parameters = '''
    c = a + b
    ''')
print p.c
p.a = 1
print p.c
```

The first `print` statement will give 11, the second gives 7.

Details:

Call as:

```
p = Parameters(...)
```

Where the `...` consists of a list of keyword / value pairs (like a `dict`). Keywords must not start with the underscore _ character. Any keyword that starts with `computed_` should be a string of valid Python statements that compute new values based on the given ones. Whenever a non-computed value is changed, the computed parameters are recomputed, in alphabetical order of their keyword names (so `computed_a` is computed before `computed_b` for example). Non-computed values can be accessed and set via `p.x`, `p.x=1` for example, whereas computed values can only be accessed and not set. New parameters can be added after the `Parameters` object is created, including new `computed_*` parameters. You can 'derive' a new parameters object from a given one as follows:

```
p1 = Parameters(x=1)
p2 = Parameters(y=2,**p1)
print p2.x
```

Note that changing the value of `x` in `p2` will not change the value of `x` in `p1` (this is a copy operation).

6.6 Precalculated tables

One way to speed up simulations is to use precalculated tables for complicated functions. The `Tabulate` class defines a table of values of the given function at regularly sampled points. The `TabulateInterp` class defines a table with linear interpolation, which is much more precise. Both work with scalar and vector arguments.

class `Tabulate` (*f, xmin, xmax, n*)
An object to tabulate a numerical function.

Sample use:

```
g=Tabulate(f,0.,1.,1000)
y=g(.5)
v=g([.1,.3])
v=g(array([.1,.3]))
```

Arguments of g must lie in [xmin,xmax). An IndexError is raised is arguments are above xmax, but not always when they are below xmin (it can give weird results).

class `TabulateInterp` (*f, xmin, xmax, n*)
An object to tabulate a numerical function with linear interpolation.

Sample use:

```
g=TabulateInterp(f,0.,1.,1000)
y=g(.5)
v=g([.1,.3])
v=g(array([.1,.3]))
```

Arguments of g must lie in [xmin,xmax). An IndexError is raised is arguments are above xmax, but not always when they are below xmin (it can give weird results).

6.7 Preferences

6.7.1 Functions

Setting and getting global preferences is done with the following functions:

`set_global_preferences` (***kwds*)
Set global preferences for Brian

Usage:

```
``set_global_preferences(...)``
```

where ... is a list of keyword assignments.

`get_global_preference` (*k*)
Get the value of the named global preference

6.7.2 Global configuration file

If you have a module named `brian_global_config` anywhere on your Python path, Brian will attempt to import it to define global preferences. For example, to automatically enable weave compilation for all your Brian projects, create a file `brian_global_config.py` somewhere in the Python path with the following contents:

```
from brian.globalprefs import *
set_global_preferences(useweave=True)
```

6.7.3 Global preferences for Brian

The following global preferences have been defined:

`defaultclock = Clock(dt=0.1*msecond)` The default clock to use if none is provided or defined in any enclosing scope.

`useweave_linear_diffeq = False` Whether to use weave C++ acceleration for the solution of linear differential equations. Note that on some platforms, typically older ones, this is faster and on some platforms, typically new ones, this is actually slower.

`useweave = False` Defines whether or not functions should use inlined compiled C code where defined. Requires a compatible C++ compiler. The `gcc` and `g++` compilers are probably the easiest option (use Cygwin on Windows machines). See also the `weavecompiler` global preference.

`weavecompiler = gcc` Defines the compiler to use for weave compilation. On Windows machines, installing Cygwin is the easiest way to get access to the gcc compiler.

`magic_useframes = True` Defines whether or not the magic functions should search for objects defined only in the calling frame or if they should find all objects defined in any frame. This should be set to `False` if you are using Brian from an interactive shell like IDLE or IPython where each command has its own frame, otherwise set it to `True`.

6.8 Logging

Brian uses the standard Python `logging` package to generate information and warnings. All messages are sent to the logger named `brian` or loggers derived from this one, and you can use the standard logging functions to set options, write the logs to files, etc. Alternatively, Brian has four simple functions to set the level of the displayed log (see below). There are four different levels for log messages, in decreasing order of severity they are ERROR, WARN, INFO and DEBUG. By default, Brian displays only the WARN and ERROR level messages. Some useful information is at the INFO level, so if you are having problems with your program, setting the level to INFO may help.

`log_level_error``()`
Shows log messages only of level ERROR or higher.

`log_level_warn``()`
Shows log messages only of level WARNING or higher (including ERROR level).

`log_level_info``()`
Shows log messages only of level INFO or higher (including WARNING and ERROR levels).

`log_level_debug``()`
Shows log messages only of level DEBUG or higher (including INFO, WARNING and ERROR levels).

CHAPTER

SEVEN

EXTENDING BRIAN

TODO: Description of how to extend Brian, add new model types, and maybe at some point how to upload them to a database, share with others, etc.

For the moment, see the documentation on *Projects with multiple files or functions*.

CHAPTER

EIGHT

REFERENCE

For an overview of Brian, see the *User manual* section.

8.1 SciPy, NumPy and PyLab

See the following web sites:

- http://www.scipy.org/Getting_Started
- http://www.scipy.org/Documentation
- http://matplotlib.sourceforge.net/matplotlib.pylab.html

8.2 Units system

have_same_dimensions(*obj1, obj2*)
Tests if two scalar values have the same dimensions, returns a `bool`.

Note that the syntax may change in later releases of Brian, with tighter integration of scalar and array valued quantities.

is_dimensionless(*obj*)
Tests if a scalar value is dimensionless or not, returns a `bool`.

Note that the syntax may change in later releases of Brian, with tighter integration of scalar and array valued quantities.

exception DimensionMismatchError
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. If you want to check for inconsistent units in your code, do something like:

```
try:
    ...
    your code here
    ...
except DimensionMismatchError, inst:
    ...
    cleanup code here, e.g.
    print "Found dimension mismatch, details:", inst
    ...
```

check_units(**au)
Decorator to check units of arguments passed to a function

Sample usage:

```
@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)
```

would fail, but:

```
getvoltage(1*amp,1*ohm,wibble=1*metre)
```

would pass. String arguments are not checked (e.g. `getvoltage(wibble='hello')` would pass).

The special name `result` is for the return value of the function.

An error in the input value raises a `DimensionMismatchError`, and an error in the return value raises an `AssertionError` (because it is a code problem rather than a value problem).

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.

Typically, you shouldn't need to use any details about the following two classes, and their implementations are subject to change in future releases of Brian.

class Quantity(*value*)
A number with an associated physical dimension.

In most cases, it is not necessary to create a `Quantity` object by hand, instead use the constant unit names `second`, `kilogram`, etc. The details of how `Quantity` objects work is subject to change in future releases of Brian, as we plan to reimplement it in a more efficient manner, more tightly integrated with numpy. The following can be safely used:

- `Quantity`, this name will not change, and the usage `isinstance(x,Quantity)` should be safe.
- The standard unit objects, `second`, `kilogram`, etc. documented in the main documentation will not be subject to change (as they are based on SI standardisation).
- Scalar arithmetic will work with future implementations.

class Unit(*value*)
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). You can define new units which will be used when generating string representations of quantities simply by doing an arithmetical operation with only units, for example:

```
Nm = newton * metre
```

Note that operations with units are slower than operations with `Quantity` objects, so for efficiency if you do not need the extra information that a `Unit` object carries around, write `1*second` in preference to `second`.

8.3 Clocks

Many Brian objects store a clock object (always passed in the initialiser with the keyword `clock=...`). If no clock is specified, the program uses the global default clock. When Brian is initially imported, this is the object `defaultclock`, and it has a default time step of 0.1ms. In a simple script, you can override this by writing (for example):

```
defaultclock.dt = 1*ms
```

However, there are other ways to access or redefine the default clock (see functions below). You may wish to use multiple clocks in your program. In this case, for each object which requires one, you have to pass a copy of its `Clock` object. The network run function automatically handles objects with different clocks, updating them all at the appropriate time according to their time steps (value of `dt`).

Multiple clocks can be useful, for example, for defining a simulation that runs with a very small `dt`, but with some computationally expensive operation running at a lower frequency.

8.3.1 The `Clock` class

class `Clock`(*dt=0.1 ms, t=0.0 s, makedefaultclock=False*)

An object that holds the simulation time and the time step.

Initialisation arguments:

`dt` The time step of the simulation.

`t` The current time of the clock.

`makedefaultclock` Set to `True` to make this clock the default clock.

Methods

`reinit`(*[t=0*second]*)
Reinitialises the clock time to zero (or to your specified time).

Attributes

`t`
`dt`
Current time and time step with units.

Advanced

Attributes

`end`
The time at which the current simulation will end, set by the `Network.run()` method.

Methods

`tick`()
Advances the clock by one time step.

`set_t`(*t*)
`set_dt`(*dt*)
`set_end`(*end*)
Set the various parameters.

`get_duration`()
The time until the current simulation ends.

`set_duration`(*duration*)
Set the time until the current simulation ends.

still_running()
Returns a `bool` to indicate whether the current simulation is still running.

For reasons of efficiency, we recommend using the methods `tick()`, `set_duration()` and `still_running()` (which bypass unit checking internally).

8.3.2 The default clock

defaultclock
The default clock object

Note that this is only the default clock object if you haven't redefined it with the `define_default_clock()` function or the `makedefaultclock=True` option of a `Clock` object. A safe way to get hold of the default clock is to use the functions:

- `get_default_clock()`
- `reinit_default_clock()`

However, it is suitable for short scripts, e.g.:

```
defaultclock.dt = 1*ms
...
```

define_default_clock(***kwds*)
Create a new default clock

Uses the keywords of the `Clock` initialiser.

Sample usage:

```
define_default_clock(dt=1*ms)
```

reinit_default_clock(*t=0.0 s*)
Reinitialise the default clock (to zero or a specified time)

get_default_clock()
Returns the default clock object.

8.4 Neuron models and groups

8.4.1 The `Equations` object

class **Equations**(*expr='', level=0, **kwds*)
Container that stores equations from which models can be created

Initialised as:

```
Equations(expr[,level=0[,keywords...]])
```

with arguments:

expr An expression, which can each be a string representing equations, an `Equations` objects, or a list of strings and `Equations` objects. See below for details of the string format.

level Indicates how many levels back in the stack the namespace for string equations is found, so that e.g. `level=0` looks in the namespace of the function where the `Equations` object was created, `level=1` would look in the namespace of the function that called the function where the `Equations` object was created, etc. Normally you can just leave this out.

keywords Any sequence of keyword pairs `key=value` where the string `key` in the string equations will be replaced with `value` which can be either a string, value or `None`, in the latter case a unique name will be generated automatically (but it won't be pretty).

Systems of equations can be defined by passing lists of `Equations` to a new `Equations` object, or by adding `Equations` objects together (the usage is similar to that of a Python `list`).

String equations

String equations can be of any of the following forms:

1. `dx/dt = f : unit` (differential equation)
2. `x = f : unit` (equation)
3. `x = y` (alias)
4. `x : unit` (parameter)

Here each of `x` and `y` can be any valid Python variable name, `f` can be any valid Python expression, and `unit` should be the unit of the corresponding `x`. You can also include multi-line expressions by appending a `\` character at the end of each line which is continued on the next line (following the Python standard), or comments by including a `#` symbol.

These forms mean:

Differential equation A differential equation with variable `x` which has physical units `unit`. The variable `x` will become one of the state variables of the model.

Equation An equation defining the meaning of `x` can be used for building systems of complicated differential equations.

Alias The variable `x` becomes equivalent to the variable `y`, useful for connecting two separate systems of equations together.

Parameter The variable `x` will have physical units `unit` and will be one of the state variables of the model (but will not evolve dynamically, instead it should be set by the user).

Noise

String equations can also use the reserved term `xi` for a Gaussian white noise with mean 0 and variance 1.

Example usage

```
eqs=Equations('''
dv/dt=(u-v)/tau : volt
u=3*v : volt
w=v
''')
```

Details

For more details, see *More on equations* in the user manual.

For information on integration methods, and the `StateUpdater` class, see *Integration*.

8.4.2 The `NeuronGroup` object

class `NeuronGroup`(*N, model=None, threshold=None, reset=No reset, init=None, refractory=0.0 s, level=0, clock=None, order=1, implicit=False, unit_checking=True, max_delay=0.0 s, compile=False, freeze=False, method=None, **args*)

Group of neurons

Initialised with arguments:

N The number of neurons in the group.

model An object defining the neuron model. It can be a `Model` object, an `Equations` object, a string defining an `Equations` object, a `StateUpdater` object, or a list or tuple of `Equations` and strings.

threshold=None A `Threshold` object, a function, a scalar quantity or a string. If `threshold` is a function with one argument, it will be converted to a `SimpleFunThreshold`, otherwise it will be a `FunThreshold`. If `threshold` is a scalar, then a constant single valued threshold with that value will be used. In this case, the variable to apply the threshold to will be guessed. If there is only one variable, or if you have a variable named one of `V`, `Vm`, `v` or `vm` it will be used. If `threshold` is a string then the appropriate threshold type will be chosen, for example you could do `threshold='V>10*mV'`. The string must be a one line string.

reset=None A `Reset` object, a function, a scalar quantity or a string. If it's a function, it will be converted to a `FunReset` object. If it's a scalar, then a constant single valued reset with that value will be used. In this case, the variable to apply the reset to will be guessed. If there is only one variable, or if you have a variable named one of `V`, `Vm`, `v` or `vm` it will be used. If `reset` is a string it should be a series of expressions which are evaluated for each neuron that is resetting. The series of expressions can be multiline or separated by a semicolon. For example, `reset='Vt+=5*mV; V=Vt'`. Statements involving `if` constructions will often not work because the code is automatically vectorised. For such constructions, use a function instead of a string.

refractory=0*ms A refractory period, used in combination with the `reset` value if it is a scalar.

clock A clock to use for scheduling this `NeuronGroup`, if omitted the default clock will be used.

order=1 The order to use for nonlinear differential equation solvers. TODO: more details.

implicit=False Whether to use an implicit method for solving the differential equations. TODO: more details.

max_delay=0*ms The maximum allowable delay (larger values use more memory). This doesn't usually need to be specified because Connections will update it.

compile=False Whether or not to attempt to compile the differential equation solvers (into Python code). Typically, for best performance, both `compile` and `freeze` should be set to `True` for nonlinear differential equations.

freeze=False If True, parameters are replaced by their values at the time of initialization.

method=None If not None, the integration method is forced. Possible values are linear, nonlinear, Euler, exponential_Euler (overrides implicit and order keywords).

unit_checking=True Set to `False` to bypass unit-checking.

Methods

subgroup(*N*)
Returns the next sequential subgroup of `N` neurons. See the section on subgroups below.

state(*var*)
Returns the array of values for state variable `var`, with length the number of neurons in the group.

rest()
Sets the neuron state values at rest for their differential equations.

The following usages are also possible for a group `G`:

G[i:j] Returns the subgroup of neurons from `i` to `j`.

len(G) Returns the number of neurons in `G`.

G.x For any valid Python variable name `x` corresponding to a state variable of the the `NeuronGroup`, this returns the array of values for the state variable `x`, as for the `state()` method above.

Subgroups

A subgroup is a view on a group. It isn't a new group, it's just a convenient way of referring to a subset of the neurons in an already defined group. The subset has to be a continguous set of neurons. They can be overlapping

if defined with the slice notation, or consecutive if defined with the `subgroup()` method. Subgroups can themselves be subgrouped. Subgroups can be used in almost all situations exactly as if they were groups, except that they cannot be passed to the `Network` object.

Details

TODO: details of other methods and properties for people wanting to write extensions?

class Model (***kwds*)

Stores properties that define a model neuron

NOTE: this class has been deprecated as of Brian 1.1

The purpose of this class is to store the parameters that define a model neuron, but not actually create any neurons themselves. That is done by the `NeuronGroup` object. The parameters for initialising this object are the same as for `NeuronGroup` less `N` and with the addition of `equation` and `equations` as alternative keywords for `model` (for readability of code).

At the moment, this object simply stores a copy of these keyword assignments and passes them on to a `NeuronGroup` when you instantiate it with this model, so the definitive reference point is the `NeuronGroup`. For convenience, we include a copy of these arguments to initiate a `Model` here:

model, equation or equations An object defining the neuron model. It can be an `Equations` object, a string defining an `Equations` object, a `StateUpdater` object, or a list or tuple of `Equations` and strings.

threshold=None A `Threshold` object, a function or a scalar quantity. If `threshold` is a function with one argument, it will be converted to a `SimpleFunThreshold`, otherwise it will be a `FunThreshold`. If `threshold` is a scalar, then a constant single valued threshold with that value will be used. In this case, the variable to apply the threshold to will be guessed. If there is only one variable, or if you have a variable named one of `V`, `Vm`, `v` or `vm` it will be used.

reset=None A `Reset` object, a function or a scalar quantity. If it's a function, it will be converted to a `FunReset` object. If it's a scalar, then a constant single valued reset with that value will be used. In this case, the variable to apply the reset to will be guessed. If there is only one variable, or if you have a variable named one of `V`, `Vm`, `v` or `vm` it will be used.

refractory=0*ms A refractory period, used in combination with the `reset` value if it is a scalar.

order=1 The order to use for nonlinear differential equation solvers. TODO: more details.

implicit=False Whether to use an implicit method for solving the differential equations. TODO: more details.

max_delay=0*ms The maximum allowable delay (larger values use more memory). TODO: more details.

compile=False Whether or not to attempt to compile the differential equation solvers into `C++` code.

freeze=False If True, parameters are replaced by their values at the time of initialization.

Usage

You can either pass a `Model` as an argument to initialise a `NeuronGroup` or initialise a `NeuronGroup` by writing:

```
group = model * N
```

to create a `NeuronGroup` of `N` neurons based on that model.

Example

Starting with a model defined like this:

```
model = Model(equations='''
dv/dt = (ge+gi-(v+49*mV))/(20*ms) : volt
dge/dt = -ge/(5*ms) : volt
dgi/dt = -gi/(10*ms) : volt
''', threshold=-50*mV, reset=-60*mV)
```

The following two lines are equivalent:

```
P = NeuronGroup(4000, model=model)
P = 4000*model
```

8.4.3 Resets

Reset objects are called each network update step to reset specified state variables of neurons that have fired.

class **Reset**(*resetvalue=0.0 V, state=0*)
Resets specified state variable to a fixed value

Initialise as:

```
R = Reset([resetvalue=0*mvolt[, state=0]])
```

with arguments:

resetvalue The value to reset to.

state The name or number of the state variable to reset.

This will reset all of the neurons that have just spiked. The given state variable of the neuron group will be set to value `resetvalue`.

class **StringReset**(*expr, level=0*)
Reset defined by a string

Initialised with arguments:

expr The string expression used to reset. This can include multiple lines or statements separated by a semicolon. For example, `'V=-70*mV'` or `'V=-70*mV; Vt+=10*mV'`.

level How many levels up in the calling sequence to look for names in the namespace. Usually 0 for user code.

class **VariableReset**(*resetvaluestate=1, state=0*)
Resets specified state variable to the value of another state variable

Initialised with arguments:

resetvaluestate The state variable which contains the value to reset to.

state The name or number of the state variable to reset.

This will reset all of the neurons that have just spiked. The given state variable of the neuron group will be set to the value of the state variable `resetvaluestate`.

class **Refractoriness**(*resetvalue=0.0 V, period=5.0 ms, state=0*)
Holds the state variable at the reset value for a fixed time after a spike.

Initialised as:

```
Refractoriness([resetvalue=0*mV[,period=5*ms[,state=0]]])
```

with arguments:

resetvalue The value to reset and hold to.

period The length of time to hold at the reset value.

state The name or number of the state variable to reset and hold.

class `SimpleCustomRefractoriness`(*resetfun, period=5.0 ms, state=0*)
Holds the state variable at the custom reset value for a fixed time after a spike.

Initialised as:

```
SimpleCustomRefractoriness(resetfunc[,period=5*ms[,state=0]])
```

with arguments:

`resetfun` The custom reset function `resetfun(P, spikes)` for `P` a `NeuronGroup` and `spikes` a list of neurons that fired spikes.

`period` The length of time to hold at the reset value.

`state` The name or number of the state variable to reset and hold, it is your responsibility to check that this corresponds to the custom reset function.

The assumption is that `resetfun(P, spikes)` will reset the state variable `state` on the group `P` for the spikes with indices `spikes`. The values assigned by the custom reset function are stored by this object, and they are clamped at these values for `period`. This object does not introduce refractoriness for more than the one specified variable `state` or for spike indices other than those in the variable `spikes` passed to the custom reset function.

class `CustomRefractoriness`(*resetfun, period=5.0 ms, refracfunc=None*)
Holds the state variable at the custom reset value for a fixed time after a spike.

Initialised as:

```
CustomRefractoriness(resetfunc[,period=5*ms[,refracfunc=resetfunc]])
```

with arguments:

`resetfunc` The custom reset function `resetfunc(P, spikes)` for `P` a `NeuronGroup` and `spikes` a list of neurons that fired spikes.

`refracfunc` The custom refractoriness function `refracfunc(P, indices)` for `P` a `NeuronGroup` and `indices` a list of neurons that are in their refractory periods. In some cases, you can choose not to specify this, and it will use the reset function.

`period` The length of time to hold at the reset value.

class `FunReset`(*resetfun*)
A reset with a user-defined function.

Initialised as:

```
FunReset(resetfun)
```

with argument:

`resetfun` A function `f(G,spikes)` where `G` is the `NeuronGroup` and `spikes` is an array of the indexes of the neurons to be reset.

class `NoReset`()
Absence of reset mechanism.

Initialised as:

```
NoReset()
```

8.4.4 Thresholds

A threshold mechanism checks which neurons have fired a spike.

class `Threshold`(*threshold=1.0 mV, state=0*)
All neurons with a specified state variable above a fixed value fire a spike.

Initialised as:

```
Threshold([threshold=1*mV[,state=0])
```

with arguments:

`threshold` The value above which a neuron will fire.

`state` The state variable which is checked.

Compilation

Note that if the global variable `useweave` is set to `True` then this function will use a `C++` accelerated version which runs approximately 3x faster.

class `StringThreshold`(*expr, level=0*)
A threshold specified by a string expression.

Initialised with arguments:

`expr` The expression used to test whether a neuron has fired a spike. Should be a single statement that returns a value. For example, `'V>50*mV'` or `'V>Vt'`.

`level` How many levels up in the calling sequence to look for names in the namespace. Usually 0 for user code.

class `VariableThreshold`(*threshold_state=1, state=0*)
Threshold mechanism where one state variable is compared to another.

Initialised as:

```
VariableThreshold([threshold_state=1[,state=0]])
```

with arguments:

`threshold_state` The state holding the lower bound for spiking.

`state` The state that is checked.

If `x` is the value of state variable `threshold_state` on neuron `i` and `y` is the value of state variable `state` on neuron `i` then neuron `i` will fire if `y>x`.

Typically, using this class is more time efficient than writing a custom thresholding operation.

Compilation

Note that if the global variable `useweave` is set to `True` then this function will use a `C++` accelerated version.

class `EmpiricalThreshold`(*threshold=1.0 mV, refractory=1.0 ms, state=0, clock=None*)
Empirical threshold, e.g. for Hodgkin-Huxley models.

In empirical models such as the Hodgkin-Huxley method, after a spike neurons are not instantaneously reset, but reset themselves as part of the dynamical equations defining their behaviour. This class can be used to model that. It is a simple threshold mechanism that checks e.g. `V>=Vt` but it only does so for neurons that haven't recently fired (giving the dynamical equations time to reset the values naturally). It should be used in conjunction with the `NoReset` object.

Initialised as:

```
EmpiricalThreshold([threshold=1*mV[,refractory=1*ms[,state=0[,clock]]]])
```

with arguments:

threshold The lower bound for the state variable to induce a spike.

refractory The time to wait after a spike before checking for spikes again.

state The name or number of the state variable to check.

clock If this object is being used for a NeuronGroup which doesn't use the default clock, you need to specify its clock here.

class SimpleFunThreshold(*thresholdfun, state=0*)
Threshold mechanism with a user-specified function.

Initialised as:

```
FunThreshold(thresholdfun[,state=0])
```

with arguments:

thresholdfun A function with one argument, the array of values for the specified state variable. For efficiency, this is a numpy array, and there is no unit checking.

state The name or number of the state variable to pass to the threshold function.

Sample usage:

```
FunThreshold(lambda V:V>=Vt,state='V')
```

class FunThreshold(*thresholdfun*)
Threshold mechanism with a user-specified function.

Initialised as:

```
FunThreshold(thresholdfun)
```

where `thresholdfun` is a function with one argument, the 2d state value array, where each row is an array of values for one state, of length N for N the number of neurons in the group. For efficiency, data are numpy arrays and there is no unit checking.

Note: if you only need to consider one state variable, use the SimpleFunThreshold object instead.

class NoThreshold()
No thresholding mechanism.

Initialised as:

```
NoThreshold()
```

8.5 Integration

See *Numerical integration* for an overview.

8.5.1 StateUpdaters

Typically you don't need to worry about `StateUpdater` objects because they are automatically created from the differential equations defining your model. TODO: more details about this.

class `LinearStateUpdater` (*M, B=None, clock=None*)
A linear model with dynamics dX/dt = M(X-B) or dX/dt = MX.

Initialised as:

```
LinearStateUpdater(M[,B[,clock]])
```

with arguments:

`M` Matrix defining the differential equation.

`B` Optional linear term in the differential equation.

`clock` Optional clock.

Computes an update matrix A=exp(M dt) for the linear system, and performs the update step.

TODO: more mathematical details?

class `LazyStateUpdater` (*numstatevariables=1, clock=None*)
A StateUpdater that does nothing.

Initialised as:

```
LazyStateUpdater([numstatevariables=1[,clock]])
```

with arguments:

`numstatevariables` The number of state variables to create.

`clock` An optional clock to determine when it updates, although the update function does nothing so...

TODO: write docs for these StateUpdaters:

- StateUpdater, LinearStateUpdater more details, NonlinearStateUpdater, NonlinearStateUpdater2, ExponentialEulerStateUpdater, NonlinearStateUpdaterRK2, NonlinearStateUpdaterBE, SynapticNoise

8.6 Standard Groups

Some standard types of `NeuronGroup` have already been defined. `PoissonGroup` to generate spikes with Poisson statistics, `PulsePacket` to generate pulse packets with specified parameters, `SpikeGeneratorGroup` and `MultipleSpikeGeneratorGroup` to generate spikes which fire at prespecified times.

class `PoissonGroup` (*N, rates=0.0 Hz, clock=None*)
A group that generates independent Poisson spike trains.

Initialised as:

```
PoissonGroup(N,rates[,clock])
```

with arguments:

`N` The number of neurons in the group

`rates` A scalar, array or function returning a scalar or array. The array should have the same length as the number of neurons in the group. The function should take one argument `t` the current simulation time.

`clock` The clock which the group will update with, do not specify to use the default clock.

class PulsePacket(*t, n, sigma, clock=None*)
Fires a Gaussian distributed packet of n spikes with given spread

Initialised as:

```
PulsePacket(t,n,sigma[,clock])
```

with arguments:

t The mean firing time

n The number of spikes in the packet

sigma The standard deviation of the firing times.

clock The clock to use (omit to use default or local clock)

Methods

This class is derived from SpikeGeneratorGroup and has all its methods as well as one additional method:

generate(*t, n, sigma*)
Change the parameters and/or generate a new pulse packet.

class SpikeGeneratorGroup(*N, spiketimes, clock=None, period=None*)
Emits spikes at given times

Initialised as:

```
SpikeGeneratorGroup(N,spiketimes[,clock[,period]])
```

with arguments:

N The number of neurons in the group.

spiketimes An object specifying which neurons should fire and when. It can be a container such as a `list`, containing tuples `(i,t)` meaning neuron `i` fires at time `t`, or a callable object which returns such a container (which allows you to use generator objects, see below). If `spiketimes` is not a list or tuple, the pairs `(i,t)` need to be sorted in time. You can also pass a numpy array `spiketimes` where the first column of the array is the neuron indices, and the second column is the times in seconds. WARNING: units are not checked in this case, and you need to ensure that the spikes are sorted.

clock An optional clock to update with (omit to use the default clock).

period Optionally makes the spikes recur periodically with the given period. Note that iterator objects cannot be used as the `spikelist` with a period as they cannot be reinitialised.

Sample usages

The simplest usage would be a list of pairs `(i,t)`:

```
spiketimes = [(0,1*ms), (1,2*ms)]
SpikeGeneratorGroup(N,spiketimes)
```

A more complicated example would be to pass a generator:

```
import random
def nextspike():
    nexttime = random.uniform(0*ms,10*ms)
    while True:
        yield (random.randint(0,9),nexttime)
        nexttime = nexttime + random.uniform(0*ms,10*ms)
P = SpikeGeneratorGroup(10,nextspike())
```

This would give a neuron group `P` with 10 neurons, where a random one of the neurons fires at an average rate of one every 5ms.

Notes

Note that if a neuron fires more than one spike in a given interval `dt`, additional spikes will be discarded. If you want them to stack, consider using the less efficient `MultipleSpikeGeneratorGroup` object instead. A warning will be issued if this is detected.

Also note that if you pass a generator, then reinitialising the group will not have the expected effect because a generator object cannot be reinitialised. Instead, you should pass a callable object which returns a generator. In the example above, that would be done by calling:

```
P = SpikeGeneratorGroup(10,nextspike)
```

Whenever P is reinitialised, it will call `nextspike()` to create the required spike container.

class `MultipleSpikeGeneratorGroup`(*spiketimes, clock=None, period=None*)

Emits spikes at given times

Initialised as:

```
MultipleSpikeGeneratorGroup(spiketimes[,clock[,period]])
```

with arguments:

`spiketimes` a list of spike time containers, one for each neuron in the group, although note that elements of `spiketimes` can also be callable objects which return spike time containers if you want to be able to reinitialise (see below). At it's simplest, `spiketimes` could be a list of lists, where `spiketimes[0]` contains the firing times for neuron 0, `spiketimes[1]` for neuron 1, etc. But, any iterable object can be passed, so `spiketimes[0]` could be a generator for example. Each spike time container should be sorted in time. If the containers are numpy arrays units will not be checked (times should be in seconds).

`clock` A clock, if omitted the default clock will be used.

`period` Optionally makes the spikes recur periodically with the given period. Note that iterator objects cannot be used as the `spikelist` with a period as they cannot be reinitialised.

Note that if two or more spike times fall within the same `dt`, spikes will stack up and come out one per dt until the stack is exhausted. A warning will be generated if this happens.

Also note that if you pass a generator, then reinitialising the group will not have the expected effect because a generator object cannot be reinitialised. Instead, you should pass a callable object which returns a generator, this will be called each time the object is reinitialised by calling the `reinit()` method.

Sample usage:

```
spiketimes = [[1*msecond, 2*msecond]]
P = MultipleSpikeGeneratorGroup(spiketimes)
```

8.7 Connections

The best way to understand the concept of a `Connection` in Brian is to work through Tutorial 2: Connections.

class `Connection`(*source, target, state=0, delay=0.0 s, modulation=None, structure='sparse', weight=None, sparseness=None, max_delay=5.0 ms, **kwds*)

Mechanism for propagating spikes from one group to another

A Connection object declares that when spikes in a source group are generated, certain neurons in the target group should have a value added to specific states. See Tutorial 2: Connections to understand this better.

With arguments:

source The group from which spikes will be propagated.

target The group to which spikes will be propagated.

state The state variable name or number that spikes will be propagated to in the target group.

delay The delay between a spike being generated at the source and received at the target. Depending on the type of `delay` it has different effects. If `delay` is a scalar value, then the connection will be initialised with all neurons having that delay. For very long delays, this may raise an error. If `delay=True` then the connection will be initialised as a `DelayConnection`, allowing heterogeneous delays (a different delay for each synapse). `delay` can also be a pair `(min,max)` or a function of one or two variables, in both cases it will be initialised as a `DelayConnection`, see the documentation for that class for details. Note that in these cases, initialisation of delays will only have the intended effect if used with the `weight` and `sparseness` arguments below.

max_delay If you are using a connection with heterogeneous delays, specify this to set the maximum allowed delay (smaller values use less memory). The default is 5ms.

modulation The state variable name from the source group that scales the synaptic weights (for short-term synaptic plasticity).

structure Data structure: `sparse` (default), `dense` or `dynamic`. See below for more information on structures.

weight If specified, the connection matrix will be initialised with values specified by `weight`, which can be any of the values allowed in the methods *connect*'* below.

sparseness If `weight` is specified and `sparseness` is not, a full connection is assumed, otherwise random connectivity with this level of sparseness is assumed.

Methods

connect_random(P,Q,p[,weight=1[,fixed=False[,seed=None]]]) Connects each neuron in `P` to each neuron in `Q` with independent probability `p` and weight `weight` (this is the amount that gets added to the target state variable). If `fixed` is True, then the number of presynaptic neurons per neuron is constant. If `seed` is given, it is used as the seed to the random number generators, for exactly repeatable results.

connect_full(P,Q[,weight=1]) Connect every neuron in `P` to every neuron in `Q` with the given weight.

connect_one_to_one(P,Q) If `P` and `Q` have the same number of neurons then neuron `i` in `P` will be connected to neuron `i` in `Q` with weight 1.

connect(P,Q,W) You can specify a matrix of weights directly (can be in any format recognised by NumPy). Note that due to internal implementation details, passing a full matrix rather than a sparse one may slow down your code (because zeros will be propagated as well as nonzero values). **WARNING:** No unit checking is done at the moment.

Additionally, you can directly access the matrix of weights by writing:

```
C = Connection(P,Q)
print C[i,j]
C[i,j] = ...
```

Where here `i` is the source neuron and `j` is the target neuron. Note: if `C[i,j]` should be zero, it is more efficient not to write `C[i,j]=0`, if you write this then when neuron `i` fires all the targets will have the value 0 added to them rather than just the nonzero ones. **WARNING:** No unit checking is currently done if you use this method. Take care to set the right units.

Connection matrix structures

Brian currently features three types of connection matrix structures, each of which is suited for different situations. Brian has two stages of connection matrix. The first is the construction stage, used for building a weight

matrix. This stage is optimised for the construction of matrices, with lots of features, but would be slow for runtime behaviour. Consequently, the second stage is the connection stage, used when Brian is being run. The connection stage is optimised for run time behaviour, but many features which are useful for construction are absent (e.g. the ability to add or remove synapses). Conversion between construction and connection stages is done by the `compress()` method of `Connection` which is called automatically when it is used for the first time.

The structures are:

dense A dense matrix. Allows runtime modification of all values. If connectivity is close to being dense this is probably the most efficient, but in most cases it is less efficient. In addition, a dense connection matrix will often do the wrong thing if using STDP. Because a synapse will be considered to exist but with weight 0, STDP will be able to create new synapses where there were previously none. Memory requirements are `8NM` bytes where `(N,M)` are the dimensions. (A `double` float value uses 8 bytes.)

sparse A sparse matrix. See `SparseConnectionMatrix` for details on implementation. This class features very fast row access, and slower column access if the `column_access=True` keyword is specified (making it suitable for learning algorithms such as STDP which require this). Memory requirements are 12 bytes per nonzero entry for row access only, or 20 bytes per nonzero entry if column access is specified. Synapses cannot be created or deleted at runtime with this class (although weights can be set to zero).

dynamic A sparse matrix which allows runtime insertion and removal of synapses. See `DynamicConnectionMatrix` for implementation details. This class features row and column access. The row access is slower than for `sparse` so this class should only be used when insertion and removal of synapses is crucial. Memory requirements are 24 bytes per nonzero entry. However, note that more memory than this may be required because memory is allocated using a dynamic array which grows by doubling its size when it runs out. If you know the maximum number of nonzero entries you will have in advance, specify the `nnzmax` keyword to set the initial size of the array.

Advanced information

The following methods are also defined and used internally, if you are writing your own derived connection class you need to understand what these do.

propagate(spikes) Action to take when source neurons with indices in `spikes` fired.

do_propagate() The method called by the `Network` `update()` step, typically just propagates the spikes obtained by calling the `get_spikes` method of the `source` `NeuronGroup`.

class DelayConnection(*source, target, state=0, modulation=None, structure='sparse', weight=None, sparseness=None, delay=None, max_delay=5.0 ms, **kwds*)

Connection which implements heterogeneous postsynaptic delays

Initialised as for a `Connection`, but with the additional keyword:

max_delay Specifies the maximum delay time for any neuron. Note, the smaller you make this the less memory will be used.

Overrides the following attribute of `Connection`:

delay
A matrix of delays. This array can be changed during a run, but at no point should it be greater than `max_delay`.

In addition, the methods `connect`, `connect_random`, `connect_full`, and `connect_one_to_one` have a new keyword `delay=...` for setting the initial values of the delays, where `delay` can be one of:

- A float, all delays will be set to this value
- A pair (min, max), delays will be uniform between these two values.
- A function of no arguments, will be called for each nonzero entry in the weight matrix.
- A function of two argument `(i,j)` will be called for each nonzero entry in the weight matrix.

•A matrix of an appropriate type (e.g. ndarray or lil_matrix).

Finally, there is a method:

set_delays(delay) Where `delay` must be of one of the types above.

Notes

This class implements post-synaptic delays. This means that the spike is propagated immediately from the presynaptic neuron with the synaptic weight at the time of the spike, but arrives at the postsynaptic neuron with the given delay. At the moment, Brian only provides support for presynaptic delays if they are homogeneous, using the `delay` keyword of a standard `Connection`.

Implementation

`DelayConnection` stores an array of size `(n,m)` where `n` is `max_delay/dt` for `dt` of the target `NeuronGroup`'s clock, and `m` is the number of neurons in the target. This array can potentially be quite large. Each row in this array represents the array that should be added to the target state variable at some particular future time. Which row corresponds to which time is tracked using a circular indexing scheme.

When a spike from neuron `i` in the source is encountered, the delay time of neuron `i` is looked up, the row corresponding to the current time plus that delay time is found using the circular indexing scheme, and then the spike is propagated to that row as for a standard connection (although this won't be propagated to the target until a later time).

Warning

If you are using a dynamic connection matrix, it is your responsibility to ensure that the nonzero entries of the weight matrix and the delay matrix exactly coincide. This is not an issue for sparse or dense matrices.

class IdentityConnection(*source, target, state=0, weight=1, delay=0.0 s*)

A `Connection` between two groups of the same size, where neuron `i` in the source group is connected to neuron `i` in the target group.

Initialised with arguments:

source, target The source and target `NeuronGroup` objects.

state The target state variable.

weight The weight of the synapse, must be a scalar.

delay Only homogeneous delays are allowed.

The benefit of this class is that it has no storage requirements and is optimised for this special case.

8.7.1 Connection matrix types

class ConnectionMatrix()

Base class for connection matrix objects

Connection matrix objects support a subset of the following methods:

get_row(i), get_col(i) Returns row/col `i` as a `DenseConnectionVector` or `SparseConnectionVector` as appropriate for the class.

set_row(i, val), set_col(i, val) Sets row/col with an array, `DenseConnectionVector` or `SparseConnectionVector` (if supported).

get_element(i, j), set_element(i, j, val) Gets or sets a single value.

get_rows(rows) Returns a list of rows, should be implemented without Python function calls for efficiency if possible.

insert(i,j,x), remove(i,j) For sparse connection matrices which support it, insert a new entry or remove an existing one.

getnnz() Return the number of nonzero entries.

todense() Return the matrix as a dense array.

The `__getitem__` and `__setitem__` methods are implemented by default, and automatically select the appropriate methods from the above in the cases where the item to be got or set is of the form `:`, `i,:`, `:,j` or `i,j`.

class DenseConnectionMatrix(*val, **kwds*)

Dense connection matrix

See documentation for `ConnectionMatrix` for details on connection matrix types.

This matrix implements a dense connection matrix. It is just a numpy array. The `get_row` and `get_col` methods return `DenseConnectionVector`` objects.

class SparseConnectionMatrix(*val, column_access=True*)

Sparse connection matrix

See documentation for `ConnectionMatrix` for details on connection matrix types.

This class implements a sparse matrix with a fixed number of nonzero entries. Row access is very fast, and if the `column_access` keyword is `True` then column access is also supported (but is not as fast as row access).

The matrix should be initialised with a scipy sparse matrix.

The `get_row` and `get_col` methods return `SparseConnectionVector` objects. In addition to the usual slicing operations supported, `M[:]=val` is supported, where `val` must be a scalar or an array of length `nnz`.

Implementation details:

The values are stored in an array `alldata` of length `nnz` (number of nonzero entries). The slice `alldata[rowind[i]:rowind[i+1]]` gives the values for row `i`. These slices are stored in the list `rowdata` so that `rowdata[i]` is the data for row `i`. The array `rowj[i]` gives the corresponding column `j` indices. For row access, the memory requirements are 12 bytes per entry (8 bytes for the float value, and 4 bytes for the column indices). The array `allj` of length `nnz` gives the column `j` coordinates for each element in `alldata` (the elements of `rowj` are slices of this array so no extra memory is used).

If column access is being used, then in addition to the above there are lists `coli` and `coldataindices`. For column `j`, the array `coli[j]` gives the row indices for the data values in column `j`, while `coldataindices[j]` gives the indices in the array `alldata` for the values in column `j`. Column access therefore involves a copy operation rather than a slice operation. Column access increases the memory requirements to 20 bytes per entry (4 extra bytes for the row indices and 4 extra bytes for the data indices).

class DynamicConnectionMatrix(*val, nnzmax=None, dynamic_array_const=2, **kwds*)

Dynamic (sparse) connection matrix

See documentation for `ConnectionMatrix` for details on connection matrix types.

This class implements a sparse matrix with a variable number of nonzero entries. Row access and column access are provided, but are not as fast as for `SparseConnectionMatrix`.

The matrix should be initialised with a scipy sparse matrix.

The `get_row` and `get_col` methods return `SparseConnectionVector` objects. In addition to the usual slicing operations supported, `M[:]=val` is supported, where `val` must be a scalar or an array of length `nnz`.

Implementation details

The values are stored in an array `alldata` of length `nnzmax` (maximum number of nonzero entries). This is a dynamic array, see:

http://en.wikipedia.org/wiki/Dynamic_array

You can set the resizing constant with the argument `dynamic_array_const`. Normally the default value 2 is fine but if memory is a worry it could be made smaller.

Rows and column point in to this data array, and the list `rowj` consists of an array of column indices for each row, with `coli` containing arrays of row indices for each column. Similarly, `rowdataind` and `coldataind` consist of arrays of pointers to the indices in the `alldata` array.

8.7.2 Construction matrix types

class ConstructionMatrix()
Base class for construction matrices

A construction matrix is used to initialise and build connection matrices. A `ConstructionMatrix` class has to implement a method `connection_matrix(*args, **kwds)` which returns a `ConnectionMatrix` object of the appropriate type.

class DenseConstructionMatrix(*val, **kwds*)
Dense construction matrix. Essentially just numpy.ndarray.

The `connection_matrix` method returns a `DenseConnectionMatrix` object.

The `__setitem__` method is overloaded so that you can set values with a sparse matrix.

class SparseConstructionMatrix(*arg, **kwds*)
SparseConstructionMatrix is converted to SparseConnectionMatrix.

class DynamicConstructionMatrix(*arg, **kwds*)
DynamicConstructionMatrix is converted to DynamicConnectionMatrix.

8.7.3 Connection vector types

class ConnectionVector()
Base class for connection vectors, just used for defining the interface

ConnectionVector objects are returned by ConnectionMatrix objects when they retrieve rows or columns. At the moment, there are two choices, sparse or dense.

This class has no real function at the moment.

class DenseConnectionVector()
Just a numpy array.

class SparseConnectionVector()
Sparse vector class

A sparse vector is typically a row or column of a sparse matrix. This class can be treated in many cases as if it were just a vector without worrying about the fact that it is sparse. For example, if you write `2*v` it will evaluate to a new sparse vector. There is one aspect of the semantics which is potentially confusing. In a binary operation with a dense vector such as `sv+dv` where `sv` is sparse and `dv` is dense, the result will be a sparse vector with zeros where `sv` has zeros, the potentially nonzero elements of `dv` where `sv` has no entry will be simply ignored. It is for this reason that it is a `SparseConnectionVector` and not a general `SparseVector`, because these semantics make sense for rows and columns of connection matrices but not in general.

Implementation details:

The underlying numpy array contains the values, the attribute `n` is the length of the sparse vector, and `ind` is an array of the indices of the nonzero elements.

8.8 Plasticity

8.8.1 Spike timing dependent plasticity (STDP)

class STDP(*C, eqs, pre, post, wmax=1.#INF, level=0, clock=None, delay_pre=None, delay_post=None*)
Spike-timing-dependent plasticity

Initialised with arguments:

C connection object

eqs differential equations (with units)

pre Python code for presynaptic spikes

post Python code for postsynaptic spikes

wmax maximum weight (default unlimited)

delay_pre presynaptic delay

delay_post postsynaptic delay (backward propagating spike)

Example

```
eqs_stdp = """
dA_pre/dt  = -A_pre/tau_pre   : 1
dA_post/dt = -A_post/tau_post : 1
"""
stdp = STDP(synapses, eqs=eqs_stdp, pre='A_pre+=dA_pre; w+=A_post',
            post='A_post+=dA_post; w+=A_pre', wmax=gmax)
```

Technical details

The equations are split into two groups, pre and post. Two groups are created to carry these variables and to update them (these are implemented as `NeuronGroup` objects). As well as propagating spikes from the source and target of `C` via `C`, spikes are also propagated to the respective groups created. At spike propagation time the weight values are updated.

class ExponentialSTDP(*C, taup, taum, Ap, Am, interactions='all', wmax=None, update='additive', delay_pre=None, delay_post=None*)

Exponential STDP.

Initialised with the following arguments:

taup, taum, Ap, Am Synaptic weight change (relative to the maximum weight wmax):

```
f(s) = Ap*exp(-s/taup) if s >0
f(s) = Am*exp(s/taum) if s <0
```

interactions
- 'all': contributions from all pre-post pairs are added
- 'nearest': only nearest-neighbour pairs are considered
- 'nearest_pre': nearest presynaptic spike, all postsynaptic spikes
- 'nearest_post': nearest postsynaptic spike, all presynaptic spikes

wmax maximum synaptic weight

update
- 'additive': modifications are additive (independent of synaptic weight) (or "hard bounds")
- 'multiplicative': modifications are multiplicative (proportional to w) (or "soft bounds")
- 'mixed': depression is multiplicative, potentiation is additive

See documentation for `STDP` for more details.

8.8.2 Short term plasticity (STP)

class STP(*C, taud, tauf, U*)

Short-term synaptic plasticity, following the Tsodyks-Markram model.

Implements the short-term plasticity model described in Markram et al (1998). Differential signaling via the same axon of neocortical pyramidal neurons, PNAS. Synaptic dynamics is described by two variables x and u, which follow the following differential equations:

```
dx/dt=(1-x)/taud  (depression)
du/dt=(U-u)/tauf  (facilitation)
```

where taud, tauf are time constants and U is a parameter in 0..1. Each presynaptic spike triggers modifications of the variables:

```
u<-u+U*(1-u)
x<-x*(1-u)
```

Synaptic weights are modulated by the product `u*x` (in 0..1) (before update).

Reference:

- Markram et al (1998). "Differential signaling via the same axon of neocortical pyramidal neurons", PNAS.

8.9 Network

The `Network` object stores simulation objects and runs simulations. Usage is described in detail below. For simple scripts, you don't even need to use the `Network` object itself, just directly use the ''magic" functions `run()` and `reinit()` described below.

class `Network` (**args, **kwds*)

Contains simulation objects and runs simulations

Initialised as:

```
Network(...)
```

with `...` any collection of objects that should be added to the `Network`. You can also pass lists of objects, lists of lists of objects, etc. Objects that need to passed to the `Network` object are:

- `NeuronGroup` and anything derived from it such as `PoissonGroup`.
- `Connection` and anything derived from it.
- Any monitor such as `SpikeMonitor` or `StateMonitor`.
- Any network operation defined with the `network_operation()` decorator.

Models, equations, etc. do not need to be passed to the `Network` object.

The most important method is the `run(duration)` method which runs the simulation for the given length of time (see below for details about what happens when you do this).

Example usage:

```
G = NeuronGroup(...)
C = Connection(...)
net = Network(G,C)
net.run(1*second)
```

Methods

`add(...)` Add additional objects after initialisation, works the same way as initialisation.

`run(duration)` Runs the network for the given duration. See below for details about what happens when you do this.

`reinit()` Reinitialises the network, runs each object's `reinit()` and each clock's `reinit()` method (resetting them to 0).

`stop()` Can be called from a `network_operation()` for example to stop the network from running.

`__len__()` Returns the number of neurons in the network.

__call__(obj) Similar to `add`, but you can only pass one object and that object is returned. You would only need this in obscure circumstances where objects needed to be added to the network but were either not stored elsewhere or were stored in a way that made them difficult to extract, for example below the NeuronGroup object is only added to the network if certain conditions hold:

```
net = Network(...)
if some_condition:
    x = net(NeuronGroup(...))
```

What happens when you run

For an overview, see the Concepts chapter of the main documentation.

When you run the network, the first thing that happens is that it checks if it has been prepared and calls the `prepare()` method if not. This just does various housekeeping tasks and optimisations to make the simulation run faster. Also, an update schedule is built at this point (see below).

Now the `update()` method is repeatedly called until every clock has run for the given length of time. After each call of the `update()` method, the clock is advanced by one tick, and if multiple clocks are being used, the next clock is determined (this is the clock whose value of `t` is minimal amongst all the clocks). For example, if you had two clocks in operation, say `clock1` with `dt=3*ms` and `clock2` with `dt=5*ms` then this will happen:

1. `update()` for `clock1`, tick `clock1` to `t=3*ms`, next clock is `clock2` with `t=0*ms`.
2. `update()` for `clock2`, tick `clock2` to `t=5*ms`, next clock is `clock1` with `t=3*ms`.
3. `update()` for `clock1`, tick `clock1` to `t=6*ms`, next clock is `clock2` with `t=5*ms`.
4. `update()` for `clock2`, tick `clock2` to `t=10*ms`, next clock is `clock1` with `t=6*ms`.
5. `update()` for `clock1`, tick `clock1` to `t=9*ms`, next clock is `clock1` with `t=9*ms`.
6. `update()` for `clock1`, tick `clock1` to `t=12*ms`, next clock is `clock2` with `t=10*ms`. etc.

The `update()` method simply runs each operation in the current clock's update schedule. See below for details on the update schedule.

Update schedules

An update schedule is the sequence of operations that are called for each `update()` step. The standard update schedule is:

- Network operations with `when = 'start'`
- Network operations with `when = 'before_groups'`
- Call `update()` method for each `NeuronGroup`, this typically performs an integration time step for the differential equations defining the neuron model.
- Network operations with `when = 'after_groups'`
- Network operations with `when = 'middle'`
- Network operations with `when = 'before_connections'`
- Call `do_propagate()` method for each `Connection`, this typically adds a value to the target state variable of each neuron that a neuron that has fired is connected to. See Tutorial 2: Connections for a more detailed explanation of this.
- Network operations with `when = 'after_connections'`
- Network operations with `when = 'before_resets'`
- Call `reset()` method for each `NeuronGroup`, typically resets a given state variable to a given reset value for each neuron that fired in this update step.
- Network operations with `when = 'after_resets'`
- Network operations with `when = 'end'`

There is one predefined alternative schedule, which you can choose by calling the `update_schedule_groups_resets_connections()` method before running the network for the first time. As the name suggests, the reset operations are done before connections (and the appropriately named network operations are called relative to this rearrangement). You can also define your own update schedule with the `set_update_schedule` method (see that method's API documentation for details). This might be useful for example if you have a sequence of network operations which need to be run in a given order.

network_operation(**args, **kwds*)

Decorator to make a function into a `NetworkOperation`

A `NetworkOperation` is a callable class which is called every time step by the `Network` `run` method. Sometimes it is useful to just define a function which is to be run every update step. This decorator can be used to turn a function into a `NetworkOperation` to be added to a `Network` object.

Example usages

Operation doesn't need a clock:

```
@network_operation
def f():
    ...
```

Automagically detect clock:

```
@network_operation
def f(clock):
    ...
```

Specify a clock:

```
@network_operation(specifiedclock)
def f(clock):
    ...
```

Specify when the network operation is run (default is `'end'`):

```
@network_operation(when='start')
def f():
    ...
```

Then add to a network as follows:

```
net = Network(f,...)
```

class **NetworkOperation**(*function, clock=None, when='end'*)

Callable class for operations that should be called every update step

Typically, you should just use the `network_operation()` decorator, but if you can't for whatever reason, use this. Note: current implementation only works for functions, not any callable object.

Initialisation:

```
NetworkOperation(function[,clock])
```

If your function takes an argument, the clock will be passed as that argument.

The ''magic" functions `run()` and `reinit()` work by searching for objects which could be added to a network, constructing a network with all these objects, and working with that. They are suitable for simple scripts only. If you have problems where objects are unexpectedly not being added to the network, the best thing to do would probably be to just use an explicit `Network` object as above rather than trying to tweak your program to make the magic functions work. However, details are available in the `brian/magic.py` source code.

run(*duration, threads=1*)
Run a network created from any suitable objects that can be found

Usage:

```
run(duration)
```

where `duration` is the length of time to run the network for.

Works by constructing a `MagicNetwork` object from all the suitable objects that could be found (`NeuronGroup`, `Connection`, etc.) and then running that network. Not suitable for repeated runs or situations in which you need precise control.

reinit()
Reinitialises any suitable objects that can be found

Usage:

```
reinit()
```

Works by constructing a `MagicNetwork` object from all the suitable objects that could be found (`NeuronGroup`, `Connection`, etc.) and then calling `reinit()` for each of them. Not suitable for repeated runs or situations in which you need precise control.

stop()
Globally stops any running network, this is reset the next time a network is run

clear(*erase=True*)
Clears all Brian objects.

Specifically, it stops all existing Brian objects from being collected by `MagicNetwork` (objects created after clearing will still be collected). If `erase` is `True` then it will also delete all data from these objects. This is useful in, for example, `ipython` which stores persistent references to objects in any given session, stopping the data and memory from being freed up.

class **MagicNetwork**(*verbose=False, level=1*)
Creates a `Network` object from any suitable objects

Initialised as:

```
MagicNetwork()
```

The object returned can then be used just as a regular `Network` object. It works by finding any object in the ''execution frame" (i.e. in the same function, script or section of module code where the `MagicNetwork` was created) derived from `NeuronGroup`, `Connection` or `NetworkOperation`.

Sample usage:

```
G = NeuronGroup(...)
C = Connection(...)
@network_operation
def f():
    ...
net = MagicNetwork()
```

Each of the objects `G`, `C` and `f` are added to `net`.

Advanced usage:

```
MagicNetwork([verbose=False[,level=1]])
```

with arguments:

verbose Set to `True` to print out a list of objects that were added to the network, for debugging purposes.

level Where to find objects. `level=1` means look for objects where the `MagicNetwork` object was created. The `level` argument says how many steps back in the stack to look.

8.10 Monitors

Monitors are used to record properties of your network. The two most important are `SpikeMonitor` which records spikes, and `StateMonitor` which records values of state variables. These objects are just added to the network like a `NeuronGroup` or `Connection`.

Implementation note: monitors that record spikes are classes derived from `Connection`, and overwrite the `propagate` method to store spikes. If you want to write your own custom spike monitors, you can do the same (or just use `SpikeMonitor` with a custom function). Monitors that record values are classes derived from `NetworkOperation` and implement the `__call__` method to store values each time the network updates. Custom state monitors are most easily written by just writing your own network operation using the `network_operation` decorator.

class SpikeMonitor(*source, record=True, delay=0, function=None*)

Counts or records spikes from a `NeuronGroup`

Initialised as one of:

```
SpikeMonitor(source(,record=True))
SpikeMonitor(source,function=function)
```

Where:

source A `NeuronGroup` to record from

record `True` or `False` to record all the spikes or just summary statistics.

function A function `f(spikes)` which is passed the array of neuron numbers that have fired called each step, to define custom spike monitoring.

Has two attributes:

nspikes The number of recorded spikes

spikes A time ordered list of pairs `(i,t)` where neuron `i` fired at time `t`.

For `M` a `SpikeMonitor`, you can also write:

M[i] A qarray of the spike times of neuron `i`.

Notes:

`SpikeMonitor` is subclassed from `Connection`. To define a custom monitor, either define a subclass and rewrite the `propagate` method, or pass the monitoring function as an argument (`function=myfunction`, with `def myfunction(spikes):...`)

class SpikeCounter(*source*)

Counts spikes from a `NeuronGroup`

Initialised as:

```
SpikeCounter(source)
```

With argument:

source A `NeuronGroup` to record from

Has two attributes:

nspikes The number of recorded spikes

count An array of spike counts for each neuron

For a SpikeCounter M you can also write M[i] for the number of spikes counted for neuron i.

class PopulationSpikeCounter(*source, delay=0*)
Counts spikes from a NeuronGroup

Initialised as:

```
PopulationSpikeCounter(source)
```

With argument:

source A NeuronGroup to record from

Has one attribute:

nspikes The number of recorded spikes

class StateSpikeMonitor(*source, var*)
Counts or records spikes and state variables at spike times from a NeuronGroup

Initialised as:

```
StateSpikeMonitor(source, var)
```

Where:

source A NeuronGroup to record from

var The variable name or number to record from, or a tuple of variable names or numbers if you want to record multiple variables for each spike.

Has two attributes:

nspikes
The number of recorded spikes

spikes
A time ordered list of tuples (i,t,v) where neuron i fired at time t and the specified variable had value v. If you specify multiple variables, each tuple will be of the form (i,t,v0,v1,v2,...) where the vi are the values corresponding in order to the variables you specified in the var keyword.

And two methods:

times(*i=None*)
Returns a qarray of the spike times for the whole monitored group, or just for neuron i if specified.

values(*var, i=None*)
Returns a qarray of the values of variable var for the whole monitored group, or just for neuron i if specified.

class StateMonitor(*P, varname, clock=None, record=False, timestep=1, when='end'*)
Records the values of a state variable from a NeuronGroup.

Initialise as:

```
StateMonitor(P,varname(,record=False)
    (,when='end)(,timestep=1)(,clock=clock))
```

Where:

P The group to be recorded from

varname The state variable name or number to be recorded

record What to record. The default value is `False` and the monitor will only record summary statistics for the variable. You can choose `record=integer` to record every value of the neuron with that number, `record=``list of integers to record every value of each of those neurons, or ``record=True` to record every value of every neuron (although beware that this may use a lot of memory).

when When the recording should be made in the network update, possible values are any of the strings: `'start'`, `'before_groups'`, `'after_groups'`, `'before_connections'`, `'after_connections'`,`'before_resets'`,`'after_resets'`,`'end'` (in order of when they are run).

timestep A recording will be made each timestep clock updates (so `timestep` should be an integer).

clock A clock for the update schedule, use this if you have specified a clock other than the default one in your network, or to update at a lower frequency than the update cycle. Note though that if the clock here is different from the main clock, the when parameter will not be taken into account, as network updates are done clock by clock. Use the `timestep` parameter if you need recordings to be made at a precise point in the network update step.

The `StateMonitor` object has the following properties (where names without an underscore return `QuantityArray` objects with appropriate units and names with an underscore return `array` objects without units):

times, times_ The times at which recordings were made

mean, mean_ The mean value of the state variable for every neuron in the group (not just the ones specified in the `record` keyword)

var, var_ The unbiased estimate of the variances, as in `mean`

std, std_ The square root of `var`, as in `mean`

values, values_ A 2D array of the values of all the recorded neurons, each row is a single neuron's values.

In addition, if `M`` is a `StateMonitor` object, you write:

```
M[i]
```

for the recorded values of neuron `i` (if it was specified with the `record` keyword). It returns a `QuantityArray` object with units. Downcast to an array without units by writing `asarray(M[i])`.

Methods:

plot (*[indices=None]*)
Plots the recorded values using pylab. You can specify an index or list of indices, otherwise all the recorded values will be plotted. The graph plotted will have legends of the form `name[i]` for `name` the variable name, and `i` the neuron index.

class MultiStateMonitor (*G, vars=None, **kwds*)
Monitors multiple state variables of a group

This class is a container for multiple `StateMonitor` objects, one for each variable in the group. You can retrieve individual `StateMonitor` objects using `M[name]` or retrieve the recorded values using `M[name, i]` for neuron `i`.

Initialised with a group `G` and a list of variables `vars`. If `vars` is omitted then all the variables of `G` will be recorded. Any additional keyword argument used to initialise the object will be passed to the individual `StateMonitor` objects (e.g. the `when` keyword).

Methods:

vars() Returns the variables

items(), iteritems() Returns the pairs (var, mon)

plot(indices) Plots all the monitors.

Attributes:

times The times at which recordings were made.

monitors The dictionary of monitors indexed by variable name.

Usage:

```
G = NeuronGroup(N, eqs, ...)
M = MultiStateMonitor(G, record=True)
...
run(...)
...
plot(M['V'].times, M['V'][0])
figure()
for name, m in M.iteritems():
    plot(m.times, m[0], label=name)
legend()
show()
```

class FileSpikeMonitor(*source, filename, record=False, delay=0*)
Records spikes to a file

Initialised as:

```
FileSpikeMonitor(source, filename[, record=False])
```

Does everything that a `SpikeMonitor` does except also records the spikes to the named file. note that spikes are recorded as an ASCII file of lines each of the form:

```
i, t
```

Where `i` is the neuron that fired, and `t` is the time in seconds.

Has one additional method:

close_file() Closes the file manually (will happen automatically when the program ends).

class ISIHistogramMonitor(*source, bins, delay=0*)
Records the interspike interval histograms of a group.

Initialised as:

```
ISIHistogramMonitor(source, bins)
```

source The source group to record from.

bins The lower bounds for each bin, so that e.g. `bins = [0*ms, 10*ms, 20*ms]` would correspond to bins with intervals 0-10ms, 10-20ms and 20+ms.

Has properties:

bins The `bins` array passed at initialisation.

count An array of length `len(bins)` counting how many ISIs were in each bin.

This object can be passed directly to the plotting function `hist_plot()`.

class PopulationRateMonitor(*source, bin=None*)
Monitors and stores the (time-varying) population rate

Initialised as:

`PopulationRateMonitor(source,bin)`

Records the average activity of the group for every bin.

Properties:

rate, rate_ A `qarray` of the rates in Hz.

times, times_ The times of the bins.

bin The duration of a bin (in second).

8.11 Plotting

Most plotting should be done with the PyLab commands, all of which are loaded when you import Brian. See:

http://matplotlib.sourceforge.net/matplotlib.pylab.html

for help on PyLab.

Brian currently defines just two plotting functions of its own, `raster_plot()` and `hist_plot()`.

raster_plot(**monitors, **plotoptions*)

Raster plot of a `SpikeMonitor`

Usage

raster_plot(monitor,options...) Plots the spike times of the monitor on the x-axis, and the neuron number on the y-axis

raster_plot(monitor0,monitor1,...,options...) Plots the spike times for all the monitors given, with y-axis defined by placing a spike from neuron n of m in monitor i at position i+n/m

raster_plot(options...) Guesses the monitors to plot automagically

Options

Any of PyLab options for the `plot` command can be given, as well as:

showplot=False set to `True` to run pylab's `show()` function

newfigure=True set to `False` not to create a new figure with pylab's `figure()` function

xlabel label for the x-axis

ylabel label for the y-axis

title title for the plot

showgrouplines=False set to `True` to show a line between each monitor

grouplinecol colour for group lines

spacebetweengroups value between 0 and 1 to insert a space between each group on the y-axis

hist_plot(*histmon=None, **plotoptions*)

Plot a histogram

Usage

hist_plot(histmon,options...) Plot the given histogram monitor

hist_plot(options...) Guesses which histogram monitor to use

with argument:

histmon is a monitor of histogram type

Notes

Plots only the first n-1 of n bars in the histogram, because the nth bar is for the interval (-,infinity).

Options

Any of PyLab options for bar can be given, as well as:

showplot=False set to `True` to run pylab's `show()` function

newfigure=True set to `False` not to create a new figure with pylab's `figure()` function

xlabel label for the x-axis

ylabel label for the y-axis

title title for the plot

8.12 Analysis

8.12.1 Statistics of spike trains

firing_rate(*spikes*)
Rate of the spike train.

CV(*spikes*)
Coefficient of variation.

correlogram(*T1, T2, width=20.0 ms, bin=1.0 ms, T=None*)
Returns a cross-correlogram with lag in [-width,width] and given bin size. T is the total duration (optional) and should be greater than the duration of T1 and T2. The result is in Hz (rate of coincidences in each bin).

N.B.: units are discarded. TODO: optimise?

autocorrelogram(*T0, width=20.0 ms, bin=1.0 ms, T=None*)
Returns an autocorrelogram with lag in [-width,width] and given bin size. T is the total duration (optional) and should be greater than the duration of T1 and T2. The result is in Hz (rate of coincidences in each bin).

N.B.: units are discarded.

CCF(*T1, T2, width=20.0 ms, bin=1.0 ms, T=None*)
Returns the cross-correlation function with lag in [-width,width] and given bin size. T is the total duration (optional). The result is in Hz**2: CCF(T1,T2)=<T1(t)T2(t+s)>

N.B.: units are discarded.

ACF(*T0, width=20.0 ms, bin=1.0 ms, T=None*)
Returns the autocorrelation function with lag in [-width,width] and given bin size. T is the total duration (optional). The result is in Hz**2: ACF(T0)=<T0(t)T0(t+s)>

N.B.: units are discarded.

CCVF(*T1, T2, width=20.0 ms, bin=1.0 ms, T=None*)
Returns the cross-covariance function with lag in [-width,width] and given bin size. T is the total duration (optional). The result is in Hz**2: CCVF(T1,T2)=<T1(t)T2(t+s)>-<T1><T2>

N.B.: units are discarded.

ACVF(*T0, width=20.0 ms, bin=1.0 ms, T=None*)
Returns the autocovariance function with lag in [-width,width] and given bin size. T is the total duration (optional). The result is in Hz**2: ACVF(T0)=<T0(t)T0(t+s)>-<T0>**2

N.B.: units are discarded.

total_correlation(*T1, T2, width=20.0 ms, T=None*)
Returns the total correlation coefficient with lag in [-width,width]. T is the total duration (optional). The result is a real (typically in [0,1]): total_correlation(T1,T2)=int(CCVF(T1,T2))/rate(T1)

8.13 Magic in Brian

magic_return(*f*)
Decorator to ensure that the returned object from a function is recognised by magic functions

Usage example:

```
@magic_return
def f():
    return PulsePacket(50*ms, 100, 10*ms)
```

Explanation

Normally, code like the following wouldn't work:

```
def f():
    return PulsePacket(50*ms, 100, 10*ms)
pp = f()
M = SpikeMonitor(pp)
run(100*ms)
raster_plot()
show()
```

The reason is that the magic function `run()` only recognises objects created in the same execution frame that it is run from. The `magic_return()` decorator corrects this, it registers the return value of a function with the magic module. The following code will work as expected:

```
@magic_return
def f():
    return PulsePacket(50*ms, 100, 10*ms)
pp = f()
M = SpikeMonitor(pp)
run(100*ms)
raster_plot()
show()
```

Technical details

The `magic_return()` function uses `magic_register()` with the default `level=1` on just the object returned by a function. See details for `magic_register()`.

magic_register(**args, **kwds*)
Declare that a magically tracked object should be put in a particular frame

Standard usage

If `A` is a tracked class (derived from `InstanceTracker`), then the following wouldn't work:

```
def f():
    x = A('x')
    return x
objs = f()
print get_instances(A,0)[0]
```

Instead you write:

```
def f():
    x = A('x')
    magic_register(x)
    return x
objs = f()
print get_instances(A,0)[0]
```

Definition

Call as:

```
magic_register(...[,level=1])
```

The `...` can be any sequence of tracked objects or containers of tracked objects, and each tracked object will have its instance id (the execution frame in which it was created) set to that of its parent (or to its parent at the given level). This is equivalent to calling:

```
x.set_instance_id(level=level)
```

For each object `x` passed to `magic_register()`.

See Also:

Projects with multiple files or functions Describes difficulties and solutions for using magic functions on projects with multiple files or functions.

8.14 Tests

run_all_tests()
Run all of Brian's test functions

CHAPTER

NINE

TYPICAL TASKS

TODO: typical things you want to achieve in running your simulation, and how to go about doing them.

9.1 Projects with multiple files or functions

Brian works with the minimal hassle if the whole of your code is in a single Python module (`.py` file). This is fine when learning Brian or for quick projects, but for larger, more realistic projects with the source code separated into multiple files, there are some small issues you need to be aware of. These issues essentially revolve around the use of the ''magic" functions `run()`, etc. The way these functions work is to look for objects of the required type that have been instantiated (created) in the same ''execution frame" as the `run()` function. In a small script, that is normally just any objects that have been defined in that script. However, if you define objects in a different module, or in a function, then the magic functions won't be able to find them.

There are three main approaches then to splitting code over multiple files (or functions).

9.1.1 Use the `Network` object explicitly

The magic `run()` function works by creating a `Network` object automatically, and then running that network. Instead of doing this automatically, you can create your own `Network` object. Rather than writing something like:

```
group1 = ...
group2 = ...
C = Connection(group1,group2)
...
run(1*second)
```

You do this:

```
group1 = ...
group2 = ...
C = Connection(group1, group2)
...
net = Network(group1, group2, C)
net.run(1*second)
```

In other words, you explicitly say which objects are in your network. Note that any `NeuronGroup`, `Connection`, `Monitor` or function decorated with `network_operation()` should be included in the `Network`. See the documentation for `Network` for more details.

This is the preferred solution for almost all cases. You may want to use either of the following two solutions if you think your code may be used by someone else, or if you want to make it into an extension to Brian.

9.1.2 Use the `magic_return()` decorator or `magic_register()` function

The `magic_return()` decorator is used as follows:

```
@magic_return
def f():
        ...
        return obj
```

Any object returned by a function decorated by `magic_return()` will be considered to have been instantiated in the execution frame that called the function. In other words, the magic functions will find that object even though it was really instantiated in a different execution frame.

In more complicated scenarios, you may want to use the `magic_register()` function. For example:

```
def f():
        ...
        magic_register(obj1, obj2)
        return (obj1, obj2)
```

This does the same thing as `magic_return()` but can be used with multiple objects. Also, you can specify a `level` (see documentation on `magic_register()` for more details).

9.1.3 Use derived classes

Rather than writing a function which returns an object, you could instead write a derived class of the object type. So, suppose you wanted to have an object that emitted N equally spaced spikes, with an interval dt between them, you could use the `SpikeGeneratorGroup` class as follows:

```
@magic_return
def equally_spaced_spike_group(N, dt):
        spikes = [(0,i*dt) for i in range(N)]
        return SpikeGeneratorGroup(spikes)
```

Or alternatively, you could derive a class from `SpikeGeneratorGroup` as follows:

```
class EquallySpacedSpikeGroup(SpikeGeneratorGroup):
        def __init__(self, N, t):
                spikes = [(0,i*dt) for i in range(N)]
                SpikeGeneratorGroup.__init__(self, spikes)
```

You would use these objects in the following ways:

```
obj1 = equally_spaced_spike_group(100, 10*ms)
obj2 = EquallySpacedSpikeGroup(100, 10*ms)
```

For simple examples like the one above, there's no particular benefit to using derived classes, but using derived classes allows you to add methods to your derived class for example, which might be useful. For more experienced Python programmers, or those who are thinking about making their code into an extension for Brian, this is probably the preferred approach. Finally, it may be useful to note that there is a protocol for one object to 'contain' other objects. That is, suppose you want to have an object that can be treated as a simple `NeuronGroup` by the person using it, but actually instantiates several objects (perhaps internal `Connection` objects). These objects need to be added to the `Network` object in order for them to be run with the simulation, but the user shouldn't need to have to know about

them. To this end, for any object added to a `Network`, if it has an attribute `contained_objects`, then any objects in that container will also be added to the network.

CHAPTER

TEN

EXPERIMENTAL FEATURES

The following features are located in the `experimental` package inside Brian, and are subject to change without notice. The most likely changes are ones of syntax and naming, although functionality may also be subject to change.

10.1 Automatic C code generation for nonlinear state updaters

The `brian.experimental.ccodegen` module provides an object `AutoCompiledNonlinearStateUpdater` to automatically convert equations for a nonlinear state updater into C code rather than Python code for performing the state update operation in the case where you are using the Euler method.

The plan is to expand this module so that all nonlinear state updaters will, if weave compilation is enabled, use automatically generated C++ code by default. More speculatively, we are considering whether or not it would be possible to automatically generate all the simulation code automatically. See the `dev/ideas/cppgen` folder on the SVN version of Brian for current progress.

Sample use:

```
eqs = Equations('''
dV/dt = W*W/(100*ms) : 1
dW/dt = -V/(100*ms) : 1
''')
G = NeuronGroup(10, eqs, compile=True, freeze=True)
su = AutoCompiledNonlinearStateUpdater(eqs, G.clock, freeze=True)
G._state_updater = su
    G.V = 1
    run(100*ms)
```

10.2 Multilinear state updater

class **`MultiLinearNeuronGroup`**(*eqs, subs, clock=None, level=0, **kwds*)

Make a NeuronGroup with a linear differential equation for each neuron

You give a single set of differential equations with parameters, the variables you want substituted should be defined as parameters in the equations, but they will not be treated as parameters, instead they will be substituted. You also pass a list of variables to have their values substituted, and these names should exist in the namespace initialising the MultiLinearNeuronGroup.

Arguments:

`eqs` should be the equations, and must be a string not an `Equations` object.

`subs` A list of variables to be substituted with values.

level How many levels up to look for the equations' namespace.

clock If you want.

kwds Any additonal arguments to pass to `NeuronGroup` init.

Example:

```
eqs = '''
dv/dt = k*v/(1*second) : 1
dw/dt = k*w/(1*second) : 1
k : 1
'''
k = array([-1,-2,-3])
subs = ['k']
G = MultiLinearNeuronGroup(eqs, subs)
G.v = 1
G.w = 0
M = StateMonitor(G, 'v', record=True)
run(1*second)
for i in range(len(G)):
    plot(M.times, M[i])
show()
```

MODULE INDEX

B

INDEX

N

P

Q

R

S

T

U

V

W

X