Brian: a pure Python simulator

What is Brian for?
• Quick model coding for every day use
• Easy to learn and intuitive
• Equations-oriented: flexible

What is Brian not for?
• Very large-scale network models (distributed)
• Very detailed biophysical models
from brian import *

eqs = ""
\[ \frac{dv}{dt} = \frac{(ge+gi-(v+49*mV))/(20*ms)}{volt} \]
\[ \frac{dge}{dt} = -ge/(5*ms) : volt \]
\[ \frac{dgi}{dt} = -gi/(10*ms) : volt \]

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')
Ci = Connection(Pi, P, 'gi')
Ce.connectRandom(Pe, P, 0.02, weight=1.62*mV)
Ci.connectRandom(Pi, P, 0.02, weight=-9*mV)

M = SpikeMonitor(P)
run(1*second)
rasterPlot(M)
show()
How it works

- Synchronous operations are implemented as vector operations (Scipy)
- Cost of each vector-based operation = scales as N
- Cost of interpretation = constant = negligible for large networks

**Neuron groups**

```python
eqs = """"  
dv/dt = (ge+gi-(v+49*mV))/(20*ms)  
dge/dt = -ge/(5*ms)  
dgi/dt = -gi/(10*ms)"""

P = NeuronGroup(4000, model=eqs, threshold=-50*mV, reset=-60*mV)
```

Update matrix A

State matrix S

<table>
<thead>
<tr>
<th></th>
<th>v</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ge</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How it works (2)

\[ P_v = -60\text{mV} \]

\[ P_e = P \text{.subgroup}(3200) \]

\[ P_i = P \text{.subgroup}(800) \]

\[ Ce = \text{Connection}(P_e, P, 'ge') \]

\[ Ci = \text{Connection}(P_i, P, 'gi') \]
How it works (3)

```python
Ce.connectRandom(Pe, P, 0.02, weight=1.62*mV)
Ci.connectRandom(Pi, P, 0.02, weight=-9*mV)
```

For $i$ in `xrange(len(Pe))`:
- $k = random.binomial(m, p, 1)[0]$
- `W.rows[i] = sample(xrange(m), k)`
- `W.data[i] = [value] * k`

Vector-based construction:

Sparse matrix (0s not stored)
(scipy.sparse.lil_matrix)
M = SpikeMonitor(P)
run(1*second)

Clock-based simulation

3. Update state matrix: $S = \text{dot}(A, S)$

5. Check threshold: $\text{spikes} = (S[0, :] > vt).\text{nonzero()}[0]$

7. Propagate spikes: $S[1, :] += W[\text{spikes}, :]$

9. Reset: $S[0, \text{spikes}] = vr$

+ user-defined operations in between

M.spikes += zip(spikes, repeat(t))

(more complicated with sparse W)
Planned features

• STDP (close to finished)
• Gap junctions
• Using the GPU (project with GPULib)
• Automatic code generation
• Static analysis of neural networks
• Distributed simulations?
• Event-driven algorithms?
• Compartmental modelling?
How you can help...

- Improve physical units package
- Job scheduling (e.g. with Condor)
- Plotting and analysis (integration with NeuroTools?)
- User interfaces (e.g. HTML with CherryPy)
- PyNN interface
- Bug analyser (standardisation with PyLint?)
- Magic module (standardisation? Improvements?)
- Visualising networks (using graphviz?)
- Documentation tools (ReST+filters?)
- Or... get more deeply involved and contribute to core Brian features (get in touch!)
Data structures: output

- **StateMonitor**
  - $M.t\text{imes} = \text{qarray of length num steps with units of time}$
  - $M[i] = \text{qarray of length num steps with units of recorded state variable for neuron i}$

- **SpikeMonitor**
  - $M.sp\text{i}kes = \text{Python list of pairs (i, t)}$ [also used as an input data structure]

- **PopulationRateMonitor**
  - $M.t\text{imes} = \text{qarray of length num bins, giving the left edge of the interval, units of time}$
  - $M.r\text{ate} = \text{qarray of corresponding rates in Hz}$
Data structures: input

- **Physical units**
  - Quantity (derived from float)
  - qarray (derived from numpy.ndarray)

- **Equations**
  - \( \frac{dV}{dt} = \frac{-V + V_0 + a \sin(bt)}{\tau} \): volt [diff. equation]
  - \( w = V^2 \): volt2 [equation]
  - \( u = V \) [alias]
  - \( V_0 \): volt [parameter]

- **NeuronGroup of N neurons**
  - \( G.\text{varname} = \) qarray of length N with units of that state variable (defined in Equations)

- **SpikeGeneratorGroup**
  - spiketimes can be a list of pairs (i,t), or a function returning a list of pairs, or a Python generator

- **MultipleSpikeGeneratorGroup**
  - spiketimes is a list of sequences (t0, t1, t2, ...), one for each neuron
Units in Brian: classes

- **float**
- **numpy.ndarray**
  - **Quantity**
    - **unitarray**
      - ndarray with `dtype=object`
    - **homog_unitarray**
      - `[]` overwritten to always return a single value
  - **qarray**
    - has a unitarray or `homog_unitarray` attribute
- **Unit**
- **safeqarray**
Units in Brian: functions

• Some new versions of numpy functions, mostly just wrappers, e.g. 
  \texttt{rand(n)=qarray(numpy.rand(n))}

• Ufuncs: dimensionally consistent arithmetic and many array functions implemented via 
  ufuncs mechanism by overriding the behaviour of \texttt{qarray.__array_wrap__}

• \texttt{qarray methods}: other numpy functions such as mean, std, var, etc. implemented as methods
Units in Brian

Advantages

• Flexible system
• Written in pure Python so will run on any platform
• Transparent: in many cases, works as if you were just using numpy except with units

Disadvantages

• Slow, unusably so in the case of arrays with non-homogeneous units
• Doesn’t work transparently with numpy arrays, e.g. `array(...) * kg = array(...) not qarray(...)`
An alternative system for units

- float
- Quantity
  - Uses a Dimension object
- Unit
- Dimension
- DimensionArray
  - Non-homogeneous
    - Array with one extra dimension with 7 elements
- Mixed?
  - e.g. Each column or row could be separately homogeneous
- qarray
  - Has a DimensionArray attribute
- safeqarray
Ideas for alternative system

• Implement Dimension operations by relations like \( \text{dim}(xy) = \text{dim}(x) + \text{dim}(y) \) and use numpy. Potentially much faster.
• Implement code in C/C++ rather than pure Python.
• Mixed homogeneity of units more flexible but difficult to code.
• Could fork units off as a separate project, maybe even try for inclusion in numpy at some point.
• Possibly better to just have homogeneous units and saferqarray – less ambitious, easier, but similar to existing physical units packages.