Core concepts of BRIAN^2

Marcel Stimberg

Constants

- Constants defined outside of strings can be used inside strings (only scalar values):

```
tau = 10*ms
G = NeuronGroup(1, 'dv/dt = -v / tau : volt')
```

- Values of constants are resolved at the **run** call:

```
tau = 10*ms
G = NeuronGroup(1, 'dv/dt = -v / tau : volt')
run(100*ms)
tau = 20*ms
run(100*ms)
```

will be used for second run

Variables and equations

Equations define *state variables* of an object:

```
tau = 10*ms
G = NeuronGroup(5, '''dv/dt = (-v + inp) / tau                 : volt (unless refractory)
                   inp = a * clip(sin(2*pi*freq*t), 0, inf) : volt
                   freq                                      : Hz   (constant)
                   a                                         : volt (shared)
                ''', threshold='v>20*mV', reset='v = 0*mV',
                refractory=2*ms)
G.freq = [100, 200, 300, 400, 500] * Hz
G.a = 100*mV
G.v = 10*mV
print G.inp  # not a state variable, but can be accessed like one
```

Additional variables are defined automatically

```
>>> print(G.variables.keys())
['a', '_spikespace', 'i', 'inp', 'N', 't', 'v', 'dt', 'lastspike', 'freq', 'not_refractory']
```

neuron index   number of neurons   time step   time of last spike   refractory?

Variables and equations

A Hodgkin-Huxley equations

```
G = NeuronGroup(number_of_neurons,
            '''dv/dt = (I_l+I_Na+I_K)/C_m   : volt # membrane potential
              I_l = g_l*(-v+E_l)              : amp  # passive current
              I_Na = g_Na*(m**3)*h*(-v+E_Na) : amp  # sodium current
              I_K = g_K*(n**4)*(-v+E_K)       : amp  # potassium current
              ...# equations for n, m, h''')
```

special variable provided for noise

B Noisy membrane

```
G = NeuronGroup(number_of_neurons,
            'dv/dt = -(v-v_0)/tau_m +
                    tau_m**-0.5*3*mV*xi : volt # membrane potential')
```

C Leaky integrate-and-fire neuron

```
G = NeuronGroup(number_of_neurons,
            'dv/dt = -(v-v_0)/tau_m : volt # membrane potential',
            threshold='v > v_th', reset='v = v_0')
```

D Leaky integrate-and-fire neuron with adaptive threshold

```
G = NeuronGroup(number_of_neurons,
            '''dv/dt = -(v-v_0)/tau_m          : volt # membrane potential
              dv_th/dt = -(v_th-v_th0)/tau_th : volt # threshold''',
            threshold='v > v_th', reset='''v = v_0
                                           v_th += 3*mV''')
```

Expressions

- Can be used to
  - Specify conditions (threshold, refractoriness, synaptic connection)

```
G = NeuronGroup(10, 'dv/dt = -v / tau : 1 (unless refractory)',
                threshold='v > 1', refractory='(t-lastspike) < 3*ms')
```

  - Index and set state variables

```
min_freq = 100*Hz
print G.v['freq > min_freq']
G.v = '0*mV + rand()*10*mV'
```

- Can refer to state variables, constants, units

Defining synaptic connections with string expressions

Full connectivity:

```
S.connect('True')
```

Condition for
a connection to exist

One-to-one connectivity:

```
S.connect('i == j')
```

Convergent connectivity:

```
S.connect('(i/N) == j')
```

Ring structure, connecting to the immediate neighbours:

```
S.connect('abs((i - j + N/2)%N - N/2) == 1')
```

Connections to 2d neighbourhood:

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

Sparse random connectivity without self-connections:

Probability for a connection

```
S.connect('i !=j', p=0.1)
```

Random connectivity to a 2d neighbourhood without self-connections:

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

One-to-one connectivity with two synapses per connection:

Number of synapses
per connection

```
S.connect('i == j', n=2)
```

Abstract code statements

- Event-triggered operations (reset, synaptic event) are specified as *abstract code:*

```
G = NeuronGroup(..., reset='v = E_L')
S = Synapses(..., pre='v+=w')
update = G.custom_operation('stim = rand()')
```

- Again: can refer to state variables, constant, units
- Only assignments (and "+=" etc.) allowed
- Automatically interpreted as referring to all "relevant" elements of a group (neurons that spiked for reset, synapses that received a pre-synaptic spike for "pre", all neurons/synapses for custom operations)

Functions

- Abstract code ≠ Python code
- Functions have to be explicitly supported
- Built-in functions:
  - Random numbers: `rand(), randn()`
  - Elementary functions: `sqrt, exp, log, log10, abs`
  - Trigonometric functions: `sin, cos, tan, sinh, cosh, tanh, arcsin, arccos, arctan`
  - General utility functions: `clip, floor, ceil`
  - Boolean → integer: `int`
- Support for other functions can be added (afternoon session)

Brian's unit system

- Brian allows to use units for scalars and vectors

```
>>> E_L = -70*mV
>>> print E_L
-70.0 mV
>>> freqs = [100, 200, 300] * Hz
>>> print freqs
[ 100.  200.  300.] Hz
```

- Most numpy functions work correctly with units (make sure to not import from numpy directly)

```
>>> mean(freqs)
200.0 * hertz
>>> diff(freqs)
array([ 100.,  100.]) * hertz
```

Brian's unit system

- To remove units, use `numpy.asarray` or divide by the unit

```
>>> print freqs/Hz
[ 100.  200.  300.]
>>> print asarray(freqs)
[ 100.  200.  300.]
```

- For state variables: adding an underscore returns unitless value

```
>>> print G.v
<neurongroup.v: array([-70., -70., -70., -70., -70.]) * mvolt>
>>> print G.v_
<neurongroup_1.v: array([-0.07, -0.07, -0.07, -0.07, -0.07])>
```

- Unit consistency is also checked in equations, expressions and abstract code statements

Running simulations: “magic”

- “Magic” network – Brian collects all the object it “sees”:

```
G = NeuronGroup(...)
S = Synapses(...)
mon = SpikeMonitor(...)

run(runtime)  # G, S and mon
```

Running simulations: “magic”

- “Magic” network – Brian collects all the object it “sees”:

```
G = NeuronGroup(...)
S = Synapses(...)
monitors = [SpikeMonitor(...), StateMonitor(...)]

run(runtime)  # only G, S are "visible"!
```

Running simulations: “magic“

- “Magic“ network – Brian collects all the object it “sees“:

```
G = NeuronGroup(...)
S = Synapses(...)
monitors = [SpikeMonitor(...), StateMonitor(...)]

print collect()  # added recently, not in 2.0a8
```

Output:

```
set([NeuronGroup(..., name='neurongroup'),
    Synapses(..., name='synapses')])
```

Running simulations: explicit

- Explicitly constructed network, recommended for complicated setups:

```
G = NeuronGroup(...)
S = Synapses(...)
monitors = [SpikeMonitor(...), StateMonitor(...)]

net = Network(G, S, monitors)
net.run(runtime)
```

Running simulations: explicit

- Explicitly constructed network, recommended for complicated setups:

```
G = NeuronGroup(...)
S = Synapses(...)
monitors = [SpikeMonitor(...), StateMonitor(...)]

net = Network(collect())
net.add(monitors)
net.run(runtime)
```

Extending Brian's scope

- **Modelfitting toolbox:**
  fitting neural models to spike trains (not yet for Brian2)
  Rossant, Cyrille, et al., Frontiers in Neuroscience (2011)
- **Brian hears:**
  auditory periphery models (porting to Brian2 started, bridge to Brian1)
  Fontaine, Bertrand, et al., Frontiers in Neuroinformatics (2011)
- **Multi-compartmental modelling:**
  simulate multi-compartment models in Brian (experimental in Brian1, porting to Brian2 started)

Coffee break!

Topographical connections in Brian

```
rows, cols = 50, 50
G = NeuronGroup(rows * cols, '''x : meter
                                y : meter''')
# initialize the grid positions
grid_dist = 25*umeter
G.x = '(i / rows) * grid_dist - cols/2.0 * grid_dist'
G.y = '(i % rows) * grid_dist - rows/2.0 * grid_dist'

# Random connections (no self-connections)
S_stochastic = Synapses(G, G)
S_stochastic.connect('i != j',
                     p=('1.5 * exp(-((x_pre-x_post)**2 + '
                                    '(y_pre-y_post)**2)/(2*(60*umeter)**2))'))
```

Topographical connections in Brian