CNS Tutorial T3

Modelling of spiking neural networks with the Brian simulator

Marcel Stimberg

Dan Goodman

Schedule for today

Morning

Time	*Topic*
9.00 – 9.30	Introduction & installation
9.30 – 10.10	Core concepts of Brian2
10.10 – 10.30	COFFEE BREAK
10.40 – 12.00	Hands-on tutorial

Afternoon

13.30 – 13.45	Going from Brian 1 to Brian 2
13.45 – 15.10	Advanced Brian 2 + Extending Brian 2
15.10 – 15.40	COFFEE BREAK
15.40 – 16.30	Question & answers

The spirit of BRIAN

"A simulator should not only save the time of processors, but also the time of scientists"

scientist

computer

Writing code often takes more time than running it

Goals:

- Quick model coding
- Flexible

models are defined by equations (rather than pre-defined)

Goodman, D. and R. Brette (2009). The Brian simulator. Front Neurosci, doi:10.3389/neuro.01.026.2009.

An example

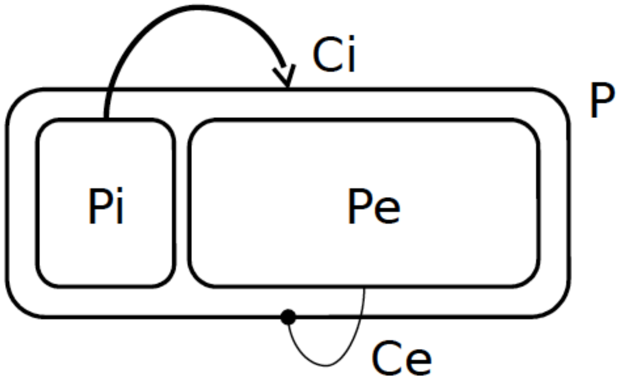
```
from brian2 import *

tau_m = 20*ms
tau_e = 5*ms
tau_i = 10*ms
V_th = -50*mV
E_L = -60*mV
I_const = 11*mV
eqs = '''
dv/dt = (-(v-E_L) + I_const + g_e + g_i)/tau_m : volt
dg_e/dt = -g_e/tau_e : volt
dg_i/dt = -g_i/tau_i : volt
'''
P = NeuronGroup(4000, model=eqs,
                threshold='v>V_th', reset='v=E_L')
P.v = 'E_L+10*mV*rand()'
Pe = P[:3200]
Pi = P[3200:]

w_e = 1.62*mV
w_i = 9*mV
Ce = Synapses(Pe, P, pre='g_e+=w_e')
Ci = Synapses(Pi, P, pre='g_i-=w_i')
Ce.connect(True, p=0.02)
Ci.connect(True, p=0.02)

M = SpikeMonitor(P)

run(1*second)
plot(M.t/ms, M.i, '.')
xlabel('Time (in ms)'); ylabel('Neuron number')
show()
```

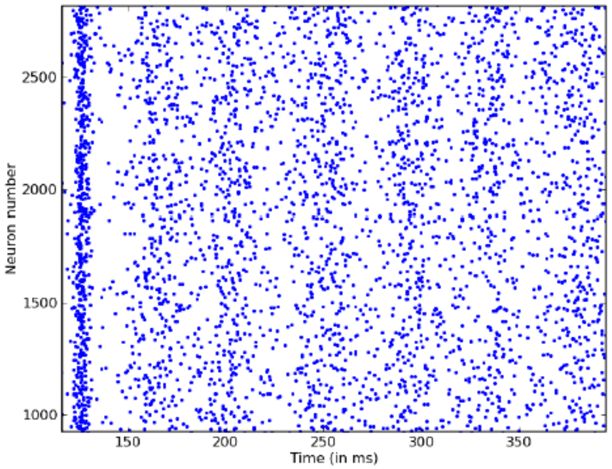
Stimberg M, Goodman DFM, Benichoux V, Brette R (2014).**Equation-oriented specification of neural models for simulations**. Frontiers Neuroinf, doi: 10.3389/fninf.2014.00006.

$$\tau_m \frac{\mathrm{d}V}{\mathrm{d}t} = -(V - E_L) + g_e + g_i$$

$$\tau_e \frac{\mathrm{d}g_e}{\mathrm{d}t} = -g_e$$

$$\tau_i \frac{\mathrm{d}g_i}{\mathrm{d}t} = -g_i$$





Standardization issues

Key issue: each simulator has its own language, how to communicate models?

Example 1: PyNEST (Python interface to NEST)

```
SetDefaults("iaf_psc_delta",
            {"C_m" : 20,
             "tau_m": 20,
             "t_ref": 2,
             "E_L" : 0,
             "V_th" : 20})
nodes_ex = Create("iaf_psc_delta", NE)
```

Component-based approach:
you need to know what the components are exactly
you need to know parameter names
you need to know implicit units

Standardization issues

Solution: *equation-oriented approach*

Example 2: NineML (Izhikevich model)

Issues:

- can’t specify a full simulation (including initialization & stimulus)
- heavy!

```
<?xml version='1.0' encoding='UTF-8'?>
<NineML xmlns="http://nineml.org/9ML/0.1">
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://nineml.org/9ML/0.1 NineML\_v0.2.xsd">

  <ComponentClass name="izhikevichCellNew">

    <Parameter name="a" dimension="none"/>
    <Parameter name="c" dimension="voltage"/>
    <Parameter name="b" dimension="per_time"/>
    <Parameter name="d" dimension="voltage_per_time"/>
    <Parameter name="theta" dimension="voltage"/>

    <AnalogPort name="iSyn" mode="reduce" reduce_op="+" dimension="current"/>
    <AnalogPort name="U" mode="send" dimension="none"/>
    <AnalogPort name="V" mode="send" dimension="voltage"/>
    <EventPort name="spikeOutput" mode="send"/>


    <Dynamics>

        <StateVariable name="V" dimension="voltage"/>
        <StateVariable name="U" dimension="voltage_per_time"/>

        <Alias name="rv" dimension="none">
            <MathInline>V*U</MathInline>
        </Alias>

        <Regime name="subthresholdRegime">

          <TimeDerivative variable="U">
            <MathInline>a*(b*V - U)</MathInline>
          </TimeDerivative>

          <TimeDerivative variable="V">
            <MathInline>0.04*V*V + 5*V + 140.0 - U + iSyn</MathInline>
          </TimeDerivative>


          <OnCondition>
            <Trigger>
              <MathInline>V \> theta </MathInline>
            </Trigger>

            <StateAssignment variable="V" >
              <MathInline>c</MathInline>
            </StateAssignment>

            <StateAssignment variable="U" >
              <MathInline>U+d</MathInline>
            </StateAssignment>

            <EventOut port="spikeOutput" />

          </OnCondition>

        </Regime>
    </Dynamics>

  </ComponentClass>
</NineML>
```

Standardization issues: the BRIAN way

Goal: to minimize «language entropy» = uncertainty about syntax and names, given the meaning

Brette R (2012). On the design of script languages for neural simulation. Network 23(4), 150-156

<u>Key points:</u>

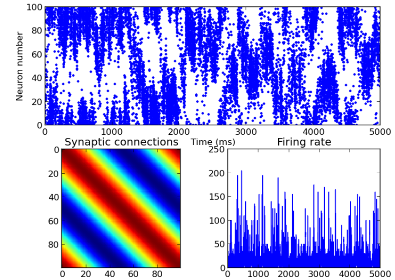
1) There is already an accepted standard for models: math!
2) If you specify names, units and equations yourself, you don't need to know them in advance
3) Full expressivity requires a programming language

The future of BRIAN

BRIAN2 relies on *code generation* to run on multiple targets

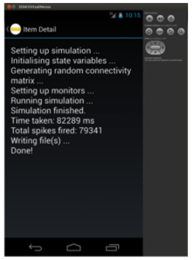
```
# Neurons
input = PoissonGroup(N, rates=F)
neurons = NeuronGroup(1, '''dv/dt = (g_e*(E_e-v_r)+E_l-v)/tau_m : volt
                            dg_e/dt = -g_e/tau_e : 1''',
                      threshold='v>vt', reset='v=v_r')
# Synaptic connections
S = Synapses(input, neurons,
             '''w:1
                dApre/dt = -Apre/taupre : 1 (event-driven)
                dApost/dt = -Apost/taupost : 1 (event-driven)''',
             pre='''ge += w
                    Apre += dApre
                    w = clip(w+Apost, 0, gmax)''',
             post='''Apost += dApost
                     w = clip(w+Apre, 0, gmax)''',
             connect=True)
S.w = 'rand()*gmax'
```

Simulation on PC, clusters, GPU



Interface with robots

Embedded simulation on Android smartphones



Website

briansimulator.org

The Brian spiking neural network simulator

Posts Comments

- About
- Download
- Demo
- Learn
- Features
- Manual
- Support
- Showcase
- Publications
- Contribute
- Join the project
- Team

News

Brian tutorial at CNS 2015
17 Jul 2015

Google Summer of Code 2015
5 Mar 2015

Brian tutorial at CNS 2014
8 Jul 2014

New Brian paper: Equation-oriented specification of neural models for simulations

About

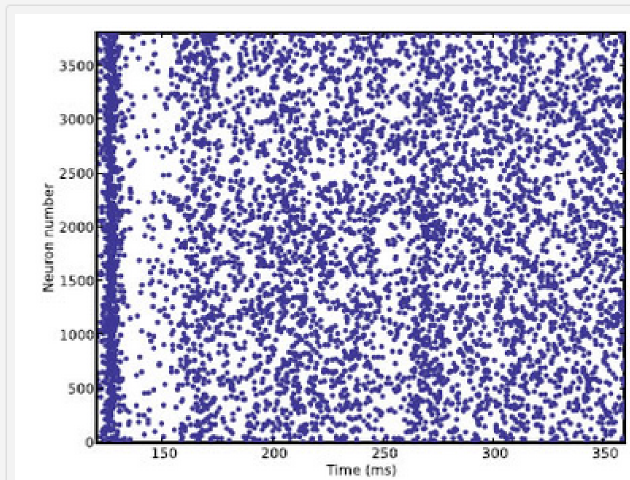
*** Brian 2.0 fourth beta release:** release notes *****
*** Brian tutorial at the CNS 2015 meeting in Prague:** more information *****

Brian is a simulator for spiking neural networks available on almost all platforms. The motivation for this project is that a simulator should not only save the time of processors, but also the time of scientists.

Brian is easy to learn and use, highly flexible and easily extensible. The Brian package itself and simulations using it are all written in the Python programming language, which is an easy, concise and highly developed language with many advanced features and development tools, excellent documentation and a large community of users providing support and extension packages.

The following code defines a randomly connected network of integrate and fire neurons with exponential inhibitory and excitatory currents, runs the simulation and makes the raster plot on the right.

```
from brian2 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, eqs, threshold='v>-50*mV', reset='v=-60*mV')
P.v = -60*mV
Pe = P[:3200]
Pi = P[3200:]
Ce = Synapses(Pe, P, pre='ge+=1.62*mV')
Ce.connect(True, p=0.02)
Ci = Synapses(Pi, P, pre='gi-=9*mV')
Ci.connect(True, p=0.02)
M = SpikeMonitor(P)
run(1*second)
plot(M.t/ms, M.i, '.')
show()
```



See the manuals for more examples.

Documentation

brian2.readthedocs.org

Brian 2

Search docs

- Introduction
- Tutorials
- User's guide
- Advanced guide
- Examples
- Reference documentation
- Developer's guide

Read the Docs v: 2.0b4

Docs » Brian 2 documentation

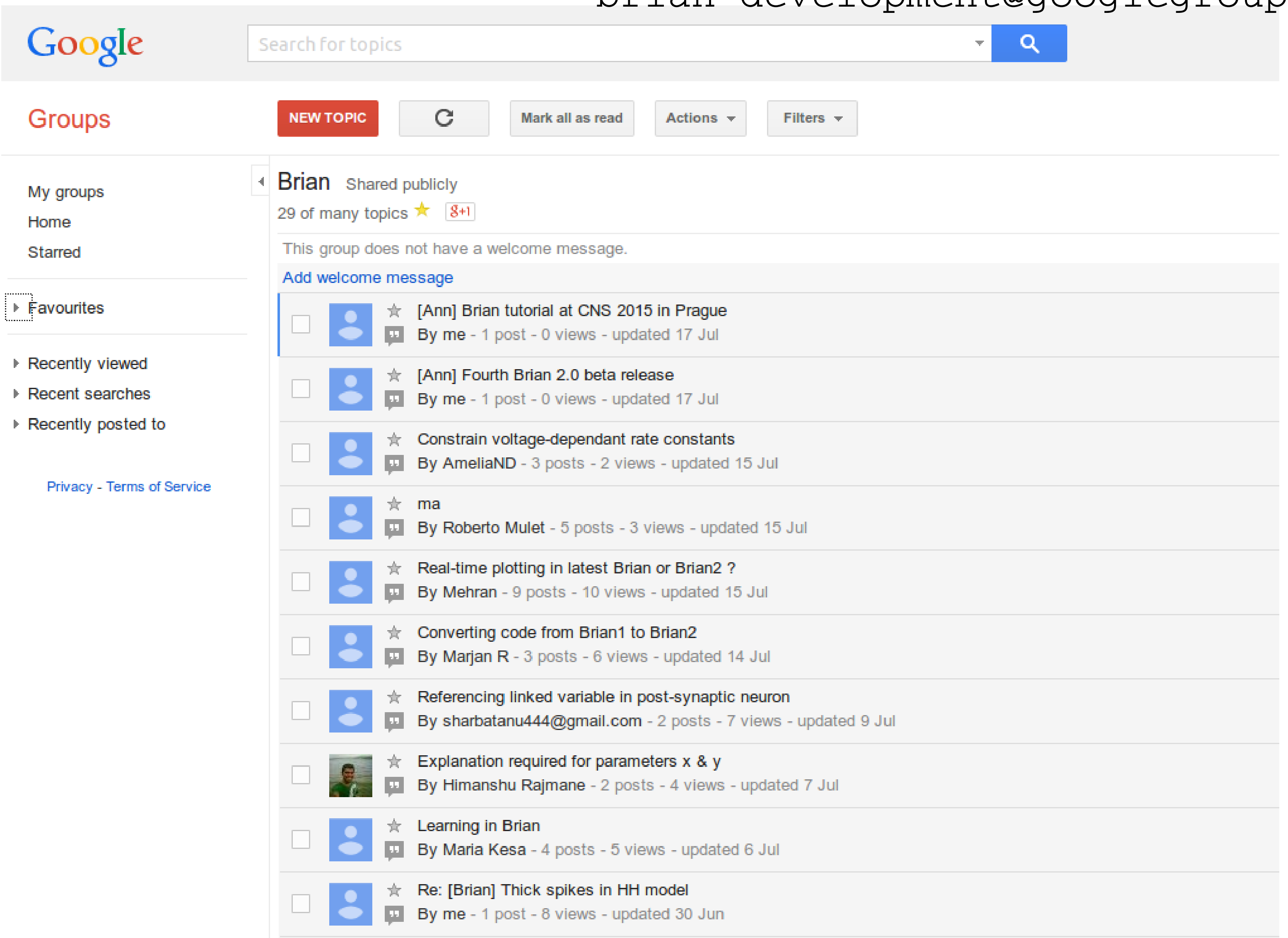
Brian 2 documentation

Contents:

Mailing lists

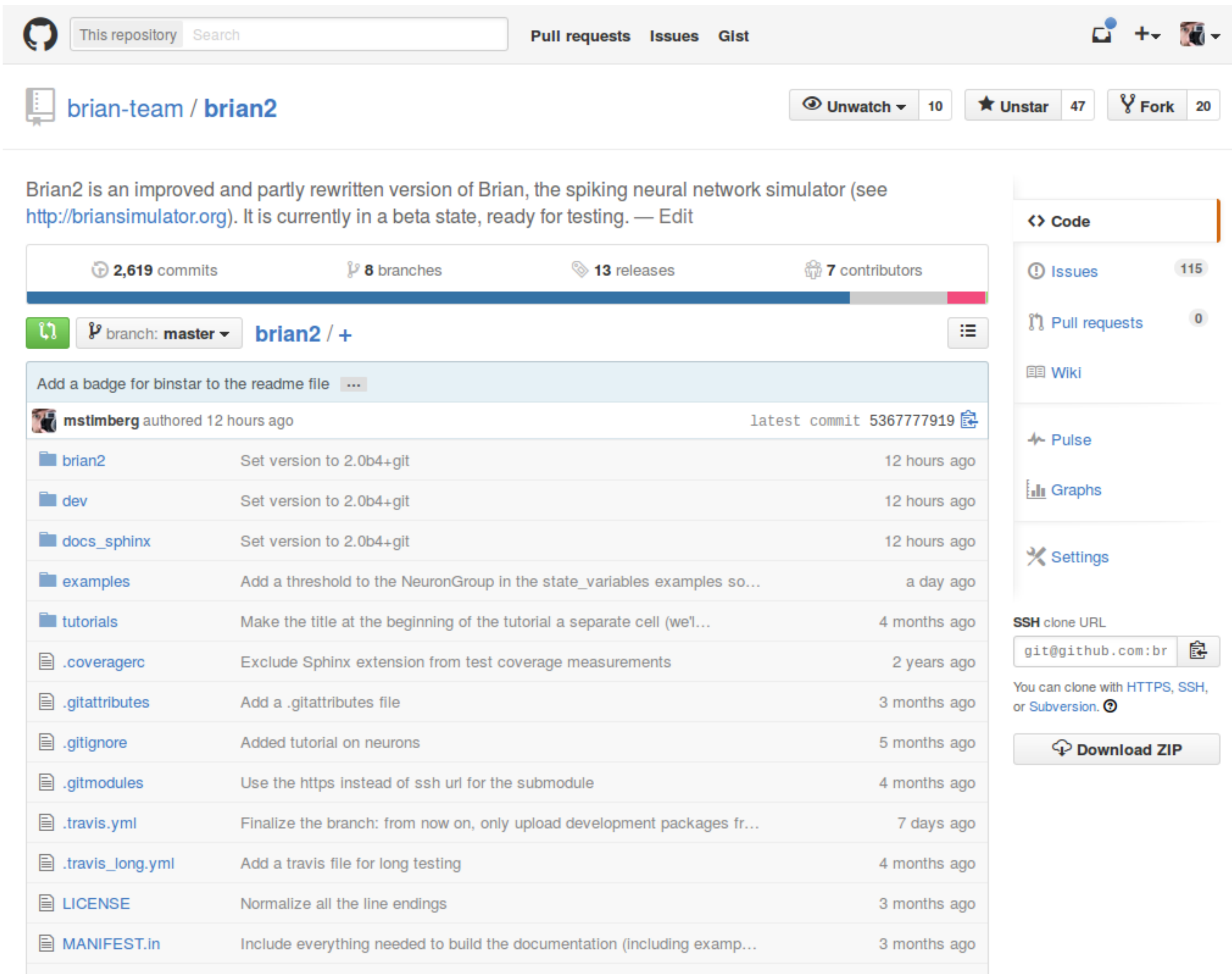
briansupport@googlegroups.com

brian-development@googlegroups.com



Code repository

https://github.com/brian-team/brian2



Installation

Recommended way:

- Use Anaconda distribution
- Add the "brian-team" channel

```
conda config --add channels brian-team
```

- Install brian2

```
conda install brian2
```

More infos and alternative installation:
brian2.readthedocs.org

Schedule for today

Morning

Time	*Topic*
9.00 – 9.30	Introduction & installation
9.30 – 10.30	Core concepts of Brian2
10.30 – 10.50	COFFEE BREAK
10.50 – 12.00	Hands-on tutorial

Afternoon

13.30 – 13.45	Going from Brian 1 to Brian 2
13.45 – 15.10	Advanced Brian 2 + Extending Brian 2
15.10 – 15.40	COFFEE BREAK
15.40 – 16.30	Question & answers

Core concepts of BRIAN2

Anatomy of a Brian script

```
from brian2 import *

tau_m = 20*ms
tau_e = 5*ms
tau_i = 10*ms
V_th = -50*mV
E_L = -60*mV
I_const = 11*mV
eqs = '''
dv/dt = (-(v-E_L) + I_const + g_e + g_i)/tau_m : volt
dg_e/dt = -g_e/tau_e : volt
dg_i/dt = -g_i/tau_i : volt
'''
P = NeuronGroup(4000, model=eqs,
                threshold='v>V_th', reset='v=E_L')
P.v = 'E_L+10*mV*rand()'
Pe = P[:3200]
Pi = P[3200:]

w_e = 1.62*mV
w_i = 9*mV
Ce = Synapses(Pe, P, pre='g_e+=w_e')
Ci = Synapses(Pi, P, pre='g_i-=w_i')
Ce.connect(True, p=0.02)
Ci.connect(True, p=0.02)

M = SpikeMonitor(P)

run(1*second)
plot(M.t/ms, M.i, '.')
xlabel('Time (in ms)'); ylabel('Neuron number')
show()
```

Import Brian library

Set constants

Specify the neuronal model

Initialise state variables

Specify subgroups of neurons

Specify synaptic model

Connect neurons

Record activity

Run the simulation

Plot the activity

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')
G.run_regularly('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, sign`
  - 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: explicit

- Explicitly constructed network, recommended for complicated setups:

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

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

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