Code generation in Brian 2

From mathematical model descriptions to executable code

Marcel Stimberg, ENS Paris
marcel.stimberg@ens.fr

Dan Goodman, Harvard Medical School
Romain Brette, ENS Paris

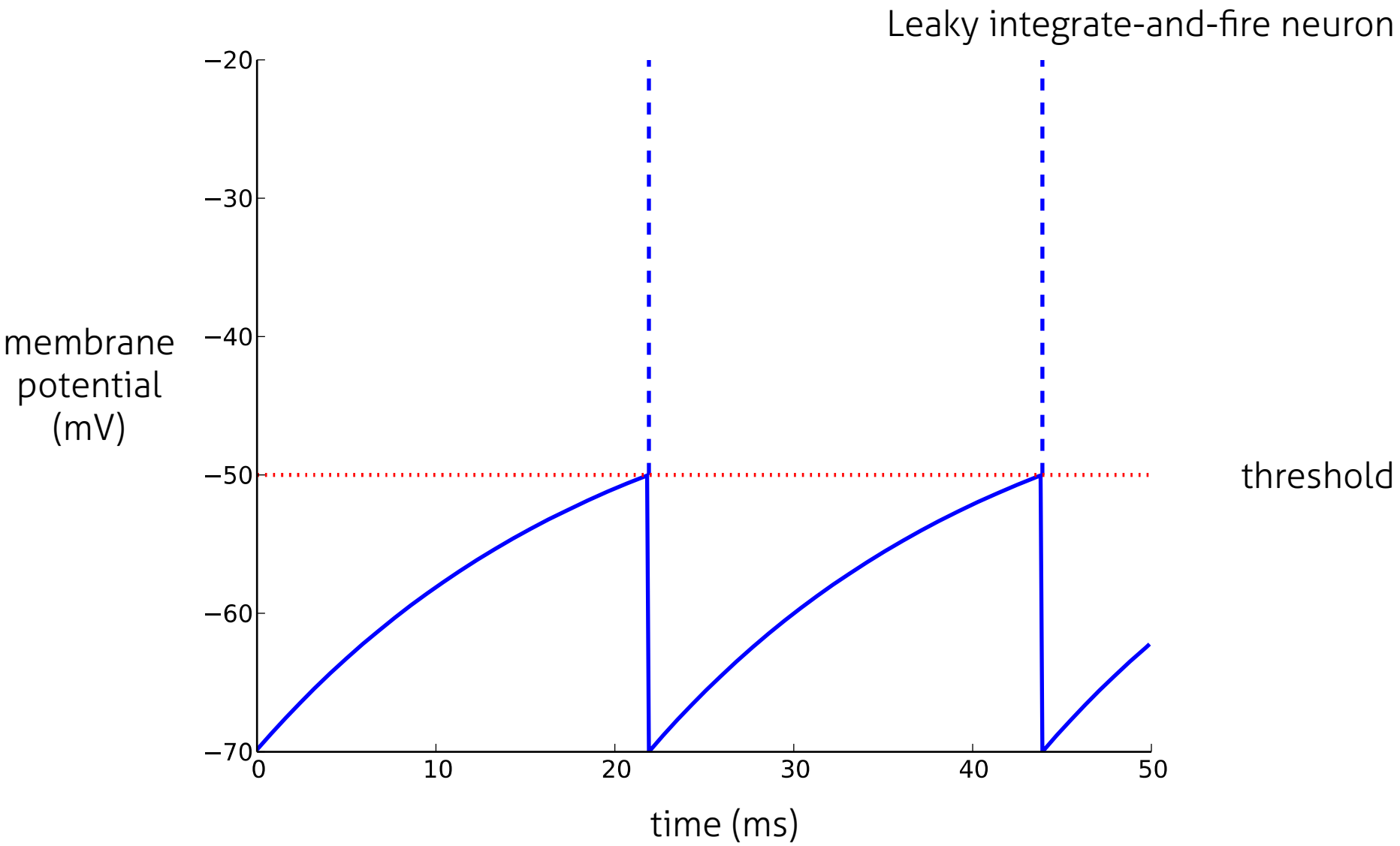
Who is Brian?

- A neural simulator, originally started by Romain Brette and Dan Goodman in 2007 at ENS Paris
- Focuses on ease-of-use and flexibility, while being "reasonably fast"
- Written in Python, free-and-open-source
- Brian2: Rewrite of core parts of Brian, more consistent use of string-based model descriptions and code generation

Goodman DF and Brette R (2009). The Brian simulator. *Frontiers in Neuroscience*

String-based model descriptions

Modelling neural activity



Modelling neural activity

Mathematical description:

$$E_L = -70\,\mathrm{mV}$$

$$v_{\mathrm{th}} = -50\,\mathrm{mV}$$

$$v_{\mathrm{r}} = E_L$$

$$\tau = 20\,\mathrm{ms}$$

$$\frac{\mathrm{d}v}{\mathrm{d}t} = -\frac{(v - E_L - I)}{\tau}$$

(v and I have units of volt)

When $v > v_{\mathrm{th}}$:

$v \leftarrow v_r$

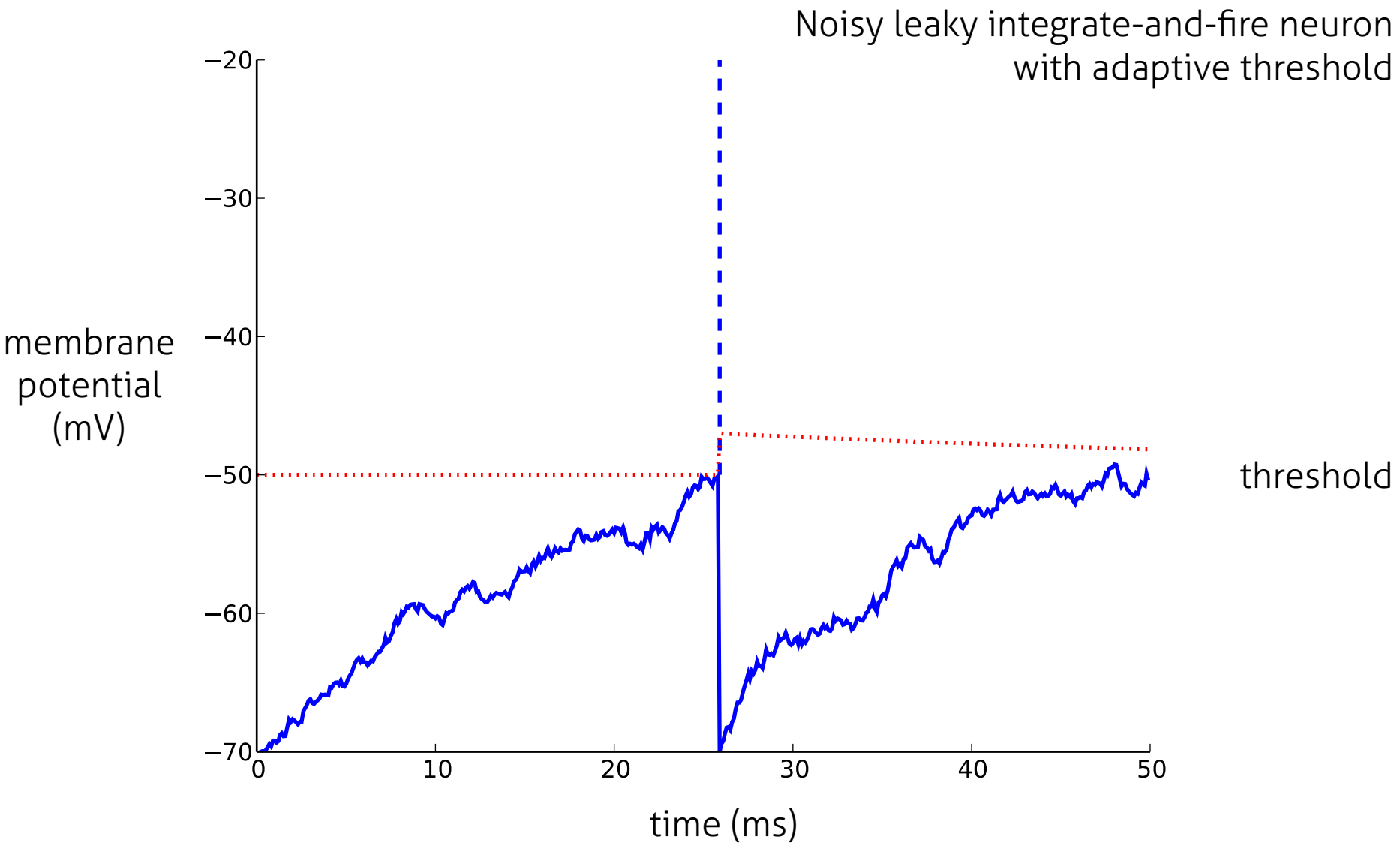
Modelling neural activity

Brian description:

```
E_L = -70*mV
v_r = E_L
v_th = -50*mV
tau = 20*ms

G = NeuronGroup(N,
                '''
                dv/dt = -(v - E_L - I) / tau : volt
                I : volt
                ''',
                threshold='v>v_th',
                reset='v=v_r')
```

Modelling neural activity



Modelling neural activity

Brian description:

```
E_L = -70*mV
v_r = E_L
v_th0 = -50*mV
tau = 20*ms
tau_th = 50*ms
sigma = 4*mV

G = NeuronGroup(N,
                '''
                dv/dt = -(v - E_L - I) / tau + sigma*tau**-.5*xi: volt
                dv_th/dt = -(v_th - v_th0) / tau_th : volt
                I : volt
                ''',
                threshold='v>v_th',
                reset='''v=v_r
                         v_th+=3*mV''')
```

Modelling synaptic learning

Brian description:

```
S = Synapses(inp, G,
             '''
             w : 1
             dA_source/dt = -A_source/tau_source : 1 (event-driven)
             dA_target/dt = -A_target/tau_target : 1 (event-driven)
             ''',
             pre='''ge += w
                    A_source += delta_source
                    w = clip(w+A_target, 0, w_max)''',
             post='''A_target += delta_target
                     w = clip(w+A_source, 0, w_max)''')
```

Connecting neurons

String expressions referring to indices `i` and `j`:

- one-to-one connectivity: `S.connect('i == j')`
- local neighbourhood:
  `S.connect('abs(i - j) < 5')`
- Random connectivity, avoiding self-connections:

```
S.connect('i != j',
          p='1./(sigma*sqrt(2*pi))*  ↪
             exp(-(i - j)**2/(2*sigma**2)')
```

- Connecting cells depending on properties (stored as state variables):
  `S.connect('sqrt((x_pre-x_post)**2) < 100*umetre')`

Code generation

Code generation

- **Event-based updates** (impact of an incoming spike on a neuron, reset after a spike, …) are formulated as "abstract code"
- **Continuous updates** (neuronal dynamics, synaptic dynamics, …) are formulated as differential equations → need to be numerically integrated

  → yields abstract code
- Syntax and consistency checks (e.g. units) happen on abstract code

Abstract code

Model equations

```
dv/dt =  (-v + w) / tau_v : volt
dw/dt = -w / tau_w : volt
```

+

Numerical integration

```
x_new = x + dt*f(x, t)
```

=

Abstract code

```
_v = dt*(-v + w)/tau_v + v
_w = -dt*w/tau_w + w
v = _v
w = _w
```

Uses **sympy** for symbolic mathematics

Interface to code generation

Abstract code

```
_v = dt*(-v + w)/tau_v + v
_w = -dt*w/tau_w + w
v = _v
w = _w
```

Abstract namespace

```
{'v': ArrayVariable('v',
                    unit=volt,
                    dtype=float64,
                    ...),
 'tau_v': Constant('tau_v',
                   unit=second,
                   value=0.010),
 ...}
```

Indices

```
{'v': 'n_idx', 'w': 'n_idx'}
```


Code generation

Executable code generation (C++)

general template

```
for(int _n_idx=0; _n_idx<_num_neurons; _n_idx++)
{
    double v = _ptr_array_neurongroup_v[_n_idx];
    double w = _ptr_array_neurongroup_w[_n_idx];
    const double _w = -(dt) * w / tau_w + w;
    const double _v = dt * (-(v) + w) / tau_v + v;
    w = _w;
    v = _v;
    _ptr_array_neurongroup_v[_n_idx] = v;
    _ptr_array_neurongroup_w[_n_idx] = w;
}
```

model-specific code

Executable “Code object”

Abstract code

Abstract namespace

Indices

C++ code

```
for(int _n_idx=0; _n_idx<_num_neurons; _n_idx++)
{
    double v = _ptr_array_neurongroup_v[_n_idx];
    double w = _ptr_array_neurongroup_w[_n_idx];
    const double _w = -(dt) * w / tau_w + w;
    const double _v = dt * (-(v) + w) / tau_v + v
    w = _w;
    v = _v;
    _ptr_array_neurongroup_v[_n_idx] = v;
    _ptr_array_neurongroup_w[_n_idx] = w;
}
```

Namespace

```
{'v': array([-70.02276181, -70.50338621,
             -70.86131424, ...]),
 'tau_v': 0.010, 'tau_w': 0.050,
 'dt': 0.0001, ...}
```

Brian's “runtime” mode

```
# Network.run:

# Pseudocode, the actual run loop also takes care of
# multiple clocks, etc.

while clock.t < run_time:
    # self.objects contains objects such as
    # NeuronGroup, Synapses, StateMonitor, ...
    for obj in self.objects:
        # The "code objects" are the objects doing
        # the actual computations, e.g. the numerical integration,
        # thresholding, resetting, synaptic propagation, etc.
        for code_object in object.code_objects:
            code_object.run()

    clock.t += clock.dt
```

Brian's “standalone” mode

We are already generating code for the core computational parts – why not go all the way?

- Add memory allocation in target language
- Add simulation loop in target language

  → full simulation runs independent of Brian
- Also a possible starting point for simulations that connect to special hardware, etc.

Standalone mode: usage

```
from brian2 import *



E_L = -70*mV
v_r = E_L
v_th0 = -50*mV
tau = 20*ms
tau_th = 50*ms
sigma = 4*mV
N = 100

G = NeuronGroup(N,
                '''
                dv/dt = -(v - E_L - I) / tau + sigma*tau**-.5*xi: volt
                dv_th/dt = -(v_th - v_th0) / tau_th : volt
                I : volt
                ''',
                threshold='v>v_th',
                reset='''v=v_r
                         v_th+=3*mV''')
G.I = np.linspace(0, 50*mV, N)
mon = SpikeMonitor(G)
run(1000*ms)
```

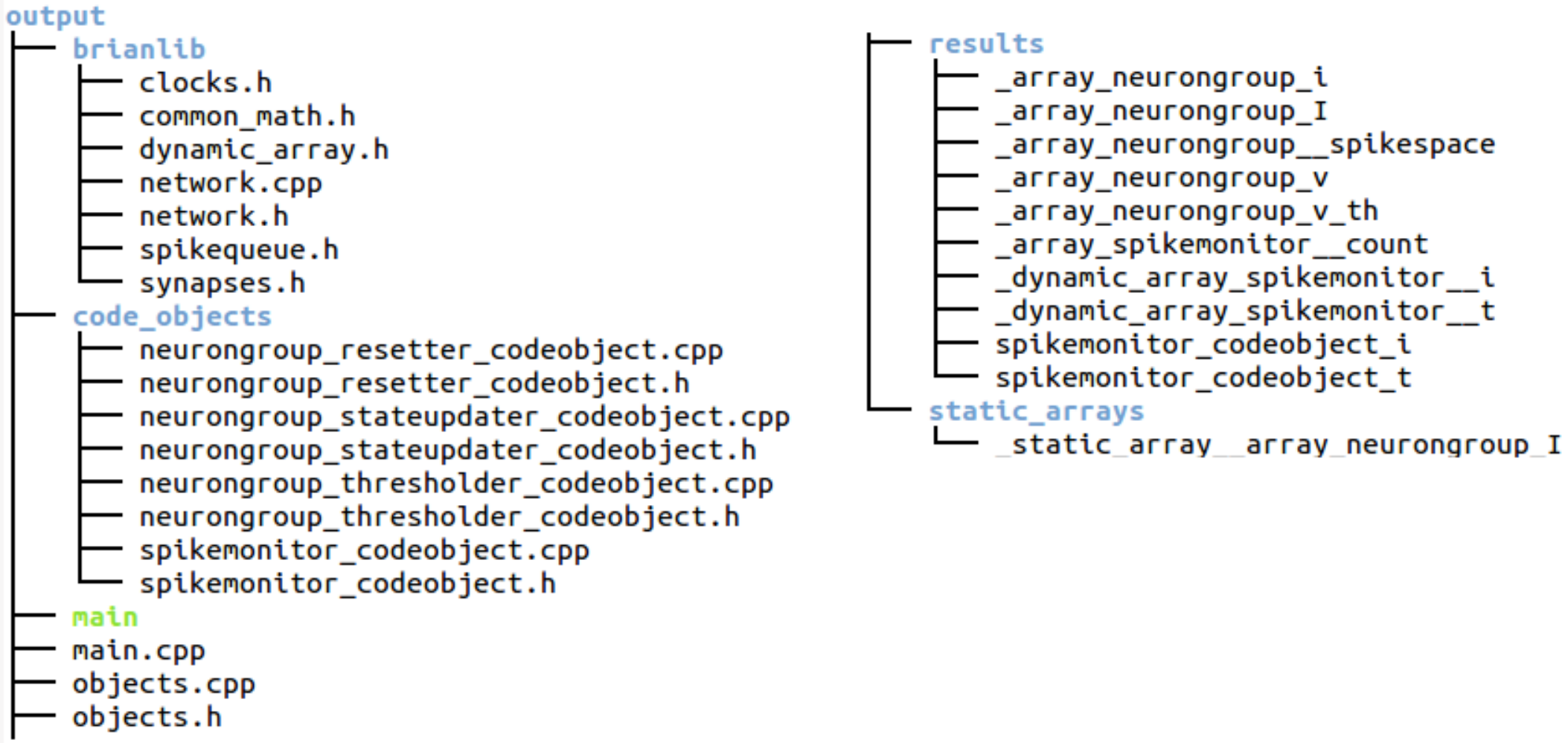
Standalone mode: usage

```
from brian2 import *
from brian2.devices.cpp_standalone import *
set_device('cpp_standalone')

E_L = -70*mV
v_r = E_L
v_th0 = -50*mV
tau = 20*ms
tau_th = 50*ms
sigma = 4*mV
N = 100

G = NeuronGroup(N,
                '''
                dv/dt = -(v - E_L - I) / tau + sigma*tau**-.5*xi: volt
                dv_th/dt = -(v_th - v_th0) / tau_th : volt
                I : volt
                ''',
                threshold='v>v_th',
                reset='''v=v_r
                         v_th+=3*mV''')
G.I = np.linspace(0, 50*mV, N)
mon = SpikeMonitor(G)
run(1000*ms)
build(compile_project=True, run_project=True)
```

Standalone mode: generated files



Summary: code generation

- Combine easy-to-use Python interface with efficient code
- Generating code doesn't mean to write a Python-to-X compiler:
  Model-specific code only uses a small subset of Python (assignments, arithmetic expressions)
- Allows to run scripts on devices where it couldn't run otherwise

Thanks for listening

Try it out (alpha version):
https://pypi.python.org/pypi/Brian2

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

Read the docs:

http://brian2.readthedocs.org

Discuss/ask/comment:

brian-development@googlegroups.com

General information about Brian

http://briansimulator.org

Stimberg, Goodman, Benichoux, and Brette.
'Equation-Oriented Specification of Neural Models for Simulations'.
Frontiers in Neuroinformatics 8: (2014): 6