Brian 2: an intuitive and efficient neural simulator

Marcel Stimberg[1*], Romain Brette[1], and Dan F. M. Goodman[2]

[1]Sorbonne Université, INSERM, CNRS, Institut de la Vision, Paris, France
[2]Department of Electrical and Electronic Engineering, Imperial College London, UK

Abstract

To be maximally useful for neuroscience research, neural simulators must make it possible to define original models. This is especially important because a computational experiment might not only need descriptions of neurons and synapses, but also models of interactions with the environment (e.g. muscles), or the environment itself. To preserve high performance when defining new models, current simulators offer two options: low-level programming, or mark-up languages (and other domain specific languages). The first option requires time and expertise, is prone to errors, and contributes to problems with reproducibility and replicability. The second option has limited scope, since it can only describe the range of neural models covered by the ontology. Other aspects of a computational experiment, such as the stimulation protocol, cannot be expressed within this framework. "Brian" 2 is a complete rewrite of Brian that addresses this issue by using runtime code generation with a procedural equation-oriented approach. Brian 2 enables scientists to write code that is particularly simple and concise, closely matching the way they conceptualise their models, while the technique of runtime code generation automatically transforms high level descriptions of models into efficient low level code tailored to different hardware (e.g. CPU or GPU). We illustrate it with several challenging examples: a plastic model of the pyloric network of crustaceans, a closed-loop sensorimotor model, programmatic exploration of a neuron model, and an auditory model with real-time input from a microphone.

1 Introduction

Neural simulators are increasingly used to develop models of the nervous system, at different scales and in a variety of contexts (Brette et al., 2007). Popular tools for simulating spiking neurons and networks of such neurons are NEURON (Carnevale & Hines, 2006), GENESIS (Bower & Beeman, 1998), NEST (Gewaltig & Diesmann, 2007), and Brian (Goodman & Brette, 2009). Most of these simulators come with a library of standard models that they allow the user to choose from. However, we argue that to be maximally useful for research, a simulator should also be designed to facilitate work that goes beyond what is known at the time that the tool is created, and therefore enable the user to investigate new mechanisms. Simulators take widely different approaches to this issue. For some simulators, adding new mechanisms requires specifying them in a low-level programming language such as C++, and integrating them with the simulator code (e.g. NEST). Amongst these, some provide domain-specific languages, e.g. NMODL (Hines & Carnevale, 2000, for NEURON) or NESTML (Plotnikov et al., 2016, for NEST), and tools to transform these descriptions into compiled modules that can then be used in simulation scripts. Finally, the Brian simulator has been built around mathematical model descriptions that are part of the simulation script itself.

Another approach to model definitions has been established by the development of simulator-independent markup languages, for example NeuroML/LEMS (Gleeson et al., 2010; Cannon et al., 2014) and NineML (Raikov et al., 2011). However, markup languages address only part of the problem. A computational experiment is not fully specified by a neural model: it also includes a particular protocol, for example a sequence of visual stimuli. Capturing the full range of potential protocols cannot be done with a purely declarative markup language, but is straightforward in a general purpose programming language. For this reason, the Brian simulator combines the model descriptions with a procedural, *computational experiment* approach: a simulation is a

user script written in Python, with models described in their mathematical form, without any reference to predefined models. This script may implement arbitrary protocols by loading data, defining models, running simulations and analysing results. Due to Python's expressiveness, there is no limit on the structure of the computational experiment: stimuli can be changed in a loop, or presented conditionally based on the results of the simulation, etc. This flexibility can only be obtained with a general-purpose programming language, and is necessary to specify the full range of computational experiments that scientists are interested in.

While the procedural, equation-oriented approach addresses the issue of flexibility for both the modelling and the computational experiment, it comes at the cost of reduced performance, especially for small-scale models that do not benefit much from vectorization techniques (Brette & Goodman, 2011). The reduced performance results from the use of an interpreted language to implement arbitrary models, instead of the use of pre-compiled code for a set of previously defined models. Thus, simulators generally have to find a trade-off between flexibility and performance, and previous approaches have often chosen one over the other. In practice, this makes computational experiments that are based on non-standard models either difficult to implement or slow to perform. We will describe four case studies in this article: exploring unconventional plasticity rules for a small neural circuit (***Figure 1***, ***Figure 2***); running a model of a sensorimotor loop (***Figure 3***); determining the spiking threshold of a complex model by bisection (***Figure 4***, ***Figure 5***); and running an auditory model with real-time input from a microphone (***Figure 6***, ***Figure 7***).

Brian 2, a complete rewrite of the Brian simulator, solves the apparent dichotomy between flexibility and performance using the technique of code generation, which transparently transforms high-level user-defined models into efficient compiled code (Goodman, 2010; Stimberg et al., 2014; Blundell et al., 2018). This generated code is inserted within the flow of the simulation script, which makes it compatible with the procedural approach. Code generation is used not only to run the models but also to build them, and therefore also accelerates stages such as synapse creation. The code generation framework has been designed to be extensible on several levels. On a general level, code generation targets can be added to generate code for other architectures, e.g. graphical processing units, from the same simulation description. On a more specific level, new functionality can be added by providing a small amount of code written in the target language, e.g. to connect the simulation to an input device. Implementing this solution in a way that is transparent to the user requires solving important design and computational problems, which we will describe in the following.

2 Design and Implementation

We will explain the key design decisions by starting from the requirements that motivated them. Note that from now on we will use the term "Brian" as referring to its latest version, i.e. Brian 2, and only use "Brian 1" and "Brian 2" when discussing differences between them.

Our first requirement is that users can easily define non-standard models, which may include models of neurons and synapses but also of other aspects such as muscles and environment. This is made possible by an equation-oriented approach, i.e., models are described by mathematical equations. We first focus on the design at the *mathematical level*, and we illustrate with two unconventional models: a model of intrinsic plasticity in the pyloric network of the crustacean stomatogastric ganglion (***Figure 1***, ***Figure 2***), and a closed-loop sensorimotor model of ocular movements (***Figure 3***).

Our second requirement is that users should be able to easily implement a complete computational experiment in Brian. Models must interact with a general control flow, which may include stimulus generation and various operations. This is made possible by taking a procedural approach to defining a complete computational experiment, rather than a declarative model definition, allowing users to make full use of the generality of the Python language. In the section on the *computational experiment level*, we demonstrate the interaction between a general control flow expressed in Python and the simulation run in a case study that uses a bisection algorithm to determine a neuron's firing threshold as a function of sodium channel density (***Figure 4***, ***Figure 5***).

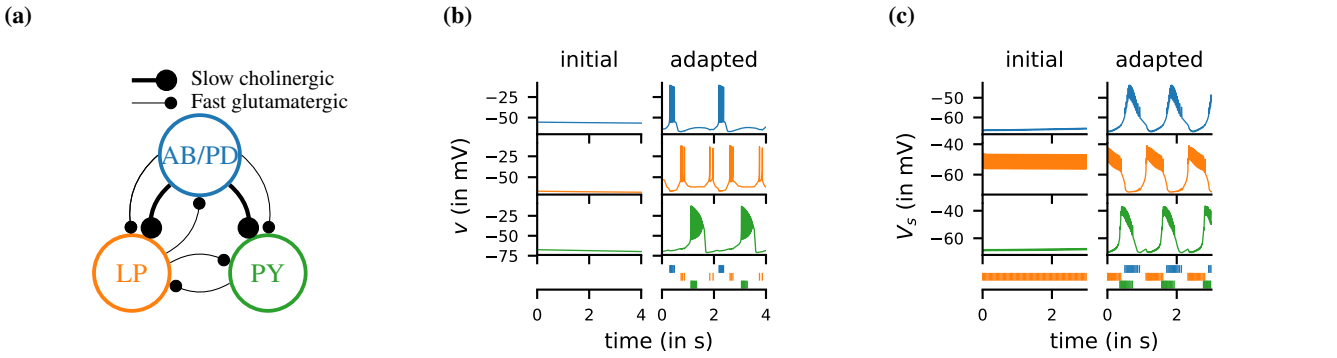


Figure 1. Case study: a model of the pyloric network of the crustacean stomatogastric ganglion, inspired by several modeling papers on this subject (Golowasch et al., 1999; Prinz et al., 2004; Prinz, 2006; O'Leary et al., 2014) **(a)** Schematic of the modeled circuit (after Prinz et al., 2004). The pacemaker kernel is modeled by a single neuron representing both anterior burster and pyloric dilator neurons (AB/PD, blue). There are two types of follower neurons, lateral pyloric (LP, orange), and pyloric (PY, green). Neurons are connected via slow cholinergic (thick lines) and fast glutamatergic (thin lines) synapses. **(b)** Activity of the simulated neurons. Membrane potential is plotted over time for the neurons in (a). The bottom row shows their spiking activity in a raster plot, with spikes defined as excursions of the membrane potential over −20 mV. **(c)** Activity of the simulated neurons of a biologically detailed version of the circuit shown in (a), following Golowasch et al. (1999).

Our third requirement is computational efficiency. Often, computational neuroscience research is limited more by the scientist's time spent designing and implementing models, and analysing results, rather than the simulation time. However, there are occasions where high computational efficiency is necessary. To achieve high performance while preserving maximum flexibility, Brian generates code from user-defined equations and integrates it into the simulation flow.

Our final requirement is extensibility: no simulator can implement everything that every user might conceivably want, but users shouldn't have to discard the simulator entirely if they want to go beyond its built-in capabilities. We therefore provide the possibility for users to extend the code either at a high or low level. We illustrate these last two requirements at the *implementation level* with a case study of a model of pitch perception using real-time audio input (***Figure 6***, ***Figure 7***).

In this section, we give a high level overview of the major decisions. A detailed analysis of the case studies and the features of Brian they use can be found in Appendix A.

Mathematical level

Case study: Pyloric network

We start with a case study of a model of the pyloric network of the crustacean stomatogastric ganglion (***Figure 1***a), adapted and simplified from earlier studies (Golowasch et al., 1999; Prinz et al., 2004; Prinz, 2006; O'Leary et al., 2014). This network has a small number of well characterized neuron types – anterior burster (AB), pyloric dilator (PD), lateral pyloric (LP), and pyloric (PY) neurons – and is known to generate a stereotypical triphasic motor pattern (***Figure 1***b–c). Following previous studies, we lump AB and PD neurons into a single neuron type (AB/PD) and consider a circuit with one neuron of each type. The neurons in this circuit have rebound and bursting properties. We model this using a variant of the model proposed by Hindmarsh & Rose (1984), a three-variable model exhibiting such properties. We make this choice only for simplicity: the biophysical equations originally used in Golowasch et al. (1999) can be used instead (see ***Figure Supplement 1***).

Although this model is based on widely used neuron models, it has the unusual feature that some of the conductances are regulated by activity as monitored by a calcium trace. One of the first design requirements of Brian, then, is that non-standard aspects of models such as this should be as easy to implement in code as they are to describe in terms of their mathematical equations. We briefly summarise how it applies to this model (see appendix A and Stimberg et al. (2014) for more detail). The three-variable underlying neuron model is implemented by writing its differential equations directly in standard mathematical

```
from brian2 import *
defaultclock.dt = 0.01*ms;
Delta_T = 17.5*mV        ; v_T = -40*mV       ; tau = 2*ms          ; tau_adapt = .02*second
tau_Ca = 150*ms          ; tau_x = 2*second   ; v_r = -68*mV        ; tau_z = 5*second
a = 1/Delta_T**3         ; b = 3/Delta_T**2   ; c = 1.2*nA          ; d = 2.5*nA/Delta_T**2
C = 60*pF                ; S = 2*nA/Delta_T   ; G = 28.5*nS
eqs = '''
dv/dt = (Delta_T*g*(-a*(v - v_T)**3 + b*(v - v_T)**2) + w - x - I_fast - I_slow)/C : volt
dw/dt = (c - d*(v - v_T)**2 - w)/tau : amp
dx/dt = (s*(v - v_r) - x)/tau_x : amp
s = S*(1 - tanh(z)) : siemens
g = G*(1 + tanh(z)) : siemens
dCa/dt = -Ca/tau_Ca : 1
dz/dt = tanh(Ca - Ca_target)/tau_z : 1
I_fast : amp
I_slow : amp
Ca_target : 1 (constant)
label : integer (constant)
'''
ABPD, LP, PY = 0, 1, 2
circuit = NeuronGroup(3, eqs, threshold='v>-20*mV', refractory='v>-20*mV', reset='Ca += 0.1',
                      method='rk2')
circuit.label = [ABPD, LP, PY]
circuit.v = v_r
circuit.w = '-5*nA*rand()'
circuit.z = 'rand()*0.2 - 0.1'
circuit.Ca_target = [0.048, 0.0384, 0.06]

s_fast = 0.2/mV; V_fast = -50*mV; E_syn = -75*mV
eqs_fast = '''
g_fast : siemens (constant)
I_fast_post = g_fast*(v_post - E_syn)/(1+exp(s_fast*(V_fast-v_pre))) : amp (summed)
'''
fast_synapses = Synapses(circuit, circuit, model=eqs_fast)
fast_synapses.connect('label_pre != label_post and not (label_pre == PY and label_post == ABPD)')
fast_synapses.g_fast['label_pre == ABPD and label_post == LP'] = 0.015*uS
fast_synapses.g_fast['label_pre == ABPD and label_post == PY'] = 0.005*uS
fast_synapses.g_fast['label_pre == LP and label_post == ABPD'] = 0.01*uS
fast_synapses.g_fast['label_pre == LP and label_post == PY']   = 0.02*uS
fast_synapses.g_fast['label_pre == PY and label_post == LP']   = 0.005*uS

s_slow = 1/mV; V_slow = -55*mV; k_1 = 1/ms
eqs_slow = '''
k_2 : 1/second (constant)
g_slow : siemens (constant)
I_slow_post = g_slow*m_slow*(v_post-E_syn) : amp (summed)
dm_slow/dt = k_1*(1-m_slow)/(1+exp(s_slow*(V_slow-v_pre))) - k_2*m_slow : 1 (clock-driven)
'''
slow_synapses = Synapses(circuit, circuit, model=eqs_slow, method='exact')
slow_synapses.connect('label_pre == ABPD and label_post != ABPD')
slow_synapses.g_slow['label_post == LP'] = 0.025*uS
slow_synapses.k_2['label_post == LP']    = 0.03/ms
slow_synapses.g_slow['label_post == PY'] = 0.015*uS
slow_synapses.k_2['label_post == PY']    = 0.008/ms

run(59.5*second)
```

Figure 2. Case study: a model of the pyloric network of the crustacean stomatogastric ganglion. Simulation code for the model shown in ***Figure 1***a, producing the circuit activity shown in ***Figure 1***b.

Figure 2–Figure supplement 1. Simulation code for the more biologically detailed model of the circuit shown in ***Figure 1***a, producing the circuit activity shown in ***Figure 1***c

form (***Figure 2***, l. 8–10). The calcium trace increases at each spike (l. 21; defined by a discrete event triggered after a spike, `reset='Ca += 0.1'`) and then decays (l. 13; again defined by a differential equation). A slow variable z tracks the difference of this calcium trace to a neuron-type-specific target value (l. 14) which then regulates the conductances s and g (l. 11–12).

Not only the neuron model but also their connections are non-standard. Neurons are connected together by nonlinear graded synapses of two different types, slow and fast (l. 29–54). These are unconventional synapses in that the synaptic current has a graded dependence on the pre-synaptic action potential and a continuous effect rather than only being triggered by pre-synaptic action potentials (Abbott & Marder, 1998). A key design requirement of Brian was to allow for the same expressivity for synaptic models as for neuron models, which led us to a number of features that allow for a particularly flexible specification of synapses in Brian. Firstly, we allow synapses to have dynamics defined by differential equations in precisely the same way as neurons. In addition to the usual role of triggering instantaneous changes in response to discrete neuronal events such as spikes, synapses can directly and continuously modify neuronal variables allowing for a very wide range of synapse types. To illustrate this, for the slow synapse, we have a synaptic variable (`m_slow`) that evolves according to a differential equation (l. 47) that depends on the pre-synaptic membrane potential (`v_pre`). The effect of this synapse is defined by setting the value of a post-synaptic neuron current (`I_slow`) in the definition of the synapse model (l. 46; referred to there as `I_slow_post`). The keyword `(summed)` in the equation specifies that the post-synaptic neuron variable is set using the summed value of the expression across all the synapses connected to it. Note that this mechanism also allows Brian to be used to specify abstract rate-based neuron models in addition to biophysical graded synapse models.

The model is defined not only by its dynamics, but also the values of parameters and the connectivity pattern of synapses. The next design requirement of Brian was that these essential elements of specifying a model should be equally flexible and readable as the dynamics. In this case, we have added a label variable to the model that can take values ABPD, LP or PY (l. 18, 20, 23) and used this label to set up the initial values (l. 36–40, 51–54) and connectivity patterns (l. 35, 50). Human readability of scripts is a key aspect of Brian code, and important for reproducibility (which we will come back to in the Discussion). We highlight line 35 to illustrate this. We wish to have synapses between all neurons of different types but not of the same type, except that we do not wish to have synapses from PY neurons to AB/PD neurons. Having set up the labels, we can now express this connectivity pattern with the expression `'label_pre!=label_post and not (label_pre==PY and label_post==ABPD)'`. This example illustrates one of the many possibilities offered by the equation-oriented approach to concisely express connectivity patterns (for more details see Appendix A and Stimberg et al. (2014)).

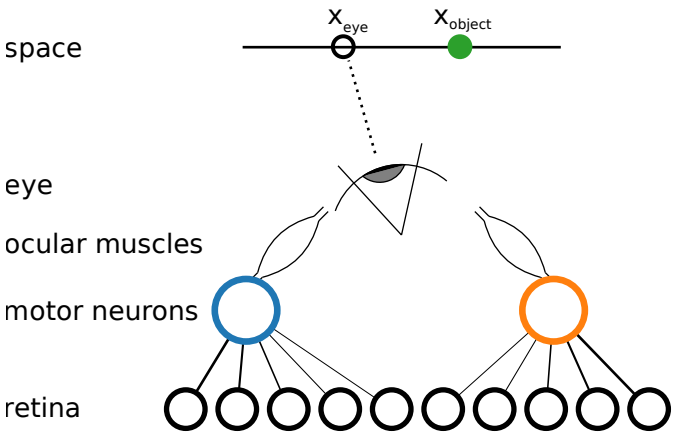
Case study: Ocular model

The second example is a closed-loop sensorimotor model of ocular movements (used for illustration and not intended to be a realistic description of the system), where the eye tracks an object (***Figure 3***a, b). Thus, in addition to neurons, the model also describes the activity of ocular muscles and the dynamics of the stimulus. Each of the two antagonistic muscles is modelled mechanically as an elastic spring with some friction, which moves the eye laterally. The next design requirement of Brian was that it should be both possible and straightforward to define non-neuronal elements of a model, as these are just as essential to the model as a whole, and the importance of connecting with these elements is often neglected in neural simulators. We will come back to this requirement in various forms over the next few case studies, but here we emphasise how the mechanisms for specifying arbitrary differential equations can be re-used for non-neuronal elements of a simulation.

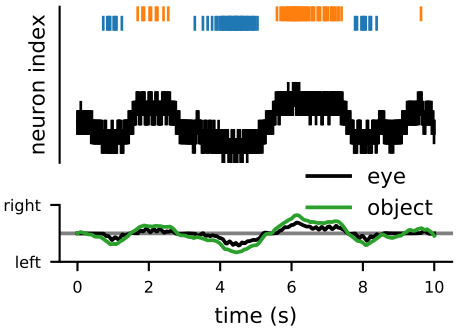
The position of the eye follows a second order differential equation, with resting position x_0, the difference in resting positions of the two muscles (***Figure 3***c, l. 4–5). The stimulus is an object that moves in front of the eye according to a stochastic process (l. 7–8). Muscles are controlled by two motoneurons (l. 11–13), for which each spike triggers a muscular "twitch". This corresponds to a transient change in the resting position x_0 of the eye in either direction, which then decays back to zero (l. 6, 15).

Retinal neurons receive a visual input, modelled as a Gaussian function of the difference between the neuron's preferred position and the actual position of the object, measured in retinal coordinates (l. 21). Thus, the input to the neurons depends on

(a)



(b)



(c)

```
from brian2 import *

alpha = (1/(50*ms))**2; beta = 1/(50*ms); tau_muscle = 20*ms; tau_object = 500*ms
eqs_eye = '''dx/dt = velocity : 1
             dvelocity/dt = alpha*(x0-x)-beta*velocity : 1/second
             dx0/dt = -x0/tau_muscle : 1
             dx_object/dt = (noise - x_object)/tau_object:  1
             dnoise/dt = -noise/tau_object + tau_object**-0.5*xi : 1'''
eye = NeuronGroup(1, model=eqs_eye, method='euler')

taum = 20*ms
motoneurons = NeuronGroup(2, model='dv/dt = -v/taum : 1', threshold='v>1', reset='v=0',
                          refractory=5*ms, method='exact')

motosynapses = Synapses(motoneurons, eye, model='w : 1', on_pre='x0_post += w')
motosynapses.connect() # connects all motoneurons to the eye
motosynapses.w = [-0.5, 0.5]

N = 20; width = 2./N; gain = 4.
eqs_retina = '''dv/dt = (I-(1+gs)*v)/taum : 1
                I = gain*exp(-((x_object-x_eye-x_neuron)/width)**2) : 1
                x_neuron : 1 (constant)
                x_object : 1 (linked) # position of the object
                x_eye : 1 (linked) # position of the eye
                gs : 1 # total synaptic conductance'''
retina = NeuronGroup(N, model=eqs_retina, threshold='v>1', reset='v=0', method='exact')
retina.v = 'rand()'
retina.x_eye = linked_var(eye, 'x')
retina.x_object = linked_var(eye, 'x_object')
retina.x_neuron = '-1.0 + 2.0*i/(N-1)'

sensorimotor_synapses = Synapses(retina, motoneurons, model='w : 1 (constant)', on_pre='v_post += w')
sensorimotor_synapses.connect(j='int(x_neuron_pre > 0)')
# Strength scales with eccentricity:
sensorimotor_synapses.w = '20*abs(x_neuron_pre)/N_pre'

run(10*second)
```

Figure 3. Case Study: Smooth pursuit eye movements. **(a)** Schematics of the model. An object (green) moves along a line and activates retinal neurons (bottom row; black) that are sensitive to the relative position of the object to the eye. Retinal neurons activate two motor neurons with weights depending on the eccentricity of their preferred position in space. Motor neurons activate the ocular muscles responsible for turning the eye. **(b)** Top: Simulated activity of the sensory neurons (black), and the left (blue) and right (orange) motor neurons. Bottom: Position of the eye (black) and the stimulus (green). **(c)** Simulation code.

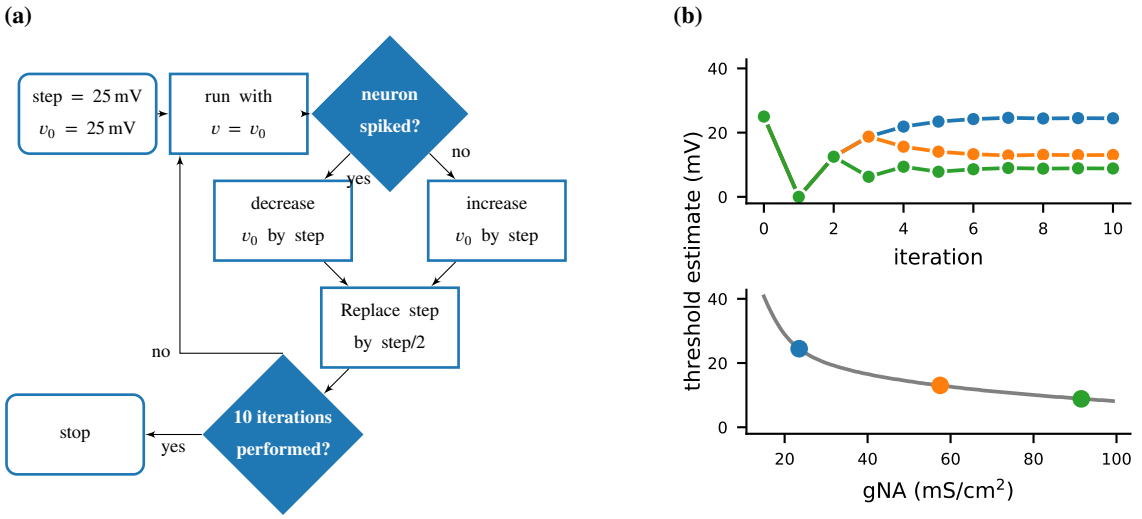


Figure 4. Case study: Using bisection to find a neuron's voltage threshold. **(a)** Schematic of the bisection algorithm for finding a neuron's voltage threshold. The algorithm is applied in parallel for different values of sodium density. **(b)** Top: Refinement of the voltage threshold estimate over iterations for three sodium densities (blue: 23.5 mS cm^{-2}, orange: 57.5 mS cm^{-2}, green: 91.5 mS cm^{-2}); Bottom: Voltage threshold estimation as a function of sodium density.

dynamical variables external to the neuron model. This is a further illustration of the design requirement above that we need to include non-neuronal elements in our model specifications. In this case, to achieve this we link the variables in the eye model with the variables in the retina model using the `linked_var` function (l. 4, 7, 23–24, 28–29).

Finally, we implement a simple feedback mechanism by having retinal neurons project onto the motoneuron controlling the contralateral muscle (l. 33), with a strength proportional to their eccentricity (l. 35): thus, if the object appears on the edge of the retina, the eye is strongly pulled towards the object; if the object appears in the center, muscles are not activated. This simple mechanism allows the eye to follow the object (***Figure 3***b), and the code illustrates the previous design requirement that the code should reflect the mathematical description of the model.

Computational experiment level

The mathematical model descriptions discussed in the previous section provide only a partial description of what we might call a "computational experiment". Let us consider the analogy to an electrophysiological experiment: for a full description, we would not only state the model animal, the cell type and the preparation that was investigated, but also the stimulation and analysis protocol. In the same way, a full description of a computational experiment requires not only a description of the neuron and synapse models, but also information such as how input stimuli are generated, or what sequence of simulations is run. Capturing all these potential protocols in a single declarative framework is impossible, but it can be easily expressed in a programming language with control structures such as loops and conditionals. The Brian simulator allows the user to write complete computational experimental protocols that include both the model description and the simulation protocol in a single, readable Python script.

2.0.1 Case study: Threshold finding

In this case study, we want to determine the voltage firing threshold of a neuron (***Figure 4***), modelled with three conductances, a passive leak conductance and voltage-dependent sodium and potassium conductances (***Figure 5*** l. 4–24).

To get an accurate estimate of the threshold, we use a bisection algorithm (***Figure 4***a): starting from an initial estimate and with an initial step width (***Figure 5***, l. 30–31), we set the neuron's membrane potential to the estimate (l. 35) and simulate its dynamics for 20 ms (l. 36). If the neuron spikes, i.e. if the estimate was above the neuron's threshold, we decrease our

```
from brian2 import *
defaultclock.dt = 0.01*ms

El = 10.613*mV; ENa = 115*mV; EK = -12*mV
gl = 0.3*mS/cm**2; gK = 36*mS/cm**2; C = 1*uF/cm**2
gNa0 = 120*mS/cm**2; gNa_min = 15*mS/cm**2; gNa_max = 100*mS/cm**2

eqs = '''dv/dt = (gl*(El - v) + gNa*m**3*h*(ENa - v) + gK*n**4*(EK - v)) / C : volt
         gNa : siemens/meter**2
         dm/dt = alpham*(1 - m) - betam*m : 1
         dn/dt = alphan*(1 - n) - betan*n : 1
         dh/dt = alphah*(1 - h) - betah*h : 1
         alpham = (0.1/mV)*(-v + 25*mV)/(exp((-v + 25*mV)/(10*mV)) - 1)/ms : Hz
         betam = 4 * exp(-v/(18*mV))/ms : Hz
         alphah = 0.07 * exp(-v/(20*mV))/ms : Hz
         betah = 1/(exp((-v+30*mV) / (10*mV)) + 1)/ms : Hz
         alphan = (0.01/mV) * (-v+10*mV) / (exp((-v+10*mV) / (10*mV)) - 1)/ms : Hz
         betan = 0.125*exp(-v/(80*mV))/ms : Hz'''
neurons = NeuronGroup(100, eqs, method='exponential_euler', threshold='v>50*mV')
neurons.gNa = 'gNa_min + (gNa_max - gNa_min)*1.0*i/N'
neurons.v = 0*mV
neurons.m = '1/(1 + betam/alpham)'
neurons.n = '1/(1 + betan/alphan)'
neurons.h = '1/(1 + betah/alphah)'
S = SpikeMonitor(neurons)

store()

# We locate the threshold by bisection
v0 = 25*mV*ones(len(neurons))
step = 25*mV

for i in range(10):
    restore()
    neurons.v = v0
    run(20*ms)
    v0[S.count == 0] += step
    v0[S.count > 0] -= step
    step /= 2.0
```

Figure 5. Simulation code to find a neuron's voltage threshold, implementing the bisection algorithm detailed in ***Figure 4***a. The code simulates 100 unconnected axon compartments with sodium densities between 15 mS cm^{-2} and 100 mS cm^{-2}, following the model of Hodgkin & Huxley (1952). Results from these simulations are shown in ***Figure 4***b.

estimate (l. 38); if the neuron does not spike, we increase it (l. 37). We then halve the step width (l. 39) and perform the same process again until we have performed a certain number of iterations (l. 33) and converged to a precise estimate (***Figure 4***b top). Note that the order of operations is important here. When we modify the variable v in lines 37–38, we use the output of the simulation run on line 36, and this determines the parameters for the next iteration. A purely declarative definition could not represent this essential feature of the computational experiment.

For each iteration of this loop, we restore the network state (`restore()`; l. 34) to what it was at the beginning of the simulation (`store()`; l. 27). This `store()`/`restore()` mechanism is a key part of Brian's design for allowing computational experiments to be easily and flexibly expressed in Python, as it gives a very effective way of representing common computational experimental protocols. Examples that can easily be implemented with this mechanism include a training/testing/validation cycle in a synaptic plasticity setting; repeating simulations with some aspect of the model changed but the rest held constant (e.g. parameter sweeps, responses to different stimuli); or simply repeatedly running an identical stochastic simulation to evaluate its statistical properties.

At the end of the script, by performing this estimation loop in parallel for many neurons, each having a different maximal sodium conductance, we arrive at an estimate of the dependence of the voltage threshold on the sodium conductance (***Figure 4***b bottom).

Implementation level

2.0.2 Case study: Real-time audio

The case studies so far were described by equations and algorithms on a level that is independent of the programming language and hardware that will eventually perform the computation. However, in some cases this lower level cannot be ignored. To demonstrate this, we will consider the example presented in ***Figure 6***. We want to record an audio signal with a microphone and feed this signal—in real-time—into a neural network performing a crude "pitch detection" based on the autocorrelation of the signal (Licklider, 1962). This model first transforms the continuous stimulus into a sequence of spikes by feeding the stimulus into an integrate-and-fire model with an adaptive threshold (***Figure 7***, l. 36–41). It then detects periodicity in this spike train by feeding it into an array of coincidence detector neurons (***Figure 6***a; ***Figure 7***, l. 44–47). Each of these neurons receives the input spike train via two pathways with different delays (l. 49–51). This arrangement allows the network to detect periodicity in the input stimulus; a periodic stimulus will most strongly excite the neuron where the difference in delays matches the stimulus' period. Depending on the periodicity present in the stimulus, e.g. for tones of different pitch (***Figure 6***b middle), different sub-populations of neurons respond (***Figure 6***b bottom).

To perform such a study, our simulator has to meet two new requirements: firstly, the simulation has to run fast enough to be able to process the audio input in real-time. Secondly, we need a way to connect the running simulation to an audio signal via low-level code.

The challenge is to make the computational efficiency requirement compatible with the requirement of flexibility. With version 1 of Brian, we made the choice to sacrifice computational efficiency, because we reasoned that frequently in computational modelling, considerably more time was spent developing the model and writing the code than was spent on running it (often weeks versus minutes or hours) (cf. De Schutter, 1992). However, there are obviously cases where simulation time is a bottleneck. To increase computational efficiency without sacrificing flexibility, We decided to make code generation the fundamental mode of operation for Brian 2 (Stimberg et al., 2014). Code generation was used previously in Brian 1 (Goodman, 2010), but only in parts of the simulation. This technique is now being increasingly widely used in other simulators, see Blundell et al. (2018) for a review.

In brief, from the high level abstract description of the model, we generate independent blocks of code (in C++ and other languages) that, when run in sequence, carry out the simulation. To generate this code, we make use of a combination of various techniques from symbolic mathematics and compilers that are available in third party Python libraries, as well as

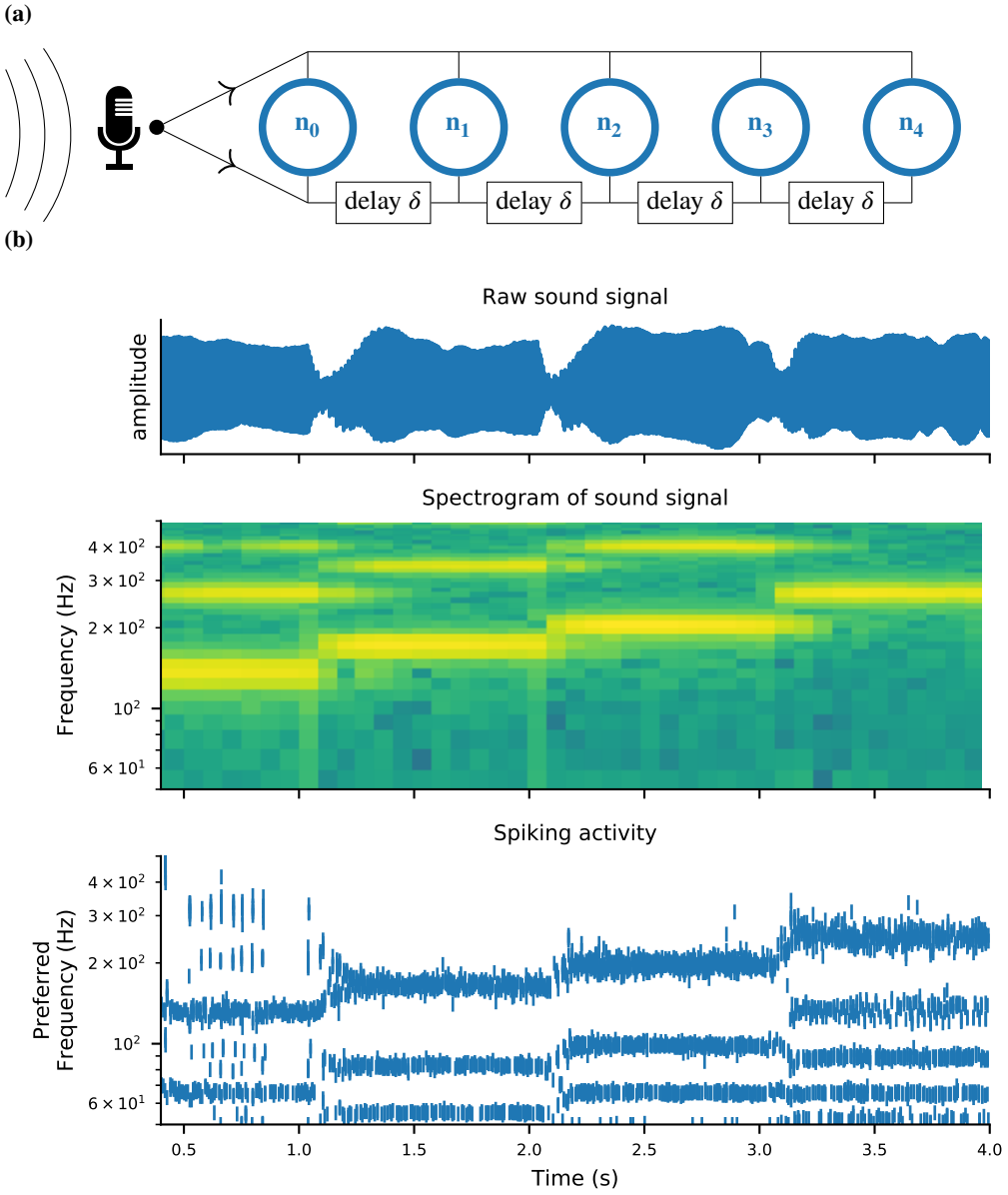


Figure 6. Case study: Neural pitch processing with real-time input. **(a)** Model schematic: Audio input is converted into spikes and fed into a population of coincidence-detection neurons via two pathways, one instantaneous, i.e. without any delay (top), and one with incremental delays (bottom). Each neuron therefore receives the spikes resulting from the audio signal twice, with different temporal shifts between the two. The inverse of this shift determines the preferred frequency of the neuron. **(b)** Simulation results for a sample run of the simulation code in ***Figure 7***. Top: Raw sound input (a rising sequence of tones – C, E, G, C – played on a synthesized flute). Middle: Spectrogram of the sound input. Bottom: Raster plot of the spiking response of receiving neurons (group `neurons` in the code), ordered by their preferred frequency.

```
from brian2 import *
import os
set_device('cpp_standalone')

sample_rate = 48*kHz; buffer_size = 128; defaultclock.dt = 1/sample_rate
max_delay = 20*ms; tau_ear = 1*ms; tau_th = 5*ms
min_freq = 50*Hz; max_freq = 1000*Hz; num_neurons = 300; tau = 1*ms; sigma = .1

@implementation('cpp','''
PaStream *_init_stream() {
    PaStream* stream;
    Pa_Initialize();
    Pa_OpenDefaultStream(&stream, 1, 0, paFloat32, SAMPLE_RATE, BUFFER_SIZE, NULL, NULL);
    Pa_StartStream(stream);
    return stream;
}

float get_sample(const double t) {
    static PaStream* stream = _init_stream();
    static float buffer[BUFFER_SIZE];
    static int next_sample = BUFFER_SIZE;

    if (next_sample >= BUFFER_SIZE)
    {
        Pa_ReadStream(stream, buffer, BUFFER_SIZE);
        next_sample = 0;
    }
    return buffer[next_sample++];
}''', libraries=['portaudio'], headers=['<portaudio.h>'],
      define_macros=[('BUFFER_SIZE', buffer_size),
                     ('SAMPLE_RATE', sample_rate)])
@check_units(t=second, result=1)
def get_sample(t):
    raise NotImplementedError('Use a C++-based code generation target.')

eqs_ear = '''dx/dt = (sound - x)/tau_ear: 1 (unless refractory)
             dth/dt = (0.1*x - th)/tau_th : 1
             sound = clip(get_sample(t), 0, inf) : 1 (constant over dt)'''
receptors = NeuronGroup(1, eqs_ear, threshold='x>th',
                        reset='x=0; th = th*2.5 + 0.01',
                        refractory=2*ms, method='exact')
receptors.th = 1

eqs_neurons = '''dv/dt = -v/tau+sigma*(2./tau)**.5*xi : 1
                 freq : Hz (constant)'''
neurons = NeuronGroup(num_neurons, eqs_neurons, threshold='v>1', reset='v=0', method='euler')
neurons.freq = 'exp(log(min_freq/Hz)+(i*1.0/(num_neurons-1))*log(max_freq/min_freq))*Hz'

synapses = Synapses(receptors, neurons, on_pre='v += 0.5', multisynaptic_index='k')
synapses.connect(n=2)  # one synapse without delay; one with delay
synapses.delay['k == 1'] = '1/freq_post'

run(10*second)
```

Figure 7. Simulation code for the model shown in ***Figure 6***a. The sound input is acquired in real time from a microphone, using user-provided low-level code written in C that makes use of an Open Source library for audio input (Bencina et al., 1999–).

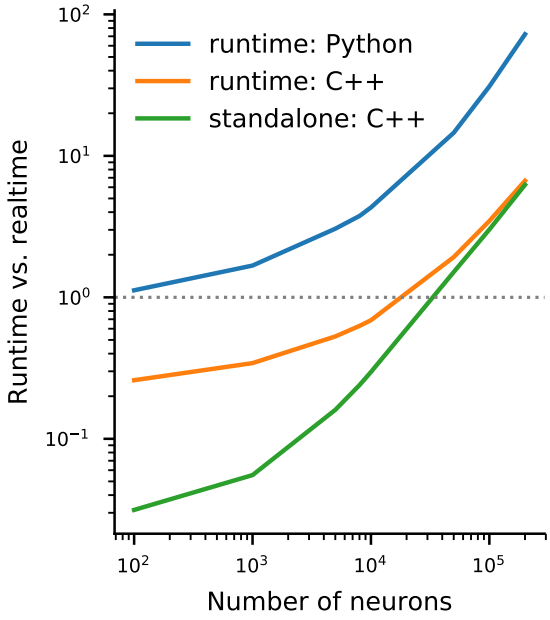


Figure 8. Benchmark of the simulation time for the CUBA network (Vogels & Abbott, 2005; Brette et al., 2007) on a Intel® Xeon(R) CPU E5-1630. This benchmark measures the simulation time relative to the simulated biological time, not taking account any preparation time for compilation, synapse creation etc. The simulation has been executed in runtime mode generating either Python (blue) or C++ (orange) code, or in the standalone mode for C++ code (green). Note that this example has been adapted to use a connection probability that scales with the size of the network. Every neuron targets on average 80 other neurons.

some domain specific optimisations to further improve performance (see Appendix A for more details). These "code objects" implement fundamental operations such as numerically integrating a group of differential equations from time t to time $t + \Delta t$, or propagating the effect of spikes via synapses. We can then run the complete simulation in one of two modes.

In *runtime* mode, the overall simulation is controlled by Python code, which calls out to the compiled code objects to do the heavy lifting. This method of running the simulation is the default, because despite some computational overhead associated with repeatedly switching from Python to another language, it allows for a great deal of flexibility in how the simulation is run: whenever Brian's model description formalism is not expressive enough for a task at hand, the researcher can interleave the execution of generated code with a hand-written function that can potentially access and modify any aspect of the model. This facility is widely used in computational models using Brian.

In *standalone* mode, additional low-level code is generated that controls the overall simulation, meaning that during the main run of the simulation it is not necessary to switch back to Python. This gives an improvement to performance, but at the cost of reduced flexibility since we cannot translate arbitrary Python code into low level code. The standalone mode can also be used to generate code to run on a platform where Python is not available or not practical (such as a GPU; Stimberg et al. 2018).

The choice of which mode to use is left to the user, and will depend on details of the simulation and how much additional flexibility is required. In ***Figure 8*** we see a fairly common pattern in performance, that in small networks the standalone mode can be dramatically faster, with diminishing returns at larger network sizes where the fixed cost Python overhead is smaller relative to the total time.

The second issue we needed to address for this case study was how to connect the running simulation to an audio signal via low-level code. The general issue here is how to extend the functionality of Brian. While Brian's syntax allows a researcher to define a wide range of models within its general framework, it inevitably will not be sufficient for all computational research projects. Taking this into account, Brian has been built with extensibility in mind. Importantly, it should be possible to extend

Brian's functionality and still include the full description of the model in the main Python script, i.e. without requiring the user to edit the source code of the simulator itself or to add and compile separate modules. As discussed previously, the runtime mode offers researchers the possibility to combine their simulation code with arbitrary Python code. However, in some cases such as a model that requires real-time access to hardware (***Figure 6***), it may be necessary to add functionality at the target-language level itself. To this end, simulations can use a general extension mechanism: model code can refer not only to predefined mathematical functions, but also to functions defined in the target language by the user (***Figure 7***, l. 9–34). This can refer to code external to Brian, e.g. to third-party libraries (as is necessary in this case to get access to the microphone). In order to establish the link, Brian allows the user to specify additional libraries , header files or macro definitions (l. 29–31) that will be taken into account during the compilation of the code. With this mechanism the Brian simulator offers researchers the possibility to add functionality to their model at the lowest possible level, without abandoning the use of a convenient simulator and forcing them to write their model "from scratch" in a low-level language. We think it is important to acknowledge that a simulator will never have every possible feature to cover all possible models, and we therefore provide researchers with the means to adapt the simulator's behaviour to their needs at every level of the simulation.

3 Discussion

Brian 2 was designed to overcome some of the major challenges we saw for neural simulators (including Brian 1). Notably: the flexibility/performance dichotomy (including the use of non-standard computational hardware such as GPUs that are increasingly important in computational science); and the need to integrate complex computational experiments that go beyond their neuronal and network components. As a result of this work, Brian can address a wide range of modelling problems faced by neuroscientists, as well as giving more robust and reproducible results and therefore contributing to a solution to the crisis of reproducibility in computational science. We now discuss these challenges in more detail.

Brian's code generation framework allows for a solution to the dichotomy between flexibility and performance. Flexibility is essential to be useful for fundamental research in neuroscience, where basic concepts and models are still being actively investigated and have not settled to the point where they can be standardised. Performance is increasingly important, for example as researchers begin to model larger scale experimental data such as that provided by the Neuropixels probe (Jun et al., 2017), or when doing comprehensive parameter sweeps to establish robustness of models (O'Leary et al., 2015). The focus of Brian 1 was on flexibility, with performance a secondary concern. Brian 2 improves on Brian 1 both in terms of flexibility (particularly the new, very general synapse model) and performance, where it performs similarly to simulators written in low-level languages which do not have the same flexibility (Tikidji-Hamburyan et al., 2017).

The modular structure of the code generation framework is also designed to be proof against future trends in both high performance computing and computational neuroscience research. Increasingly, high performance scientific computing relies on the use of heterogeneous computing architectures such as GPUs, FPGAs, and even more specialised hardware (Fidjeland et al., 2009; Richert et al., 2011; Brette & Goodman, 2012; Moore et al., 2012; Furber et al., 2014; Cheung et al., 2016), as well as techniques such as approximate computing (Mittal, 2016). In addition to the existing standalone mode, it is possible to write plugins for Brian to generate code for these platforms and techniques without modifying the core code, and there are several ongoing projects to do so. These include Brian2GeNN (Stimberg et al., 2018) which uses the GPU-enhanced Neural Network simulator (GeNN; Yavuz et al. 2016) to accelerate simulations in some cases by tens to hundreds of times, and Brian2CUDA (`https://github.com/brian-team/brian2cuda`). In addition to basic research, spiking neural networks may increasingly be used in applications thanks to their low power consumption (Merolla et al., 2014), and the standalone mode of Brian is designed to facilitate the process of converting research code into production code.

A neural computational model is more than just its components (neurons, synapses, etc.) and network structure. In designing Brian, we put a strong emphasis on the complete computational experiment, including specification of the stimulus, interaction with non-neuronal components, etc. This is important both to minimise the time and expertise required to develop computational

models, but also to reduce the chance of errors (see below). Part of our approach here was to ensure that features in Brian are as general and flexible as possible. For example the equations system intended for defining neuron models can easily be repurposed for defining non-neuronal elements of a computational experiment (***Figure 3***). However, ultimately we recognise that any way of specifying all elements of a computational experiment would be at least as complex as a fully featured programming language. We therefore simply allow users to define these aspects in Python, the same language used for defining the neural components, as this is already highly capable and readable. We made great efforts to ensure that the detailed work in designing and implementing new features should not interfere with the goal that the user script should be a readable description of the complete computational experiment, as we consider this to be an essential element of what makes a computational model valuable.

Finally, a major issue in computational science generally, and computational neuroscience in particular, is the crisis of reproducibility of computational models (LeVeque et al., 2012; Eglen et al., 2017; Podlaski et al., 2017; Manninen et al., 2018). A frequent complaint of students and researchers at all levels, is that when they try to implement published models using their own code, they get different results. A fascinating and detailed description of one such attempt is given in Pauli et al. (2018). These sorts of problems led to the creation of the ReScience journal, dedicated to publishing replications of previous models or describing when those replication attempts failed (Rougier et al., 2017). A number of issues contribute to this problem, and we designed Brian with these in mind. So, for example, users are required to write equations that are dimensionally consistent, a common source of problems. In addition, by requiring users to write equations explicitly rather than using pre-defined neuron types such as "integrate-and-fire" and "Hodgkin-Huxley", as in other simulators, we reduce the chance that the implementation expected by the user is different to the one provided by the simulator (see discussion below). Perhaps more importantly, by making user-written code simpler and more readable, we increase the chance that the implementation faithfully represents the description of a model. Allowing for more flexibility and targeting the complete computational experiment increases the chances that the entire simulation script can be compactly represented in a single file or programming language, further reducing the chances of such errors. Brian's approach to defining models leads to particularly concise code (Tikidji-Hamburyan et al., 2017), as well as code whose syntax matches closely natural language descriptions of models in papers. This is important not only because it saves scientists time if they have to write less code, but also because such code is easier to verify and reproduce. It is difficult for anyone, the authors of a model included, to verify that thousands of lines of model simulation code match the description they have given of it.

3.1 Comparison to other approaches

We have described some of the key design choices we made for version 2 of the Brian simulator. These represent a particular balance between the conflicting demands of flexibility, ease-of-use, features and performance, and we now compare the results of these choices to other available options for simulations.

There are two main differences of approach between Brian and other simulators. Firstly, we require model definitions to be explicit. Users are required to give the full set of equations and parameters that define the model, rather than using "standard" model names and default parameters (cf. Brette, 2012). This approach requires a slightly higher initial investment of effort from the user, but ensures that users know precisely what their model is doing and reduces the risk of a difference between the implementation of the model and the description of it in a paper (see discussion above).

The second main difference is that we consider the complete computational experiment to be fundamental, and so everything is tightly integrated to the extent that an entire model can be specified in a single, readable file, including equations, protocols, data analysis, etc. In Neuron and NEST, model definitions are separate from the computational experiment script, and indeed written in an entirely different language (see below). This adds complexity and increases the chance of errors. In NeuroML and NineML, there is no way of specifying the computational experiment.

A consequence of the requirement to make model definitions explicit, and an important feature for doing novel research, is that the simulator must support arbitrary user-specified equations. This is available in Neuron via the NMODL description

format (Hines & Carnevale, 2000), and in a limited form in NEST using NESTML (Gewaltig & Diesmann, 2007). In principle, NeuroML and NineML now both include the option for specifying arbitrary equations, although the level of simulator support for these aspects of the standards is unclear. Although some level of support for arbitrary model equations is now fairly widespread in simulators, Brian makes this a fundamental, core concept that is applied universally. One aspect of this approach that is missing from other simulators is the specification of additional defining network features, such as synaptic connectivity patterns, in an equally flexible, equation-oriented way. Neuron is focused on single neuron modeling rather than networks, and only supports directly setting the connectivity synapse-by-synapse. NEST, PyNN (Davison et al., 2008), NeuroML, and NineML support this too, and also include some predefined general connectivity patterns such as one-to-one and all-to-all. NEST further includes a system for specifying connectivity via a "connection set algebra" (Djurfeldt, 2012) allowing for combinations of a few core types of connectivity. However, none have yet followed Brian in allowing the user to specify connectivity patterns via equations, as is commonly done in research papers.

Running compiled code for arbitrary equations means that code generation must be used. This requirement leads to a problem: a simulator that makes use of a fixed set of models can provide hand-optimised implementations of them, whereas a fully flexible simulator must rely on automated techniques. Brian's automatic optimisations are, however, very competitive, and indeed faster than the hand-optimised code of NEST in some cases (Tikidji-Hamburyan et al., 2017). In particular, version 2 of Brian introduces the standalone mode in which the simulator converts the Python model definition into a complete C++ project, and then compiles and runs this for maximum computational efficiency (this mode was not tested in Tikidji-Hamburyan et al. 2017). This standalone code can then be used and further developed entirely independently of Python and Brian, and we include mechanisms to make it simple to extend the generated project to use other code, as in ***Figure 6***, or to embed Brian's code in another project (e.g. as a robot controller). Apart from Brian, this standalone approach has so far only been taken by GeNN and (in an undocumented way) by jLEMS, the former specifically for GPU and the latter intended primarily as a proof of concept for NeuroML. Neither includes a mechanism for easily extending and embedding this code as Brian does.

The main limitation of Brian compared to other simulators is the lack of support for supercomputers and specialised, high performance clusters, which puts a limit on the maximum feasible size of a simulation. However, the majority of neuroscientists do not have direct access to such equipment, and few computational neuroscience studies require such large scale simulations (tens of millions of neurons). More common is to run smaller networks but multiple times over a large range of different parameters. This "embarrassingly parallel" case can be easily and straightforwardly carried out with Brian at any scale, from individual machines to cloud computing platforms or the non-specialised clusters routinely available as part of university computing services.

3.2 Development and availability

Brian is released under the free and open CeCILL 2 license. Development takes place in a public code repository at `https://github.com/brian-team/brian2`. All examples in this article have been simulated with Brian 2 version 2.2.2.1 (Stimberg, Goodman, & Brette, 2019). Brian has a permanent core team of three developers (the authors of this paper), and regularly receives substantial contributions from a number of students, postdocs and users (see Acknowledgements). Code is continuously and automatically checked against a comprehensive test suite run on all platforms, with almost complete coverage. Extensive documentation, including installation instructions, is hosted at `http://brian2.readthedocs.org`. Brian is available for Python 2 and 3, and for the operating systems Windows, OS X and Linux; our download statistics show that all these versions are in active use. More information can be found at `http://briansimulator.org/`.

4 Acknowledgements

We thank the following contributors for having made contributions, big or small, to the Brian 2 code or documentation: Moritz Augustin, Victor Benichoux, Werner Beroux, Edward Betts, Daniel Bliss, Jacopo Bono, Paul Brodersen, Romain

Cazé, Meng Dong, Guillaume Dumas, Adrien F. Vincent, Charlee Fletterman, Dominik Krzemiński, Kapil Kumar, Thomas McColgan, Matthieu Recugnat, Dylan Richard, Cyrille Rossant, Jan-Hendrik Schleimer, Alex Seeholzer, Martino Sorbaro, Daan Sprenkels, Teo Stocco, Mihir Vaidya, Konrad Wartke, Pierre Yger, Friedemann Zenke. Three of these contributors (CF, DK, KK) contributed while participating in Google's Summer of Code program.

This work was supported by Agence Nationale de la Recherche (Axode ANR-14-CE13-0003).

References

Abbott, L. F., & Marder, E. (1998). Modeling small networks. In C. Koch & I. Segev (Eds.), *Methods in Neuronal Modeling* (pp. 361–410). MIT Press, Cambridge, MA, USA.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D., & Smith, K. (2011, March/April). Cython: The best of both worlds. *Computing in Science Engineering*, *13*(2), 31–39. doi: 10.1109/MCSE.2010.118

Bencina, R., Burk, P., et al. (1999–). *PortAudio: Portable real-time audio library.* `http://www.portaudio.com/`.

Blundell, I., Brette, R., Cleland, T. A., Close, T. G., Coca, D., Davison, A. P., … Eppler, J. M. (2018). Code Generation in Computational Neuroscience: A Review of Tools and Techniques. *Frontiers in Neuroinformatics*. doi: 10.3389/fninf.2018.00068

Bower, J. M., & Beeman, D. (1998). *The book of GENESIS: Exploring realistic neural models with the GEneral NEural SImulation System* (2nd ed.). Springer-Verlag.

Brette, R. (2012). On the design of script languages for neural simulation. *Network*, *23*(4), 150–156. doi: 10.3109/0954898X.2012.716902

Brette, R., & Goodman, D. (2012). Simulating spiking neural networks on GPU. *Network: Computation in Neural Systems*, *23*(4). doi: 10.3109/0954898X.2012.730170

Brette, R., & Goodman, D. F. M. (2011, June). Vectorized algorithms for spiking neural network simulation. *Neural Comput*, *23*(6), 1503–1535. doi: 10.1162/NECO_a_00123

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., … Destexhe, A. (2007, December). Simulation of networks of spiking neurons: a review of tools and strategies. *J Comput Neurosci*, *23*(3), 349–398. doi: 10.1007/s10827-007-0038-6

Cannon, R. C., Gleeson, P., Crook, S., Ganapathy, G., Marin, B., Piasini, E., & Silver, R. A. (2014, September). LEMS: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2. *Front Neuroinform*, *8*. doi: 10.3389/fninf.2014.00079

Carnevale, N. T., & Hines, M. L. (2006). *The NEURON Book*. Cambridge University Press.

Cheung, K., Schultz, S. R., & Luk, W. (2016). NeuroFlow: A general purpose spiking neural network simulation platform using customizable processors. *Frontiers in Neuroscience*, *9*(JAN). doi: 10.3389/fnins.2015.00516

Crook, S. M., Bednar, J. A., Berger, S., Cannon, R., Davison, A. P., Djurfeldt, M., … van Albada, S. (2012). Creating, documenting and sharing network models. *Network: Computation in Neural Systems*, *23*(4), 131–149. doi: 10.3109/0954898X.2012.722743

Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., … Yger, P. (2008). PyNN: A Common Interface for Neuronal Network Simulators. *Frontiers in Neuroinformatics*, *2*, 11. doi: 10.3389/neuro.11.011.2008

De Schutter, E. (1992, November). A consumer guide to neuronal modeling software. *Trends in Neurosciences*, *15*(11), 462–464. doi: 10.1016/0166-2236(92)90011-V

Djurfeldt, M. (2012). The Connection-set Algebra—A Novel Formalism for the Representation of Connectivity Structure in Neuronal Network Models. *Neuroinformatics*, *10*(3), 287–304. doi: 10.1007/s12021-012-9146-1

Eglen, S. J., Marwick, B., Halchenko, Y. O., Hanke, M., Sufi, S., Gleeson, P., … Poline, J.-B. (2017). Toward standard practices for sharing computer code and programs in neuroscience. *Nature Neuroscience*. doi: 10.1038/nn.4550

Fidjeland, A. K., Roesch, E. B., Shanahan, M. P., & Luk, W. (2009, jul). NeMo: A platform for neural modelling of spiking neurons using GPUs. In *Proceedings of the international conference on application-specific systems, architectures and processors* (pp. 137–144). doi: 10.1109/ASAP.2009.24

Furber, S. B., Galluppi, F., Temple, S., & Plana, L. A. (2014). The SpiNNaker project. *Proceedings of the IEEE*, *102*(5), 652–665. doi: 10.1109/JPROC.2014.2304638

Gewaltig, M.-O., & Diesmann, M. (2007). Nest (neural simulation tool). *Scholarpedia*, *2*(4), 1430.

Gleeson, P., Crook, S., Cannon, R. C., Hines, M. L., Billings, G. O., Farinella, M., … Silver, R. A. (2010). NeuroML: A language for describing data driven models of neurons and networks with a high degree of biological detail. *PLOS Computational Biology*, *6*(6), 1–19. doi: 10.1371/journal.pcbi.1000815

Golowasch, J., Casey, M., Abbott, L. F., & Marder, E. (1999). Network Stability from Activity-Dependent Regulation of Neuronal Conductances. *Neural Computation*, *11*(5), 1079–1096. doi: 10.1162/089976699300016359

Goodman, D. F. M. (2010, October). Code generation: A strategy for neural network simulators. *Neuroinform*, *8*(3), 183–196. doi: 10.1007/s12021-010-9082-x

Goodman, D. F. M., & Brette, R. (2009). The Brian simulator. *Front Neurosci*, *3*. doi: 10.3389/neuro.01.026.2009

Hettinger, R. (2002). *PEP 289 – Generator Expressions.* Retrieved 2017-09-06, from `https://www.python.org/dev/peps/pep-0289/`

Hindmarsh, J. L., & Rose, R. M. (1984). A model of neuronal bursting using three coupled first order differential equations. *Proceedings of the Royal Society of London. Series B, Biological Sciences*, *221*(1222), 87–102.

Hines, M. L., & Carnevale, N. T. (2000). Expanding NEURON's repertoire of mechanisms with NMODL. *Neural Comput.*, *12*(5), 995–1007. doi: 10.1162/089976600300015475

Hodgkin, A. L., & Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J Physiol*, *117*(4), 500–544.

Jones, E., Oliphant, T., Peterson, P., et al. (2001–). *SciPy: Open source scientific tools for Python.* Retrieved from `http://www.scipy.org/`

Jun, J. J., Steinmetz, N. A., Siegle, J. H., Denman, D. J., Bauza, M., Barbarits, B., … Harris, T. D. (2017, nov). Fully integrated silicon probes for high-density recording of neural activity. *Nature*, *551*(7679), 232–236. doi: 10.1038/nature24636

LeVeque, R. J., Mitchell, I. M., & Stodden, V. (2012, jul). Reproducible research for scientific computing: Tools and strategies for changing the culture. *Computing in Science & Engineering*, *14*(4), 13–17. doi: 10.1109/MCSE.2012.38

Licklider, J. C. R. (1962). Periodicity pitch and related auditory process models. *International Audiology*, *1*(1), 11–34.

Manninen, T., Aćimović, J., Havela, R., Teppola, H., & Linne, M.-L. (2018). Challenges in Reproducibility, Replicability, and Comparability of Computational Models and Tools for Neuronal and Glial Networks, Cells, and Subcellular Structures. *Frontiers in neuroinformatics*, 20. doi: 10.3389/fninf.2018.00020

Merolla, P. A., Arthur, J. V., Alvarez-Icaza, R., Cassidy, A. S., Sawada, J., Akopyan, F., … Modha, D. S. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, *345*(6197), 668–673. doi: 10.1126/science.1254642

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., … Scopatz, A. (2017, January). SymPy: symbolic computing in Python. *PeerJ Comput. Sci.*, *3*, e103. doi: 10.7717/peerj-cs.103

Mittal, S. (2016). A Survey of Techniques for Approximate Computing. *ACM Computing Surveys*, *48*(4), 1–33. doi: 10.1145/2893356

Moore, S. W., Fox, P. J., Marsh, S. J., Markettos, A. T., & Mujumdar, A. (2012). Bluehive - A field-programable custom computing machine for extreme-scale real-time neural network simulation. In *Proceedings of the 2012 ieee 20th international symposium on field-programmable custom computing machines, fccm 2012* (pp. 133–140). doi: 10.1109/FCCM.2012.32

O'Leary, T., Sutton, A. C., & Marder, E. (2015). Computational models in the age of large datasets. *Current Opinion in Neurobiology*, *32*, 87–94. doi: 10.1016/j.conb.2015.01.006

O'Leary, T., Williams, A. H., Franci, A., & Marder, E. (2014). Cell Types, Network Homeostasis, and Pathological Compensation from a Biologically Plausible Ion Channel Expression Model. *Neuron*, *82*(4), 809–821. doi: 10.1016/j.neuron.2014.04.002

Pauli, R., Weidel, P., Kunkel, S., & Morrison, A. (2018). Reproducing Polychronization: A Guide to Maximizing the Reproducibility of Spiking Network Models. *Frontiers in neuroinformatics*, *12*, 46. doi: 10.3389/fninf.2018.00046

Platkiewicz, J., & Brette, R. (2011, May). Impact of Fast Sodium Channel Inactivation on Spike Threshold Dynamics and Synaptic Integration. *PLoS Comput Biol*, *7*(5), e1001129. doi: 10.1371/journal.pcbi.1001129

Plotnikov, D., Blundell, I., Ippen, T., Eppler, J. M., Morrison, A., & Rumpe, B. (2016). Nestml: a modeling language for spiking neurons. In A. Oberweis & R. Reussner (Eds.), *Modellierung 2016* (pp. 93–108). Bonn: Gesellschaft für Informatik e.V.

Podlaski, W. F., Seeholzer, A., Groschner, L. N., Miesenböck, G., Ranjan, R., & Vogels, T. P. (2017, mar). Mapping the function of neuronal ion channels in model and experiment. *eLife*, *6*, e22152. doi: 10.7554/eLife.22152

Prinz, A. A. (2006). Insights from models of rhythmic motor systems. *Current Opinion in Neurobiology*, *16*(6), 615–620. doi: 10.1016/j.conb.2006.10.001

Prinz, A. A., Bucher, D., & Marder, E. (2004). Similar network activity from disparate circuit parameters. *Nat Neurosci*, *7*(12), 1345–1352. doi: 10.1038/nn1352

Raikov, I., Cannon, R., Clewley, R., Cornelis, H., Davison, A., Schutter, E. D., … Szatmary, B. (2011, July). NineML: the network interchange for neuroscience modeling language. *BMC Neuroscience*, *12*(Suppl 1), P330. doi: 10.1186/1471-2202-12-S1-P330

Richert, M., Nageswaran, J. M., Dutt, N., & Krichmar, J. L. (2011). An Efficient Simulation Environment for Modeling Large-Scale Cortical Processing. *Frontiers in Neuroinformatics*, *5*, 19. doi: 10.3389/fninf.2011.00019

Rougier, N. P., Hinsen, K., Alexandre, F., Arildsen, T., Barba, L. A., Benureau, F. C., … Zito, T. (2017, dec). Sustainable computational science: the ReScience initiative. *PeerJ Computer Science*, *3*, e142. doi: 10.7717/peerj-cs.142

Rudolph, M., & Destexhe, A. (2007, June). How much can we trust neural simulation strategies? *Neurocomputing*, *70*(10-12), 1966–1969. doi: 10.1016/j.neucom.2006.10.138

Stimberg, M., Goodman, D. F., & Brette, R. (2019, March). *Brian 2 (version 2.2.2.1).* doi: 10.5281/zenodo.2619969

Stimberg, M., Goodman, D. F., Brette, R., & De Pittà, M. (2019). Modeling neuron–glia interactions with the brian 2 simulator. In M. De Pittà & H. Berry (Eds.), *Computational glioscience* (pp. 471–505). Springer.

Stimberg, M., Goodman, D. F. M., Benichoux, V., & Brette, R. (2014). Equation-oriented specification of neural models for simulations. *Front Neuroinform*, *8*. doi: 10.3389/fninf.2014.00006

Stimberg, M., Goodman, D. F. M., & Nowotny, T. (2018, oct). Brian2GeNN: a system for accelerating a large variety of spiking neural networks with graphics hardware. *bioRxiv*, 448050. Retrieved from `https://www.biorxiv.org/content/early/2018/10/20/448050` doi: 10.1101/448050

Tikidji-Hamburyan, R. A., Narayana, V., Bozkus, Z., & El-Ghazawi, T. A. (2017, jul). Software for Brain Network Simulations: A Comparative Study. *Frontiers in Neuroinformatics*, *11*, 46. doi: 10.3389/fninf.2017.00046

Vogels, T. P., & Abbott, L. F. (2005, November). Signal Propagation and Logic Gating in Networks of Integrate-and-Fire Neurons. *The Journal of Neuroscience*, *25*(46), 10786 –10795. doi: 10.1523/JNEUROSCI.3508-05.2005

Yavuz, E., Turner, J., & Nowotny, T. (2016). GeNN: A code generation framework for accelerated brain simulations. *Sci. Rep.*, *6*, 18854. doi: 10.1038/srep18854

A Design details

In this appendix, we provide further details about technical design decisions behind the Brian simulator. We also more exhaustively comment on the simulation code of the four case studies. Note that the example code provided as jupyter notebooks (`https://github.com/brian-team/brian2_paper_examples`) has extensive additional annotations as well.

Mathematical level

Physical units

Neural models are models of a physical system, and therefore variables have physical dimensions such as voltage or time. Accordingly, the Brian simulator requires quantities provided by the user, such as parameters or initial values of dynamical variables, to be specified in consistent physical units such as mV or s. This is in contrast to the approach of most other simulators, which simply define expected units for all model components, e.g. units of mV for the membrane potential. This is a common source of error because conventions are not always obvious and can be inconsistent. For example, while membrane surface area is often stated in units of μm^2, channel densities are often given in $mS\,cm^{-2}$. To remove this potential source of error, the Brian simulator enforces explicit use of units. It automatically takes care of conversions—multiplying a resistance (dimensions of Ω) with a current (dimensions of A) will result in a voltage (dimensions of V)—and raises an error when physical dimensions are incompatible, e.g. when adding a current to a resistance. Unit consistency is also checked within textual model descriptions (e.g. ***Figure 2***, l. 8–18) and variable assignments (e.g. l. 23–27). To make this possible, a dimension in SI units has to be assigned to each dimensional model variable in the model description (l. 8–18).

Model dynamics

Neuron and synapse models are generally hybrid systems consisting of continuous dynamics described by differential equations and discrete events (Brette et al., 2007).

In the Brian simulator, differential equations are specified in strings using mathematical notation (***Figure 2***, l. 8–18). Differential equations can also be stochastic by using the symbol `xi` representing the noise term $\xi(t)$ (***Figure 3***c, l. 8). The numerical integration method can be specified explicitly, e.g. the pyloric circuit model chooses a second-order Runge-Kutta method (***Figure 2***, l. 22); without specification, an appropriate method is automatically chosen and reported. To this end, the user-provided equations are analysed symbolically using the Python package sympy (Meurer et al., 2017), and transformed into a sequence of operations to advance the system's state by a single time step (for more details, see Stimberg et al., 2014).

This approach applies both to neuron models and to synaptic models. In many models, synaptic conductances do not need to be calculated for each synapse individually, instead they can be lumped into a single post-synaptic variable that is part of the neuronal model description. In contrast, non-linear synaptic dynamics as in the pyloric network example need to be calculated for each synapse individually. Using the same formalism as for neurons, the synaptic model equations can describe dynamics with differential equations (e.g. ***Figure 2***, l. 31–32/44–47). Post-synaptic conductances or currents can then be calculated individually and summed up for each post-synaptic neuron as indicated by the `(summed)` annotation (l. 32 and 46).

Neuron- or synapse-specific values which are not updated by differential equations are also included in the string description. This can be used to define values that are updated by external mechanisms, e.g. the synaptic currents in each neuron (l. 15–16) are updated by the respective synapses (l. 32 and l. 46). The same mechanism can also be used for neuron-specific parameters such as the calcium target value (l. 17), or the label identifying the neuron type (l. 18). For optimisation, the flag `(constant)` can be added to indicate that the value will not change during a simulation.

Neural simulations typically refer to two types of discrete events: production of a spike, and reception of a spike. A spike is produced by a neuron when a certain condition on its variables is met. A typical case is the integrate-and-fire model, where a spike is produced when the potential reaches a threshold of a fixed value. But there are other cases when the condition is more complex, for example when the threshold is adaptive (Platkiewicz & Brette, 2011). To support conditions of all kind, Brian expects the user to define a mathematical expression as the `threshold`. In the pyloric network example, a spike is triggered whenever $v > -20\,\mathrm{mV}$ (***Figure 2***, l. 21). No explicit resetting takes place, since the model dynamics describe the membrane potential trajectory during an action potential. For a simpler integrate-and-fire model as the one used in the example modelling eye movements, the membrane potential is reset to a fixed value after the threshold crossing (***Figure 3***, l. 12). Such spike-triggered actions are most generally specified by providing one or more assignments and operations (`reset`) that should take place if the threshold condition is fulfilled; in the pyloric network example, this is mechanism is used to update the calcium trace (***Figure 2***, l. 21).

Once a spike is produced, it may affect variables of synapses and post-synaptic neurons (possibly after a delay). Again, this is specified generally as a series of assignments and operations. In the pyloric circuit example, this does not apply because the synaptic effect is continuous and not triggered by discrete spikes. In the eye movement example (***Figure 3***) however, each spike has an instantaneous effect. For example, when a motoneuron spikes, the eye resting position is increased or decreased by a fixed amount. This is specified by `on_pre='x0_post += w'` (l. 15), where `on_pre` is a keyword for stating what operations should be executed when a pre-synaptic spike is received. These operations can refer to both local synaptic variables (here w, defined in the synaptic model) and variables of the pre- and postsynaptic neuron (here x_0, a variable of the post-synaptic neuron). In the same way, the `on_post` keyword can be used to specify operations executed when a postsynaptic spike is received, which allows defining various types of spike-timing-dependent models.

This general definition scheme applies to neurons and synapses, but as the eye movement example illustrates (***Figure 3***), it can also be used to define dynamical models of muscles and the environment. It also naturally extends to the modelling of non-neuronal elements of the brain such as glial cells (Stimberg, Goodman, Brette, & De Pittà, 2019).

Links between model components

The equations defining the dynamics of variables can only refer to other variables within the same model component, e.g. within the same group of neurons or synapses. Connections to other components have to be explicitly modelled using synaptic connections as explained above. However, we may sometimes also need to directly refer to the state of variables in other model component. For example, in the eye movement model (***Figure 3***), the input to retinal neurons depends on eye and object positions, which are updated in a group separate from the group representing the retinal neurons (***Figure 3***c, l. 3–9). This can be expressed by defining a "linked variable", which refers to a variable defined in a different model component. In the group modelling the retinal neurons, the variables `x_object` and `x_eye` are annotated with the `(linked)` flag to state that they are references to variables defined elsewhere (l. 23–24). This link is then made explicit by stating the group and variable they refer to via the `linked_var` function (l. 28–29).

Initialisation

The description of its dynamics does not yet completely define a model, we also need to define its initial state. For some variables, this initial state can simply be a fixed value, e.g. in the pyloric network model, the neurons' membrane potential is initialised to the resting potential v_r (***Figure 2*** l. 23). In the general case, however, we might want to calculate the initial state; Brian therefore accepts arbitrary mathematical expressions for setting the initial value of state variables. These expressions can refer to model variables, as well as to pre-defined constant such as the index of a neuron within its group (`i`), or the total number of neurons within a group (`N`), as well as to pre-defined functions such as `rand()` (providing uniformly distributed random numbers between 0 and 1). In the pyloric circuit example, we use this mechanism to initialise variables w and z randomly (***Figure 2***, l. 25–26); in the eye movement example, we assign individual preferred positions to each retinal neuron, covering the space from -1 to 1 in a regular fashion (***Figure 3***c, l. 30).

Mathematical expressions can also be used to select a subset of neurons and synapses and make conditional assignments. In the pyloric circuit example, we assign a specific value to the conductance of synapses between ABPD and LP neurons by using the selection criterion `'label_pre == ABPD and label_post == LP'` (***Figure 2***, l. 36), referring to the custom `label` identifier of the pre- and post-synaptic neuron that has been introduced as part of the neuron model definition (l. 18). In this example there is only a single neuron per type, but the syntax generalises to groups of neurons of arbitrary size and is therefore preferable to the explicit use of numerical indices.

Synaptic connections

The second main aspect of model construction is the creation of synaptic connections. For maximal expressivity, we again allow the use of mathematical expressions to define rules of connectivity. For example, in the pyloric circuit example, following the schematic shown in ***Figure 1***a, we would like to connect neurons with fast glutamatergic synapses according to two rules: 1) connections should occur between all groups, but not within groups of the same neuron type; 2) there should not be any connections from PY neurons to AB/PD neurons. We can express this with a string condition following the same syntax that we used to set initial values for synaptic conductances earlier (***Figure 2***, l. 35):

```
fast.connect('label_pre!=label_post and not (label_pre==PY and label_post==ABPD)')
```

For more complex examples, in particular connection specifications based on the spatial location of neurons, see Stimberg et al. (2014).

For larger networks, it can be wasteful to check a condition for each possible connection. Brian therefore also offers the possibility to use a mathematical expression to directly specify the projections of each neuron. In the eye movement example, each retinal neuron on the left hemifield (i.e. $x_{\text{neuron}} < 0$) should connect to the first motoneuron (index 0), while neurons on the right hemifield (i.e. $x_{\text{neuron}} > 0$) should connect to the second motoneuron (index 1). We can express this connection scheme by

defining j, the postsynaptic target index, for each presynaptic neuron accordingly (with the `int` function converting a truth value into 0 or 1):

```
sensorimotor_synapses.connect(j='int(x_neuron_pre > 0)')
```

This syntax can also be extended to generate more than one post-synaptic target per pre-synaptic neuron, using a syntax borrowed from Python's generator syntax (Hettinger, 2002, see the Brian 2 documentation at `http://brian2.readthedocs.io` for more details) These mechanisms can also be used to define stochastic connectivity schemes, either by specifying a fixed connection probability that will be evaluated in addition to the given conditions, or by specifying a connection probability as a function of pre- and post-synaptic properties.

Specifying synaptic connections in the way presented here has several advantages over alternative approaches. In contrast to explicitly enumerating the connections by referring to pre- and post-synaptic neuron indices, the use of mathematical expressions transparently conveys the logic behind the connection pattern and automatically scales with the size of the connected groups of neurons. These advantages are shared with simulators that provide pre-defined connectivity patterns such as "one-to-one" or "all-to-all". However, such approaches are not as general—e.g. they could not concisely define the connectivity pattern shown in ***Figure 1***a—and can additionally suffer from ambiguity. For example, should a group of neurons that is "all-to-all" connected to itself form autapses or not (cf. Crook et al., 2012)?

Computational experiment level

The Brian simulator allows the user to write complete experiment descriptions that include both the model description and the simulation protocol in a single Python script as exemplified by the case studies in this article. In this section, we will discuss how the Brian simulator interacts with the statements and programming logic expressed in the surrounding script code.

Simulation flow

In the case study of finding a neuron's voltage threshold we use a specific simulation workflow, an iterative approach to finding a parameter value (***Figure 4***a). Many other simulation protocols are regularly used. For example, a simulation might consist of several consecutive runs, where some model aspect such as the external stimulation changes between runs. Alternatively, several different types of models might be tested in a single script where each is run independently. Or, a non-deterministic simulation might be run repeatedly to sample its behaviour. Capturing all these potential protocols in a single descriptive framework is hopeless, we therefore need the flexibility of a programming language with its control structures such as loops and conditionals.

Brian offers two main facilities to assist in implementing arbitrary simulation protocols. Simulations can be continued at their last state, potentially after activating/deactivating model elements, or changing global or group-specific constants and variables as shown above. Additionally, simulations can revert back to a previous state using the functions `store` and `restore` provided by Brian. In the example script shown in ***Figure 5***, this mechanism is used to reset the network to an initial state after each iteration. The same mechanism allows for more complex protocols by referring to multiple states, e.g. to implement a train/test/validate protocol in a synaptic plasticity setting.

Providing explicit support for this functionality is not only a question of convenience; while the user could approximate this functionality by storing and resetting the systems state variables (membrane potentials, gating variables, etc.) manually, some model aspects such as action potentials that have not yet triggered synaptic effects (due to synaptic delays) are not easily accessible to the user.

Model component scheduling

During each time step of a simulation run, several operations have to be performed. These include the numerical integration of the state variables, the propagation of synaptic activity, or the application of reset statements for neurons that emitted an action

potential. All these operations have to be executed in a certain order. The Brian simulator approaches this issue in a flexible and transparent way: each operation has an associated clock with a certain time granularity `dt`, as well as a "scheduling slot" and a priority value within that slot. Together, these elements determine the order of all operations across and within time steps.

By default, all objects are associated with the same clock, which simplifies setting a global simulation timestep for all objects (***Figure 5***, l. 2). However, individual objects may chose a different timestep, e.g. to record synaptic weights only sporadically during a long-running simulation run. In the same way, Brian offers a default ordering of all operations during a time step, but allows to change the schedule that is used, or to reschedule individual objects to other scheduling slots.

This amount of flexibility might appear to be unnecessary at a first glance and indeed details of the scheduling are rarely reported when describing models in a publication. Still, subtle differences in scheduling can have significant impact on simulation results (see ***Figure 9*** in ***section 4*** for an illustration). This is most obvious when investigating paradigms such as spike-timing-dependent-plasticity with a high sensitivity to small temporal differences (Rudolph & Destexhe, 2007).

Name resolution

Model descriptions refer to various "names", such as variables, constants, or functions. Some of these references, such as function names or global constants, will have the same meaning everywhere. Others, such as state variables or neuron indices, will depend on the context. This context is defined by the model component, i.e. the group of neurons or the set of synapses, to which the description is attached. For example, consider the assignment to g_{Na} (the maximum conductance of the sodium channel) in ***Figure 5*** (l. 20). Here, `gNa_min` and `gNa_max` refer to global constants (defined in l. 6)[1], whereas `i`, the neuron index, is a vector of values with one value for each neuron, and `N` refers to the total number of elements in the respective group.

It is important to note that the context is also given by its position in the program flow. For example, if we want to set the initial value for the gating variable m to its steady value, then this value will depend on the membrane potential v via the expressions for α_m and β_m. The order in which we set the values for v and m does therefore matter:

```
neuron.v = 0*mV
neuron.m = '1/(1 + betam/alpham)'
```

While this might appear trivial, it shows how the procedural aspect of models, i.e. the order of operations, can be important. A purely descriptive approach, for example stating initial values for all variables as part of the model equations, would not always be sufficient.[2]

Some Python statements are translated into code that is run immediately, for example initialising a variable or creating synapses. Others are translated into code that is run at a later time. For example, the code to numerically integrate differential equations is not run at the point where those equations are defined, but rather at the point when the simulation is run via a call to the `run()` function. In this case, any named constants referred to in the equations will use their value at the time that the `run()` function is called, and not the value at the time the equations are defined. This allows for that value to change between multiple calls to `run()`, which may be useful to switch between global behaviours. For example, a typical use case is running with no external input current for a certain time to allow a neuron to settle into its stationary state, and then running with the current switched on by just changing the value of a constant from zero to some nonzero value between two consecutive `run()` calls.

Implementation level

Code generation

In order to combine the flexibility and ease-of-use of high-level descriptions with the execution speed of low-level programming languages such as C, we employ a code generation approach (Goodman, 2010). This code generation consists of three steps.

[1] Brian also offers an alternative system where global constants and functions are explicitly provided via a Python dictionary instead of being deduced from values defined in the execution environment, but this system will not be further discussed here.

[2] However, in this specific case, setting v to 0 mV is unnecessary, since Brian automatically assigns the value 0 to all uninitialised variables.

The textual model description will first be transformed into a "code snippet". The generation of such a code snippet requires various transformations of the provided model description: some syntax elements have to be translated (e.g. the use of the `**` operator to denote the power operation to a call to the `pow` function for C/C++), variables that are specific to certain neurons or synapses have to be properly indexed (e.g. a reset statement `v = -70*mV` has to be translated into a statement along the lines of `v[neuron_index] = -70*mV`), and finally sequences of statements have to be expressed according to the target language syntax (e.g. by adding a semicolon to the end of each statement for C/C++). In a second step, these code snippets will then be embedded into a predefined target-code template, specific to the respective computation performed by the code. For example, the user-provided description of an integrate-and-fire neuron's reset would be embedded into a loop that iterates over all the neurons that emitted an action potential during the current time step. Finally, the code has to be compiled and executed, giving it access to the memory location that the code has to read and modify. For further details on this approach, see Goodman (2010); Stimberg et al. (2014).

Code optimisation

Code resulting from the procedure described above will not necessarily perform computations in the most efficient way. Brian therefore uses additional techniques to further optimise the code for performance. Consider for example the x variable—representing the receptor activity—in ***Figure 7***, described by the differential equation in l. 36. This equation can be integrated analytically, and the above described code generation process would therefore generate code like the following (here presented as "pseudo-code"):

```
for each neuron:
    x_new = sound + exp(-dt / tau_ear) * (sound - x_old)
```

However, the expression that is calculated for every neuron contains `exp(-dt / tau_ear)` which is not only identical for all neurons but also relatively costly to evaluate. Brian will identify such constant expressions, and calculate them only once outside of the loop:

```
c = exp(-dt / tau_ear)
for each neuron:
    x_new = sound + c * (sound - x_old)
```

In addition to this type of optimisation, the Brian simulator will also simplify arithmetic expressions, such as replacing `0 * x` by `0`, or `x / x` by `1`. While all these optimizations could in principle also be performed by the programming-language compiler (e.g. gcc), we have found that performing these changes before handing over the code to the compiler led to bigger and more reliable performance benefits.

Code execution: runtime mode

After the code generation process, each model component has been transformed into one or more "code objects", each performing a specific computational task. For example, a group of integrate-and-fire neurons would typically result in three code objects. The first would be responsible for integrating the state variables over a single timestep, the second for checking the threshold condition to determine which neurons emit an action potential, and the third for applying the reset statements to those neurons. By default, these code objects will be executed in Brian's "runtime mode", meaning that the simulation loop will be executed in Python and then call each of the code objects to perform the actual computation (in the order defined by the scheduling as described in the previous section). Note that while the code objects will typically be based on generated C++ code, they can be compiled and executed from within Python using binding libraries such as *weave* (formerly part of *scipy* ; Jones et al. 2001–) or *Cython* (Behnel et al., 2011).

This "mixed" approach to model execution leaves the simulation control to the main Python process while the actual computations are performed in compiled code, operating on shared memory structures. This results in a considerable amount of flexibility: whenever Brian's model description formalism is not expressive enough for a task at hand, the researcher can interleave the execution of generated code with a hand-written function that can potentially access and modify any aspect of the model. In particular, such a function could intervene in the simulation process itself, e.g. by interrupting the simulation if certain criteria are met. The jupyter notebook at `https://github.com/brian-team/brian2_paper_examples` contains an the interactive version of the eye movement example (***Figure 3***). In this example, the aforementioned mechanism is used to allow the user to interactively control a running Brian simulation, as well as for providing a graphical representation of the results that updates continuously.

While having all these advantages, the back-and-forth between the main loop in Python and the code objects also entails a performance overhead. This performance overhead takes a constant amount of time per code object and time step and does therefore matter less if the individual components perform long-running computations, such as for large networks (Brette & Goodman, 2011, see also ***Figure 8***). On the other hand, for simulations of small or medium-sized networks, such as the network presented in ***Figure 6***, this overhead can be considerable and the alternative execution mode presented in the following section might provide a better alternative.

Code execution: standalone mode

As an alternative to the mode of execution presented in the previous section, the Brian simulator offers the so-called "standalone mode", currently implemented for the C++ programming language. In this mode, Brian generates code that performs the simulation loop itself and executes the operations according to the schedule. Additionally, it creates code to manages the memory for all state variables and other data structures such as the queuing mechanism used for applying synaptic effects with delays. This code, along with the code of the individual code objects, establishes a complete "standalone" version of the simulation run. When the resulting binary file is executed, it will perform the simulation and write all the results to disk. Since the generated code does not depend on any non-standard libraries, it can be easily transferred to other machines or architectures (e.g. for robotics applications). The generated code is free from any overhead related to Python or complex data structures and therefore executes with high performance.

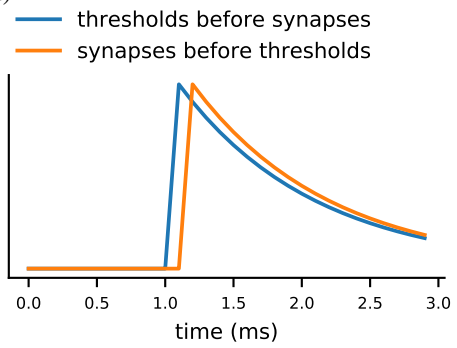
For many models, the use of this mode only requires the researcher to add a single line to the simulation script (declaring `set_device('cpp_standalone')`), all aspects of the model descriptions, including assignments to state variables and the order of operations will be faithfully conserved in the generated code. The Python script will transparently compile and execute the standalone code, and then read the results back from disk so that the researcher does not have to adapt their analysis routines.

However, in contrast to the runtime execution mode presented earlier, it is not possible to interact with the simulation during its execution from within the Python script. In addition, certain programming logic is no longer possible, since all actions such as synapse generation or variable assignments are not executed when they are stated, but only as part of the simulation run.

In this execution mode, simulations of moderate size and complexity can be run in real-time (***Figure 8***), enabling studies such as the one presented in ***Figure 6***. Importantly, this mode does not require the researcher to be actively involved in any details of the compilation, execution of the simulation or the retrieval of the results.

B Simulation scheduling

(a)



(b)

```python
from brian2 import *

tau = 1*ms
spikes = SpikeGeneratorGroup(1, [0], [1]*ms)
target = NeuronGroup(1, 'dv/dt = -v/tau : 1')
synapses = Synapses(spikes, target,
                    on_pre='v += 1')
synapses.connect()
mon = StateMonitor(target, 'v', record=True)
```

(c)

```python
# thresholds before synapses (default)
magic_network.schedule = ['start',
                          'groups',
                          'thresholds',
                          'synapses',
                          'resets',
                          'end']
run(3*ms)
```

```python
# synapses before thresholds
magic_network.schedule = ['start',
                          'groups',
                          'synapses',
                          'thresholds',
                          'resets',
                          'end']
run(3*ms)
```

(d)

object	part of	Clock dt	when	order	active
mon (*StateMonitor*)	mon (*StateMonitor*)	100. us (every step)	start	0	yes
target_stateupdater (*StateUpdater*)	target (*NeuronGroup*)	100. us (every step)	groups	0	yes
spikes (*SpikeGeneratorGroup*)	spikes (*SpikeGeneratorGroup*)	100. us (every step)	thresholds	0	yes
synapses_pre (*SynapticPathway*)	synapses (*Synapses*)	100. us (every step)	synapses	-1	yes

Figure 9. Demonstration of the effect of scheduling simulation elements. **(a)** Timing of synaptic effects on the post-synaptic cell for the two simulation schedules defined in (c). **(b)** Basic simulation code for the simulation results shown in (a). **(c)** Definition of a simulation schedule where threshold crossings trigger spikes and – assuming the absence of synaptic delays – their effect is applied directly within the same simulation time step (left; see blue line in (a)), and a schedule where synaptic effects are applied in the time step following a threshold crossing (right; see orange line in (a)). **(d)** Summary of the scheduling of the simulation elements following the default schedule (left code in (c)), as provided by Brian's `scheduling_summary` function. Note that for increased readibility, the objects from (b) have been explicitly named to match the variable names. Without this change, the code in (b) leads to the use of standard names for the objects (`spikegeneratorgroup`, `neurongroup`, `synapses`, and `statemonitor`).

```
from brian2 import *

defaultclock.dt = 0.01*ms
E_L = -68*mV; E_Na = 20*mV; E_K = -80*mV; E_Ca = 120*mV; E_proc = -10*mV
C_s = 0.2*nF; C_a = 0.02*nF; g_E = 10*nS; g_La = 7.5*nS; g_Na = 300*nS;
g_Kd = 4*uS; G_Ca = 0.2*uS; G_K = 16*uS; tau_h_Ca = 150*ms; tau_m_A = 0.1*ms;
tau_h_A = 50*ms; tau_m_proc = 6*ms; tau_m_Na = 0.025*ms; tau_z = 5*second

eqs = '''# somatic compartment
dV_s/dt = (-I_syn - I_L - I_Ca - I_K - I_A - I_proc - g_E*(V_s - V_a))/C_s : volt
I_L = g_Ls*(V_s - E_L) : amp
I_K = g_K*m_K**4*(V_s - E_K) : amp
I_A = g_A*m_A**3*h_A*(V_s - E_K) : amp
I_proc = g_proc*m_proc*(V_s - E_proc) : amp
I_syn = I_fast + I_slow: amp
I_fast : amp
I_slow : amp
I_Ca = g_Ca*m_Ca**3*h_Ca*(V_s - E_Ca)            : amp
dm_Ca/dt = (m_Ca_inf - m_Ca)/tau_m_Ca            : 1
m_Ca_inf = 1/(1 + exp(0.205/mV*(-61.2*mV - V_s))): 1
tau_m_Ca = 30*ms -5*ms/(1 + exp(0.2/mV*(-65*mV - V_s))) : second
dh_Ca/dt = (h_Ca_inf - h_Ca)/tau_h_Ca            : 1
h_Ca_inf = 1/(1 + exp(-0.15/mV*(-75*mV - V_s)))  : 1
dm_K/dt = (m_K_inf - m_K)/tau_m_K                : 1
m_K_inf = 1/(1 + exp(0.1/mV*(-35*mV - V_s)))     : 1
tau_m_K = 2*ms + 55*ms/(1 + exp(-0.125/mV*(-54*mV - V_s))) : second
dm_A/dt = (m_A_inf - m_A)/tau_m_A                : 1
m_A_inf = 1/(1 + exp(0.2/mV*(-60*mV - V_s)))     : 1
dh_A/dt = (h_A_inf - h_A)/tau_h_A                : 1
h_A_inf = 1/(1 + exp(-0.18/mV*(-68*mV - V_s)))   : 1
dm_proc/dt = (m_proc_inf - m_proc)/tau_m_proc    : 1
m_proc_inf = 1/(1 + exp(0.2/mV*(-55*mV - V_s)))  : 1
# axonal compartment
dV_a/dt = (-g_La*(V_a - E_L) - g_Na*m_Na**3*h_Na*(V_a - E_Na)
           -g_Kd*m_Kd**4*(V_a - E_K) - g_E*(V_a - V_s))/C_a : volt
dm_Na/dt = (m_Na_inf - m_Na)/tau_m_Na            : 1
m_Na_inf = 1/(1 + exp(0.1/mV*(-42.5*mV - V_a)))  : 1
dh_Na/dt = (h_Na_inf - h_Na)/tau_h_Na            : 1
h_Na_inf = 1/(1 + exp(-0.13/mV*(-50*mV - V_a)))  : 1
tau_h_Na = 10*ms/(1 + exp(0.12/mV*(-77*mV - V_a))) : second
dm_Kd/dt = (m_Kd_inf - m_Kd)/tau_m_Kd            : 1
m_Kd_inf = 1/(1 + exp(0.2/mV*(-41*mV - V_a)))    : 1
tau_m_Kd = 12.2*ms + 10.5*ms/(1 + exp(-0.05/mV*(58*mV - V_a))) : second
# class-specific fixed maximal conductances
g_Ls   : siemens (constant)
g_A    : siemens (constant)
g_proc : siemens (constant)
# Adaptive conductances
g_Ca = G_Ca/2*(1 + tanh(z)) : siemens
g_K = G_K/2*(1 - tanh(z))   : siemens
I_diff = (I_target + I_Ca) : amp
dz/dt = tanh(I_diff/nA)/tau_z : 1
I_target : amp (constant)
# Neuron class
label : integer (constant)'''
circuit = NeuronGroup(3, eqs, method='rk2',
                      threshold='m_Na > 0.5', refractory='m_Na > 0.5')
ABPD, LP, PY = 0, 1, 2
# class-specific constants
circuit.label = [ABPD, LP, PY]
circuit.I_target = [0.4, 0.3, 0.5]*nA; circuit.g_Ls = [30, 25, 15]*nS
circuit.g_A = [450, 100, 250]*nS; circuit.g_proc = [6, 8, 0]*nS
# Initial conditions
circuit.V_s = E_L; circuit.V_a = E_L
circuit.m_Ca = 'm_Ca_inf'; circuit.h_Ca = 'h_Ca_inf'; circuit.m_K = 'm_K_inf';
circuit.m_A = 'm_A_inf'; circuit.h_A = 'h_A_inf'; circuit.m_proc = 'm_proc_inf'
circuit.m_Na = 'm_Na_inf'; circuit.h_Na = 'h_Na_inf'; circuit.m_Kd = 'm_Kd_inf'
```

Figure 2–Figure supplement 1. Simulation code for the more biologically detailed model of the circuit shown in **Figure 1**a (based on Golowasch et al., 1999). The code for the synaptic model and connections is identical to the code shown in **Figure 2**, except for acting on V_s instead of v in the target cell.