Figures and figure supplements

Brian 2, an intuitive and efficient neural simulator

Marcel Stimberg *et al*

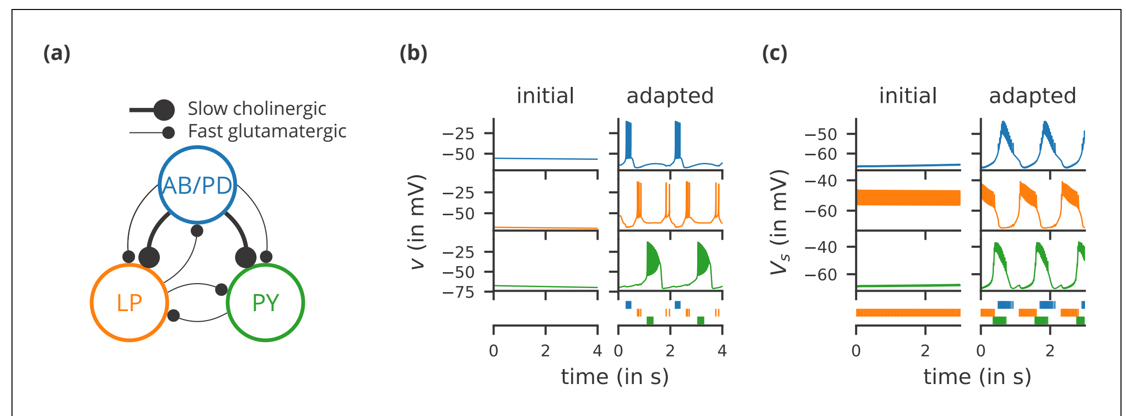


Figure 1. Case study 1: A model of the pyloric network of the crustacean stomatogastric ganglion, inspired by several modelling papers on this subject (***Golowasch et al., 1999***; ***Prinz et al., 2004***; ***Prinz, 2006***; ***O'Leary et al., 2014***). (a) Schematic of the modelled circuit (after ***Prinz et al., 2004***). The pacemaker kernel is modelled 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), using the same colour code. The bottom row shows their spiking activity in a raster plot, with spikes defined as excursions of the membrane potential over −20 mV. In the left column ('initial'), activity is shown for 4 s after an initial settling time of 2.5 s. The right column ('adapted') shows the activity with fully adapted conductances (see text for details) after an additional time of 49 s. (c) Activity of the simulated neurons of a biologically detailed version of the circuit shown in (a), following (***Golowasch et al., 1999***). All conventions as in (b), except for showing 3 s of activity after a settling time of 0.5 s ('initial'), and after an additional time of 24 s ('adapted'). Also note that the biologically detailed model consists of two coupled compartments, but only the membrane potential of the somatic compartment (V_s) is shown here.
DOI: https://doi.org/10.7554/eLife.47314.003

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

DOI: https://doi.org/10.7554/eLife.47314.004

```
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 1a***, producing the circuit activity shown in ***Figure 1c***. Simulation code for the more biologically detailed

Figure 2—figure supplement 1 continued on next page

Figure 2—figure supplement 1 continued

model of the circuit shown in ***Figure 1a*** (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.
DOI: https://doi.org/10.7554/eLife.47314.005

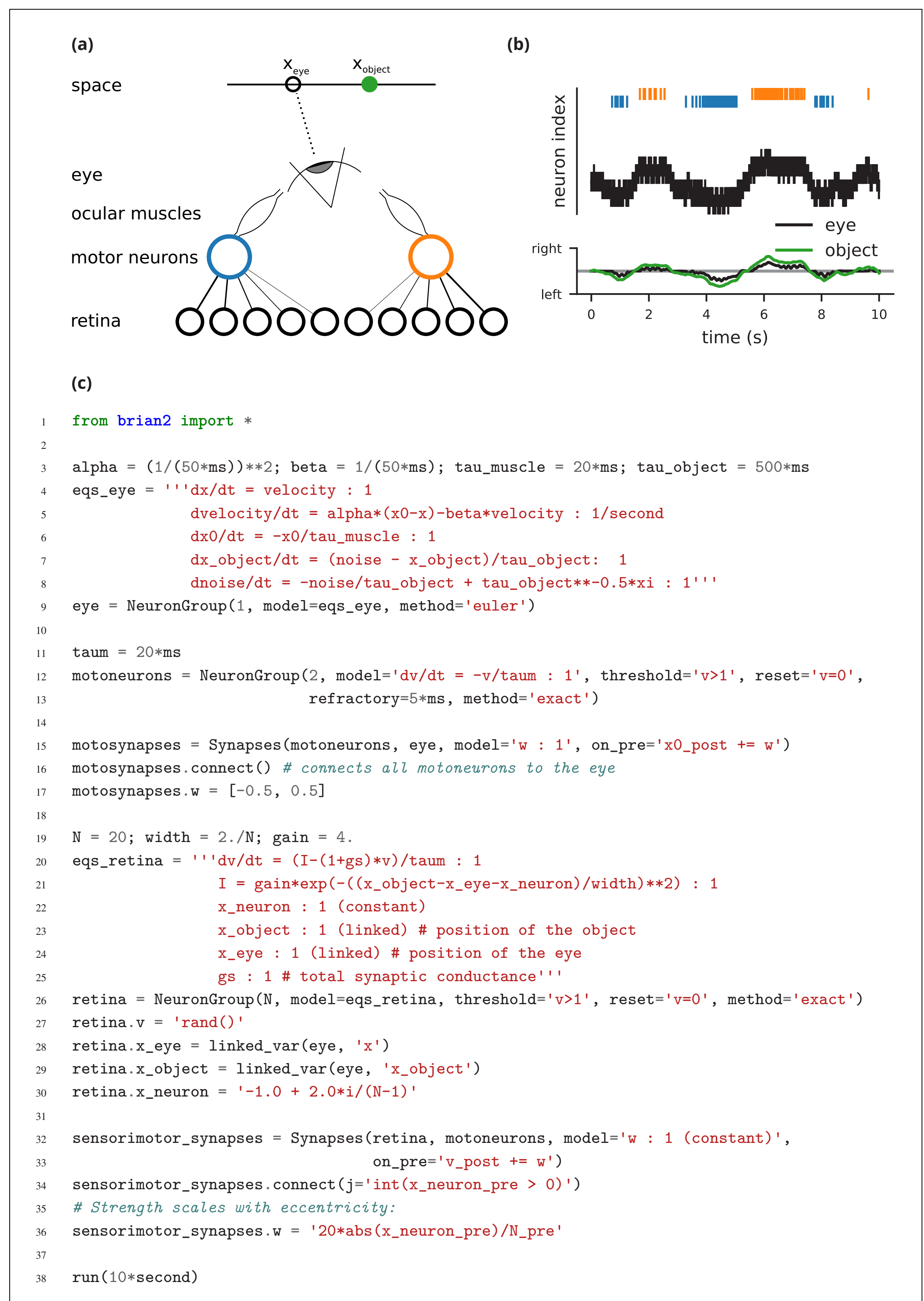


Figure 3. Case study 2: Smooth pursuit eye movements. (a) Schematics of the model. An object (green) moves along a line and activates retinal neurons (bottom row; black) that are sensitive to the relative position of the object to the eye. Retinal neurons activate two motor neurons with weights depending on the eccentricity of their

Figure 3 continued on next page

Figure 3 continued

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.

DOI: https://doi.org/10.7554/eLife.47314.006

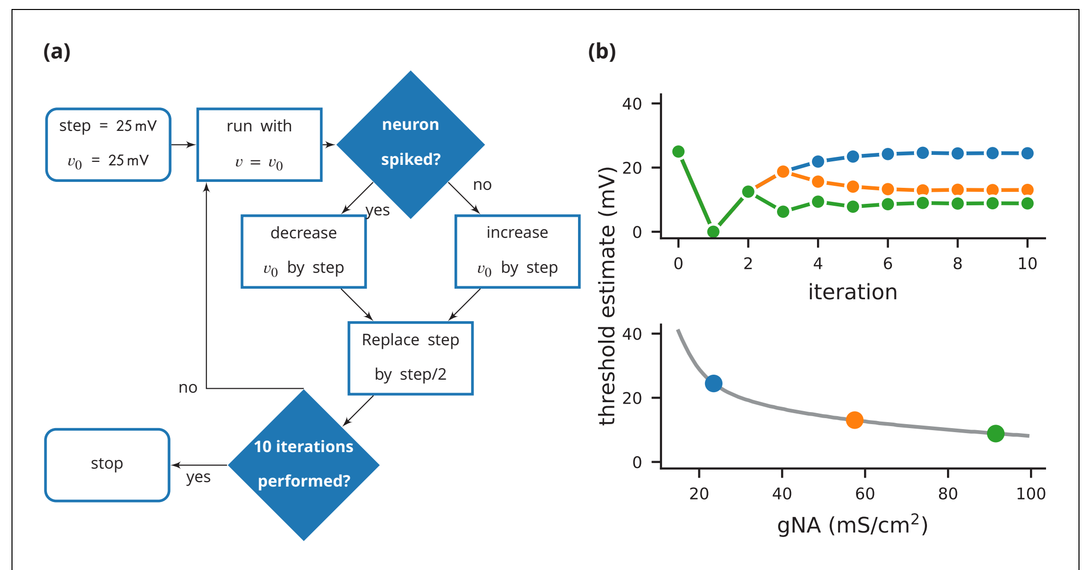


Figure 4. Case study 3: Using bisection to find a neuron's voltage threshold. (a) Schematic of the bisection algorithm for finding a neuron's voltage threshold. The algorithm is applied in parallel for different values of sodium density. (b) Top: Refinement of the voltage threshold estimate over iterations 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.

DOI: https://doi.org/10.7554/eLife.47314.007

```
1  from brian2 import *
2  defaultclock.dt = 0.01*ms
3
4  El = 10.613*mV; ENa = 115*mV; EK = -12*mV
5  gl = 0.3*mS/cm**2; gK = 36*mS/cm**2; C = 1*uF/cm**2
6  gNa_min = 15*mS/cm**2; gNa_max = 100*mS/cm**2
7
8  eqs = '''dv/dt = (gl*(El - v) + gNa*m**3*h*(ENa - v) + gK*n**4*(EK - v)) / C : volt
9           gNa : siemens/meter**2
10          dm/dt = alpham*(1 - m) - betam*m : 1
11          dn/dt = alphan*(1 - n) - betan*n : 1
12          dh/dt = alphah*(1 - h) - betah*h : 1
13          alpham = (0.1/mV)*(-v + 25*mV)/(exp((-v + 25*mV)/(10*mV)) - 1)/ms : Hz
14          betam = 4 * exp(-v/(18*mV))/ms : Hz
15          alphah = 0.07 * exp(-v/(20*mV))/ms : Hz
16          betah = 1/(exp((-v+30*mV) / (10*mV)) + 1)/ms : Hz
17          alphan = (0.01/mV) * (-v+10*mV) / (exp((-v+10*mV) / (10*mV)) - 1)/ms : Hz
18          betan = 0.125*exp(-v/(80*mV))/ms : Hz'''
19 neurons = NeuronGroup(100, eqs, threshold='v > 50*mV', method='exponential_euler')
20 neurons.gNa = 'gNa_min + (gNa_max - gNa_min)*1.0*i/N'
21 neurons.v = 0*mV
22 neurons.m = '1/(1 + betam/alpham)'
23 neurons.n = '1/(1 + betan/alphan)'
24 neurons.h = '1/(1 + betah/alphah)'
25 S = SpikeMonitor(neurons)
26
27 store()
28
29 # We locate the threshold by bisection
30 v0 = 25*mV*ones(len(neurons))
31 step = 25*mV
32
33 for i in range(10):
34     restore()
35     neurons.v = v0
36     run(20*ms)
37     v0[S.count == 0] += step
38     v0[S.count > 0] -= step
39     step /= 2.0
```

Figure 5. Case study 3: Simulation code to find a neuron's voltage threshold, implementing the bisection algorithm detailed in ***Figure 4a***. The code simulates 100 unconnected neurons with sodium densities between 15 mS cm^{-2} and 100 mS cm^{-2}, following the model of ***Hodgkin and Huxley (1952)***. Results from these simulations are shown in ***Figure 4b***.

DOI: https://doi.org/10.7554/eLife.47314.008

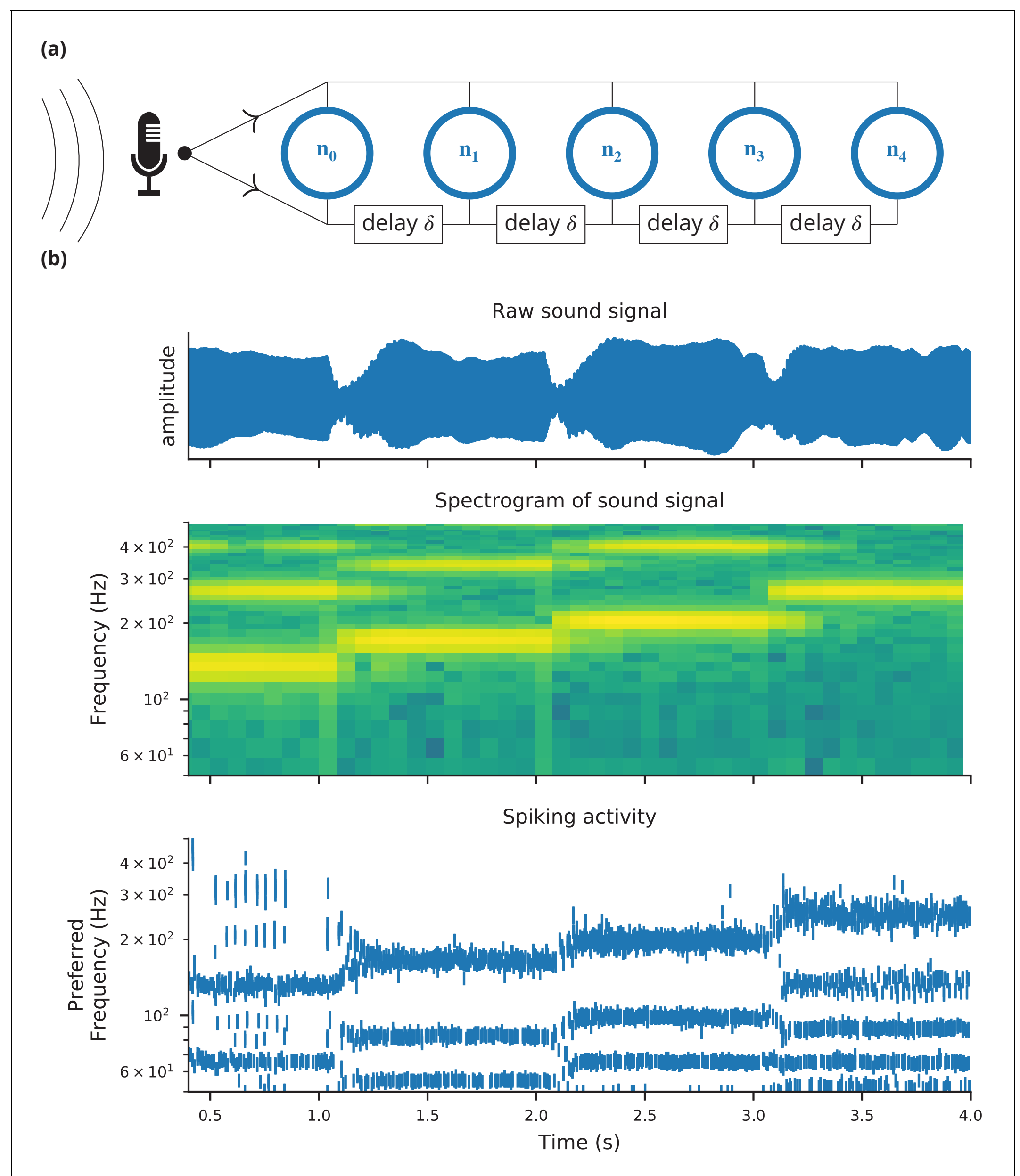


Figure 6. Case study 4: Neural pitch processing with real-time input. (**a**) Model schematic: Audio input is converted into spikes and fed into a population of coincidence-detection neurons via two pathways, one instantaneous, that is without any delay (top), and one with incremental delays (bottom). 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 synthesised 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.

DOI: https://doi.org/10.7554/eLife.47314.009

```
1  from brian2 import *
2  import os
3  set_device('cpp_standalone')
4
5  sample_rate = 48*kHz; buffer_size = 128; defaultclock.dt = 1/sample_rate
6  max_delay = 20*ms; tau_ear = 1*ms; tau_th = 5*ms
7  min_freq = 50*Hz; max_freq = 1000*Hz; num_neurons = 300; tau = 1*ms; sigma = .1
8
9  @implementation('cpp','''
10 PaStream *_init_stream() {
11     PaStream* stream;
12     Pa_Initialize();
13     Pa_OpenDefaultStream(&stream, 1, 0, paFloat32, SAMPLE_RATE, BUFFER_SIZE, NULL, NULL);
14     Pa_StartStream(stream);
15     return stream;
16 }
17
18 float get_sample(const double t) {
19     static PaStream* stream = _init_stream();
20     static float buffer[BUFFER_SIZE];
21     static int next_sample = BUFFER_SIZE;
22
23     if (next_sample >= BUFFER_SIZE)
24     {
25         Pa_ReadStream(stream, buffer, BUFFER_SIZE);
26         next_sample = 0;
27     }
28     return buffer[next_sample++];
29 }''', libraries=['portaudio'], headers=['<portaudio.h>'],
30       define_macros=[('BUFFER_SIZE', buffer_size),
31                      ('SAMPLE_RATE', sample_rate)])
32 @check_units(t=second, result=1)
33 def get_sample(t):
34     raise NotImplementedError('Use a C++-based code generation target.')
35
36 eqs_ear = '''dx/dt = (sound - x)/tau_ear: 1 (unless refractory)
37              dth/dt = (0.1*x - th)/tau_th : 1
38              sound = clip(get_sample(t), 0, inf) : 1 (constant over dt)'''
39 receptors = NeuronGroup(1, eqs_ear, threshold='x>th',
40                         reset='x=0; th = th*2.5 + 0.01',
41                         refractory=2*ms, method='exact')
42 receptors.th = 1
43
44 eqs_neurons = '''dv/dt = -v/tau+sigma*(2./tau)**.5*xi : 1
45                  freq : Hz (constant)'''
46 neurons = NeuronGroup(num_neurons, eqs_neurons, threshold='v>1', reset='v=0', method='euler')
47 neurons.freq = 'exp(log(min_freq/Hz)+(i*1.0/(num_neurons-1))*log(max_freq/min_freq))*Hz'
48
49 synapses = Synapses(receptors, neurons, on_pre='v += 0.5', multisynaptic_index='k')
50 synapses.connect(n=2)  # one synapse without delay; one with delay
51 synapses.delay['k == 1'] = '1/freq_post'
52
53 run(10*second)
```

Figure 7. Case study 4: Simulation code for the model shown in ***Figure 6a***. 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 and Burk, 1999***).

DOI: https://doi.org/10.7554/eLife.47314.010

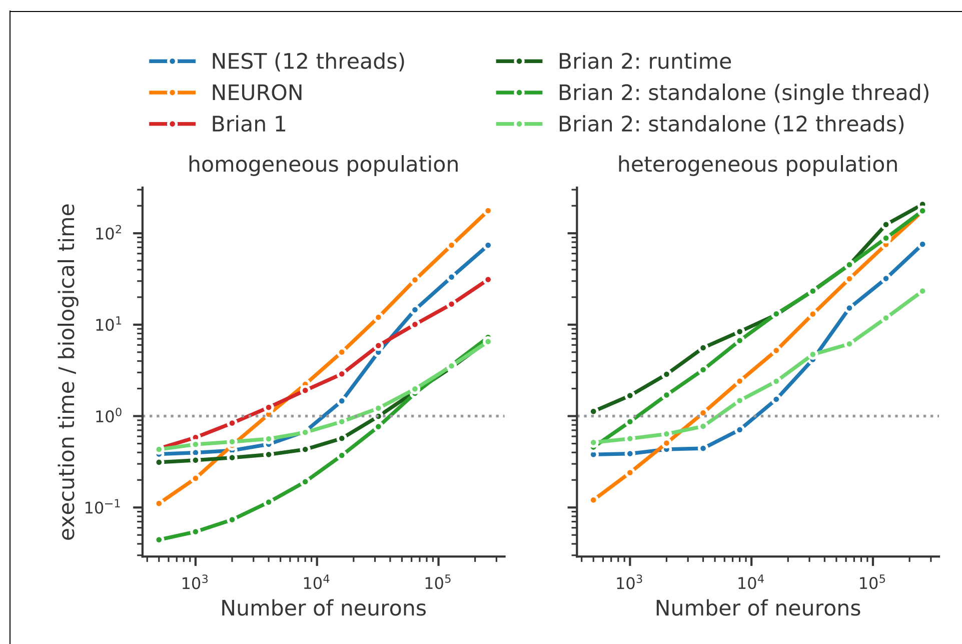
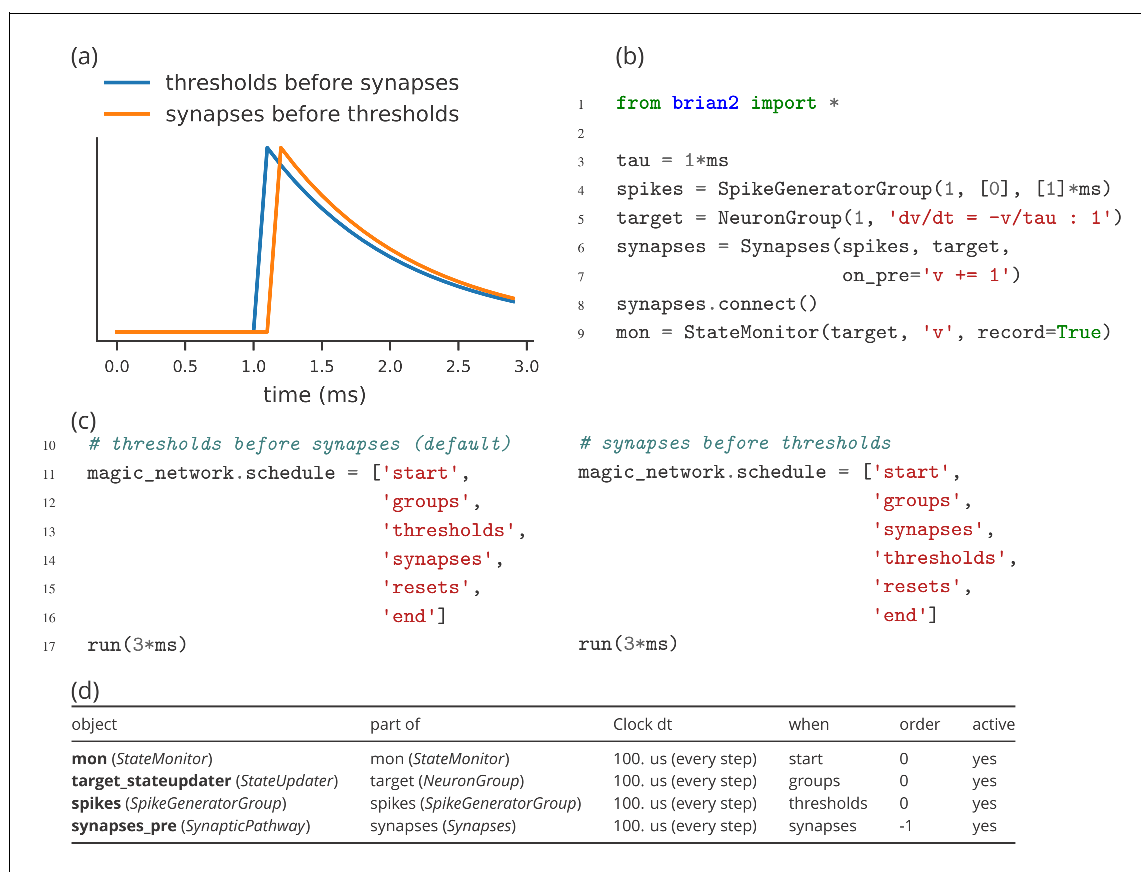
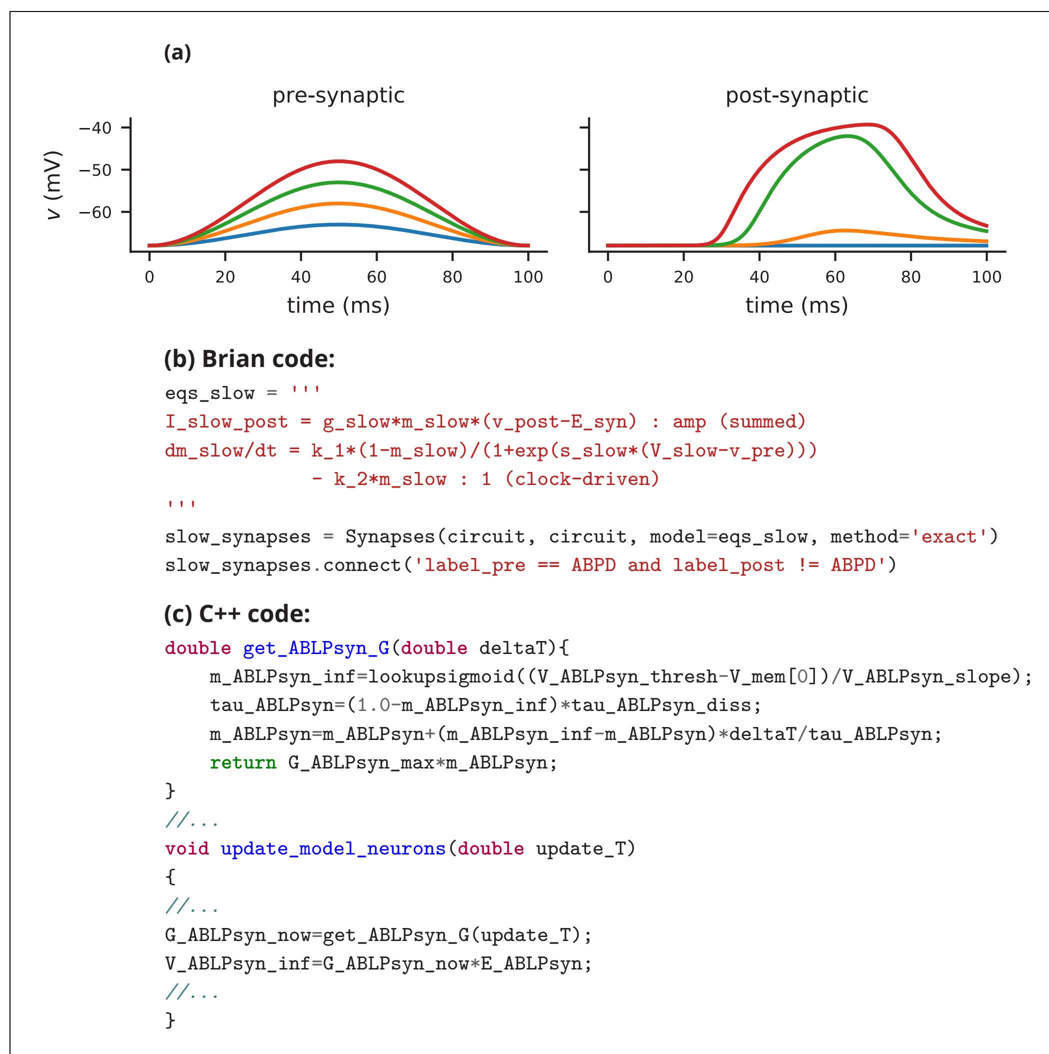


Figure 8. Benchmark of the simulation time for the CUBA network (***Vogels and Abbott, 2005***; ***Brette et al., 2007***), a sparsely connected network of leaky-integrate and fire network with synapses modelled as exponentially decaying currents. Synaptic connections are random, with each neuron receiving on average 80 synaptic inputs and weights set to ensure ongoing asynchronous activity in the network. The simulations use exact integration, but spike spike times are aligned to the simulation grid of 0.1 ms. Simulations are shown for a homogeneous population (left), where the membrane time constant, as well as the excitatory and inhibitory time constant, are the same for all neurons. In the heterogeneous population (right), these constants are different for each neuron, randomly set between 90% and 110% of the constant values used in the homogeneous population. Simulations were performed with NEST 2.16 (blue, ***Linssen et al., 2018***, RRID:SCR_002963), NEURON 7.6 (orange; RRID:SCR_005393), Brian 1.4.4 (red), and Brian 2.2.2.1 (shades of green, ***Stimberg et al., 2019b***, RRID:SCR_002998). Benchmarks were run under Python 2.7.16 on an Intel Core i9-7920X machine with 12 processor cores. For NEST and one of the Brian 2 simulations (light green), simulations made use of all processor cores by using 12 threads via the OpenMP framework. Brian 2 'runtime' simulations execute C++ code via the weave library, while 'standalone' code executes an independent binary file compiled from C++ code (see Appendix 1 for details). Simulation times do not include the one-off times to prepare the simulation and generate synaptic connections as these will become a vanishing fraction of the total time for runs with longer simulated times. Simulations were run for a biological time of 10 s for small networks (8000 neurons or fewer) and for 1 s for large networks. The times plotted here are the best out of three repetitions. Note that Brian 1.4.4 does not support exact integration for a heterogeneous population and has therefore not been included for that benchmark.
DOI: https://doi.org/10.7554/eLife.47314.011



Appendix 2—figure 1. 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`).

DOI: https://doi.org/10.7554/eLife.47314.018



Appendix 3—figure 1. Graded synapse model. (a) Demonstration of the effect of the graded synapse model used in case study 1 (***Figure 1***, ***Figure 2***). On the left, the membrane potential excursion of a pre-synaptic neuron is modelled by a squared sinusoidal function of time with varying amplitudes from 5 mV to 20 mV. The plot on the right shows the post-synaptic membrane potential of a cell receiving graded synaptic input from the pre-synaptic cell via the graded synapse model from case study 1 (slow cholinergic synapse, cf. ***Golowasch et al., 1999***). The post-synaptic cell is modelled here as a simple leaky integrator with a single synaptic input current. (b) Code excerpt showing the Brian 2 definition of the graded synapse model used in (a), taken from the code used in case study 1 (***Figure 2***). (c) Code excerpt defining a graded synapse model in C++ as part of 'The Pyloric Network Model Simulator' (http://www.biology.emory.edu/research/Prinz/database-sensors/; ***Günay and Prinz, 2010***). The complete code is 3510 lines.

DOI: https://doi.org/10.7554/eLife.47314.020

(a) NeuroML2:

```
<gradedSynapse id="gs" conductance="5pS" delta="5mV" Vth="-55mV"
 k="0.025per_ms" erev="0mV"/>
```

LEMS:

```
<ComponentType name="gradedSynapse" extends="baseGradedSynapse">
    <Property name="weight" dimension="none" defaultValue="1"/>
    <Parameter name="conductance" dimension="conductance"/>
    <Parameter name="delta" dimension="voltage">
    <Parameter name="k" dimension="per_time">
    <Parameter name="Vth" dimension="voltage">
    <Parameter name="erev" dimension="voltage">
    <Exposure name="i" dimension="current"/>
    <Exposure name="inf" dimension="none"/>
    <Exposure name="tau" dimension="time"/>
    <Requirement name="v" dimension="voltage"/>
    <InstanceRequirement name="peer" type="baseGradedSynapse"/>
    <Dynamics>
        <StateVariable name="s" dimension="none"/>
        <DerivedVariable name="vpeer" dimension="voltage" select="peer/v"/>
        <DerivedVariable name="inf" dimension="none"
                         value="1/(1 + exp((Vth - vpeer)/delta))" exposure="inf"/>
        <DerivedVariable name="tau" dimension="time"
                         value="(1-inf)/k" exposure="tau"/>
        <DerivedVariable name="i" exposure="i"
                         value="weight * conductance * s * (erev-v)"/>
        <TimeDerivative variable="s" value="s_rate" />
    </Dynamics>
</ComponentType>
```

(b) NEURON (NMODL):

```
INDEPENDENT {t FROM 0 TO 1 WITH 1 (ms)}
NEURON {
  POINT_PROCESS int1_lgsyn
  POINTER vpre
  RANGE gmax, g, e, i
  NONSPECIFIC_CURRENT i
}
UNITS {
  (nA) = (nanoamp)
  (mV) = (millivolt)
  (umho) = (micromho)
}
PARAMETER {
  gmax=0   (umho)
  e=0      (mV)
  v        (mV)
}
STATE { synon synoff}
ASSIGNED {
  i (nA)
  g (umho)
  vpre (mV)
}
BREAKPOINT {
  SOLVE syn METHOD sparse
  g = gmax *synon
  i = g*(v - e)
}
KINETIC syn {
  ~ synoff <-> synon (syninf(vpre)/tausyn(vpre),
                  (1-syninf(vpre))/tausyn(vpre))
}
INITIAL {
  synon = 0.0
  synoff = 1.0
}
FUNCTION syninf(v){
  syninf = 1/(1+exp(-0.5*(v+49)))
}
FUNCTION tausyn(v){
  tausyn = 2+98/(1+exp(-0.5*(v+49)))
}
```

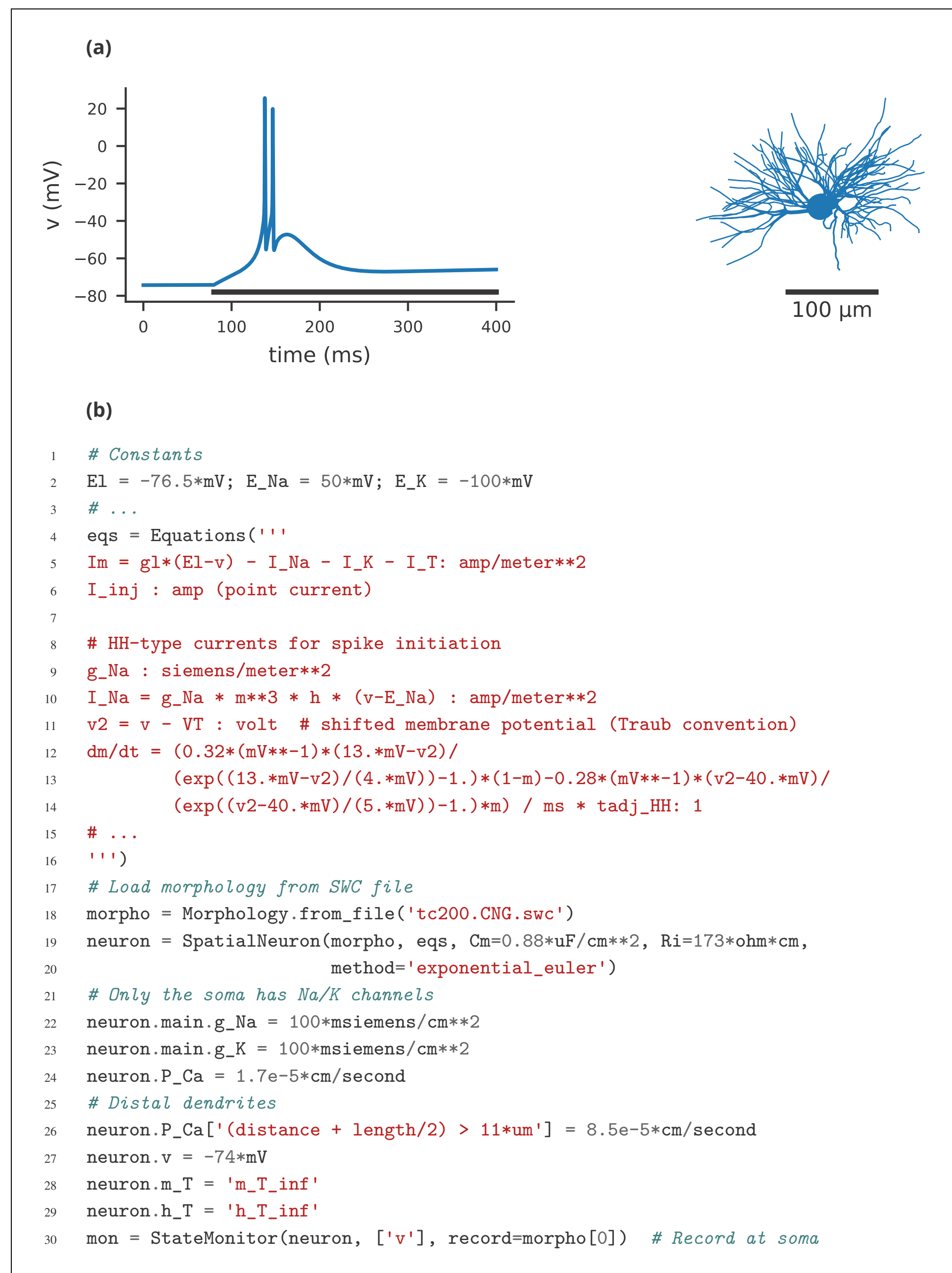
Appendix 3—figure 2. Graded synapse model (cont.). (a) Definition of a graded synapse model in NeuroML2/LEMS. The graded synapse model as described in ***Prinz et al. (2004)*** has been added as a "core type" to the (not yet finalized) NeuroML2 standard and can therefore be accessed under the name gradedSynapse (top). It is fully defined via the LEMS definition partially reproduced here. (b) A definition of a graded synapse in the

Appendix 3—figure 2 continued on next page

Appendix 3—figure 2 continued

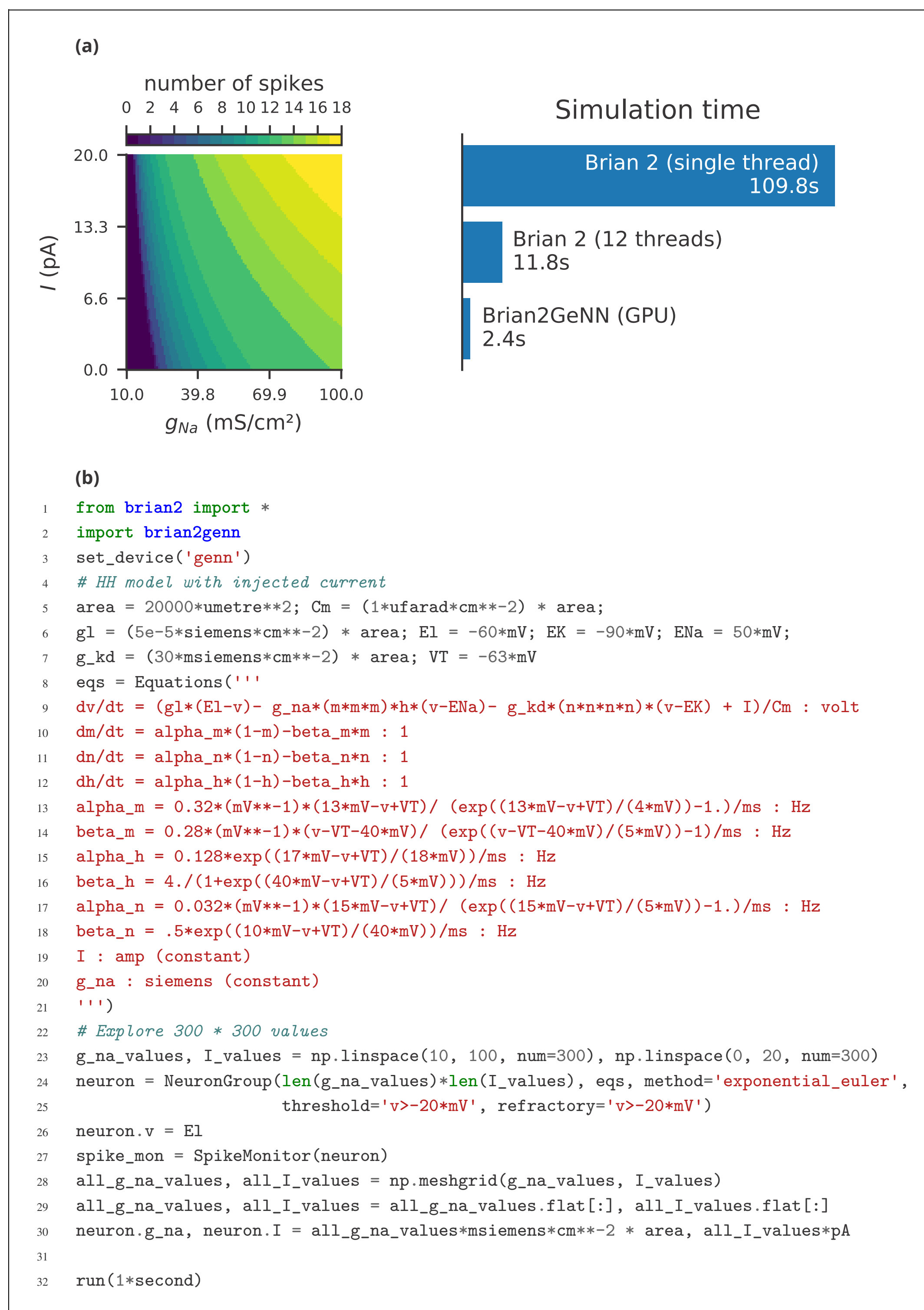
stomatogastric system, written in the NMODL language for the NEURON simulator. This model has been implemented in ***Nadim et al. (1998)***, see https://senselab.med.yale.edu/ModelDB/showmodel.cshtml?model=3511.

DOI: https://doi.org/10.7554/eLife.47314.021



Appendix 4—figure 1. A multi-compartment model of a thalamic relay cell. (a) Simulation of a thamalic relay cell with increased T-current in distal dendrites (partially reproducing Figure 9C from ***Destexhe et al. (1998)***. The plot shows the somatic membrane potential for a current injection of 75 pA during the period marked by the black line. The model consists of a total of 1291 compartments and is based on the morphology available on NeuroMorpho.Org (***Ascoli et al., 2007***) under the ID NMO_00881, displayed on the right. This morphology is a reconstruction of a cell in the rat's ventrobasal complex, originally described in ***Huguenard and Prince (1992)***. (b) Selected lines from the simulation code implementing the model shown in (a), focussing on the differences to single-compartmental models (as shown in case studies 1–4).

DOI: https://doi.org/10.7554/eLife.47314.023



Appendix 4—figure 2. Parameter exploration over two parameters. (a) In a single-compartment neuron model of the Hodgkin-Huxley type (following ***Traub and Miles, 1991***), we record the number of spikes over 1 s while varying the strength of a constant input current I, and the density of the sodium conductance g_{Na} for a total of 300 × 300 values. The bars on the right show the simulation time on the same machine used for ***Figure 8*** when using Brian 2's C++ standalone mode with a single thread (top), with 12 threads (middle), or when simulating it on a NVIDIA GeForce RTX 2080 Ti graphics card via the Brian2GeNN (***Stimberg et al., 2018***) interface to the GeNN

Appendix 4—figure 2 continued on next page

Appendix 4—figure 2 continued

(***Yavuz et al., 2016***) simulator (bottom). (**b**) Code for the simulation shown in (**a**), here configured to run on the GPU via the Brian2GeNN interface (l.2-3).

DOI: https://doi.org/10.7554/eLife.47314.024