

Table of Contents

0. Introduction	1.1
1. Neuron models	1.2
1.1 Biological background	1.2.1
1.2 Biophysical models	1.2.2
1.3 Reduced models	1.2.3
1.4 Firing rate models	1.2.4
2. Synapse models	1.3
2.1 Synaptic models	1.3.1
2.2 Plasticity models	1.3.2
3. Network models	1.4
3.1 E-I balanced network	1.4.1
3.2 Decision making networks	1.4.2
3.3 Continuous attractor neural network	1.4.3
Appendix: model code	1.5
Neuron models	1.5.1
Synapse models	1.5.2
Network models	1.5.3

BrainPy Introduction

In this chapter, we will briefly introduce how to implement computational neuroscience models with BrainPy. For more detailed documents and tutorials, please check our Github repository [BrainPy](#) and [BrainModels](#).

BrainPy is a Python platform for computational neuroscience and brain-inspired computation. To model with BrainPy, users should follow 3 steps:

- 1) Define Python classes for neuron and synapse models. BrainPy provides base classes for different kinds of models, users only need to inherit from those base classes, and define specific methods to tell BrainPy what operations they want the models to take during the simulation. In this process, BrainPy will assist users in the numerical integration of differential equations (ODE, SDE, etc.), adaptation of various backends (`Numpy`, `PyTorch`, etc.), and other functions to simplify code logic.
- 2) Instantiate Python classes as objects of neuron groups and synapse connection groups, pass the instantiated objects to BrainPy class `Network`, and call method `run` to simulate the network.
- 3) Call BrainPy modules like the `measure` module and the `visualize` module to display the simulation results.

With this overall concept of BrainPy, we will go into more detail about implementations in the following sections. In neural systems, neurons are connected by synapses to build networks, so we will introduce [1. neuron models](#), [2. synapse models](#), and [3. network models](#) in order.

1. Neuron models

Neuron models can be classified into three types from complex to simple: biophysical models, reduced models and firing rate models.

1.1 Biological Background

1.2 Biophysical models

1.3 Reduced models

1.4 Firing rate models

1.1 Biological backgrounds

As the basic unit of neural systems, neurons maintain mystique to researchers for a long while. In recent centuries, however, along with the development of experimental techniques, researchers have painted a general figure of those little things working ceaselessly in our neural system.

To achieve our final goal of modeling neurons with computational neuroscience methods, we may start with a patch of real neuron membrane.

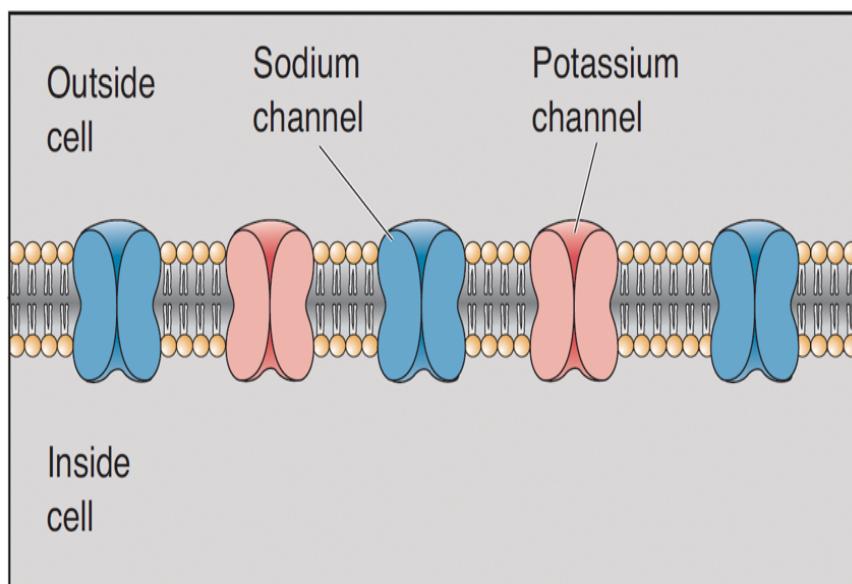


Fig. 1-1 Neuron membrane diagram (Bear et al., 2015¹)

The figure above is a general diagram of neuron membrane with phospholipid bilayer and ion channels. The membrane divides the ions and fluid into intracellular and extracellular, partially prevent them from exchanging, thus generates **membrane potential**--- the difference in electric potential across the membrane.

An ion in the fluid is subjected to two forces. The force of diffusion is caused by the ion concentration difference across the membrane, while the force of electric field is caused by the electric potential difference. When these two forces reach balance, the total forces on ions are 0, and each type of ion meets an equilibrium potential, while the neuron holds a membrane potential lower than 0.

This membrane potential integrated by all those ion equilibrium potentials is the **resting potential**, and the neuron is, in a so-called **resting state**. If the neuron is not disturbed, it will just come to the balanced resting state, and rest.

1.1 Biological background

However, our neural system receives countless inputs every millisecond, from external inputs to recurrent inputs, from specific stimulus inputs to non-specific background inputs. Receiving all these inputs, neurons generate **action potentials** (or **spikes**) to transfer and process information all across the neural system.

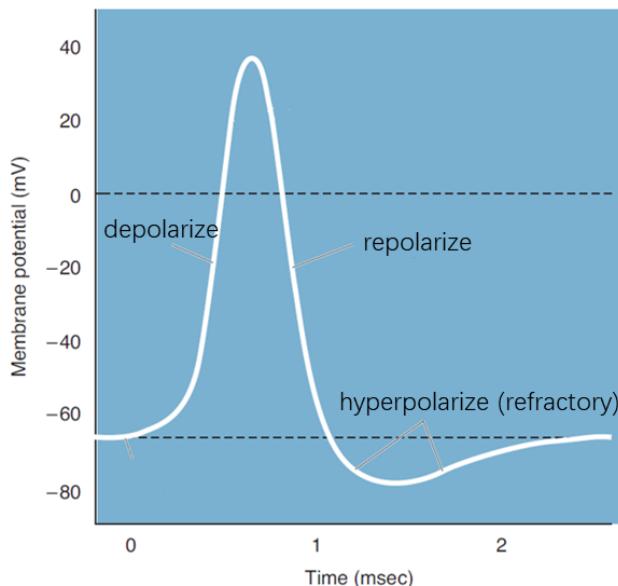


Fig. 1-2 Action Potential (Adapted from Bear et al., 2015¹)

Passing through the ion channels shown in Fig.1-1, ions on both sides of the hydrophobic phospholipid bilayer are exchanged. Due to changes in the environment caused by, for example, an external input, ion channels will switch between their open/close states, therefore allow/prohibit ion exchanges. During the switch, the ion concentrations (mainly Na^+ and K^+) change, induce a significant change on neuron's membrane potential: the membrane potential will raise to a peak value and then fall back in a short time period. Biologically, when such a series of potential changes happens, we say the neuron generates an **action potential** or a **spike**, or the neuron fires.

An action potential can be mainly divided into three periods: **depolarization**, **repolarization** and **refractory period**. During the depolarization period, Na^+ flow into the neuron and K^+ flow out of the neuron, however the inflow of Na^+ is faster, so the membrane potential raises from a low value V_{rest} to a value much higher called V_{th} , then the outflow of K^+ becomes faster than Na^+ , and the membrane potential is reset to a value lower than resting potential during the repolarization period. After that, because of the relatively lower membrane potential, the neuron is unlikely to generate another spike immediately, until the refractory period passes.

1.1 Biological background

A single action potential is complex enough, but in our neural system, one single neuron can generate several action potentials in less than a second. How, exactly, do the neurons fire? Different kinds of neurons may spike when facing different inputs, and the pattern of their spiking can be classified into several firing patterns, some of which are shown in the following figure.

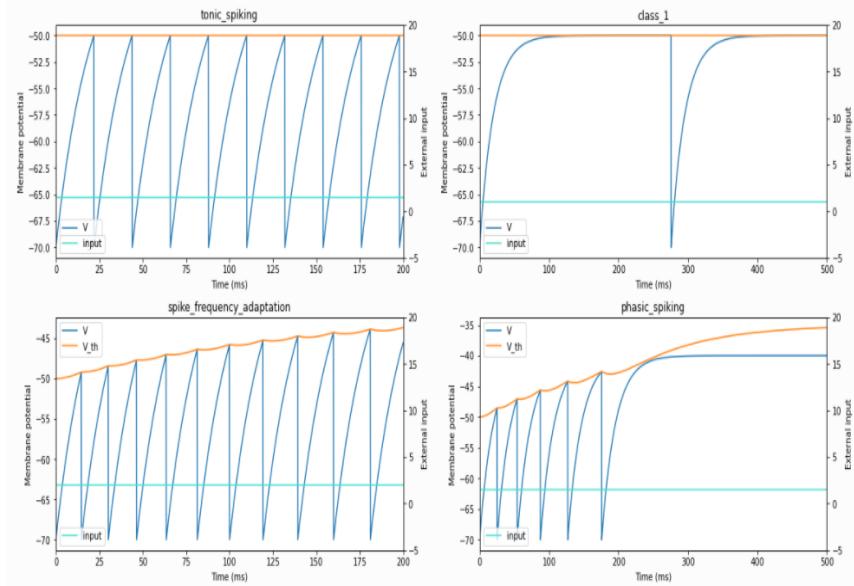


Fig. 1-3 Some firing patterns

Those firing patterns, together with the shape of action potentials, are what computational neuroscience wants to model at the cellular level.

References

- [1] Bear, Mark, Barry Connors, and Michael A. Paradiso. *Neuroscience: Exploring the brain, Fourth Edition*. Jones & Bartlett Learning, LLC, 2015.

1.2 Biophysical models

1.2.1 Hodgkin-Huxley model

Hodgkin and Huxley (1952) recorded the generation of action potential on squid giant axons with voltage clamp technique, and proposed the canonical neuron model called **Hodgkin-Huxley model (HH model)**.

In last section we have introduced a general template for neuron membrane. Computational neuroscientists always model neuron membrane as equivalent circuit like the following figure.

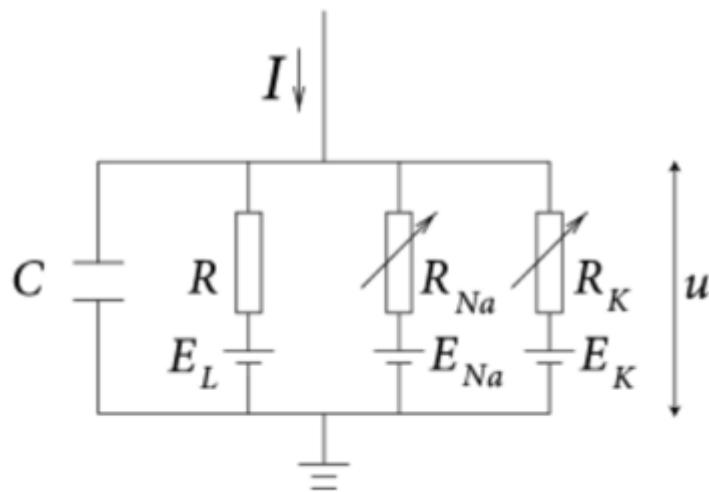


Fig. 1-4 Equivalent circuit diagram (Gerstner et al., 2014¹)

The equivalent circuit diagram of Fig.1-1 is shown in Fig. 1-4, in which the patch of neuron membrane is converted into electric components. In Fig.1-4, the capacitance C refers to the hydrophobic phospholipid bilayer with low conductance, and current I refers to the external stimulus.

As Na^+ ion channels and K^+ ion channels are important in the generation of action potentials, these two ion channels are modeled as the two variable resistances R_{Na} and R_K in parallel on the right side of the circuit diagram, and the resistance R refers to all the non-specific ion channels on the membrane. The batteries E_{Na} , E_K and E_L refer to the electric potential differences caused by the concentration differences of corresponding ions.

Consider the Kirchhoff's first law, that is, for any node in an electrical circuit, the sum of currents flowing into that node is equal to the sum of currents flowing out of that node, Fig. 1-4 can be modeled as differential equations:

1.1 Biological background

$$C \frac{dV}{dt} = -(\bar{g}_{Na} m^3 h (V - E_{Na}) + \bar{g}_K n^4 (V - E_K) + g_{leak} (V - E_{leak})) + I(t)$$

$$\frac{dx}{dt} = \alpha_x (1 - x) - \beta_x, x \in \{Na, K, leak\}$$

That is the HH model. Note that in the first equation above, the first three terms on the right hand are the current go through Na+ ion channels, K+ ion channels and other non-specific ion channels, respectively, while $I(t)$ is an external input. On the left hand, $C \frac{dV}{dt} = \frac{dQ}{dt} = I$ is the current go through the capacitance.

In the computing of ion channel currents, other than the Ohm's law $I = U/R = gU$, HH model introduces three **gating variables** m, n and h to control the open/close state of ion channels. To be precise, variables m and h control the state of Na+ ion channel, variable n controls the state of K+ ion channel, and the real conductance of an ion channel is the product of maximal conductance \bar{g} and the state of gating variables. Gating variables' dynamics can be expressed in a Markov-like form, in which α_x refers to the activation rate of gating variable x, and β_x refers to the de-activation rate of x. The expressions of α_x and β_x (as shown in equations below) are fitted by experimental data.

$$\alpha_m(V) = \frac{0.1(V + 40)}{1 - \exp(-\frac{(V+40)}{10})}$$

$$\beta_m(V) = 4.0 \exp(-\frac{(V + 65)}{18})$$

$$\alpha_h(V) = 0.07 \exp(-\frac{(V + 65)}{20})$$

$$\beta_h(V) = \frac{1}{1 + \exp(-\frac{(V+35)}{10})}$$

$$\alpha_n(V) = \frac{0.01(V + 55)}{1 - \exp(-\frac{(V+55)}{10})}$$

$$\beta_n(V) = 0.125 \exp(-\frac{(V + 65)}{80})$$

1.1 Biological background

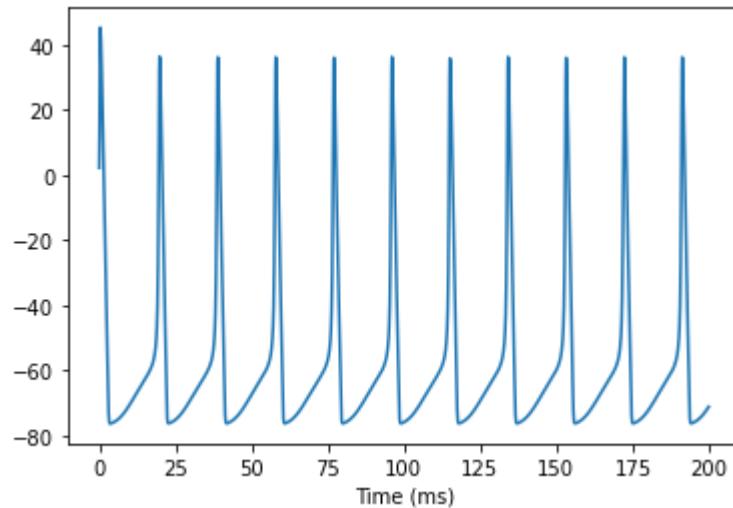
```

1   class HH(bp.NeuGroup):           bp.NeuGroup class:
2       target_backend = 'general'    Group of neurons
3
4       @staticmethod               Set backend for model.
5           @bp.odeint(method='exponential_euler')      Call `bp.odeint` to integrate ODEs.
6           def integral(V, m, h, n, t, C, gNa, ENa, gK, EK, gL, EL, Iext): Set parameter 'method' to choose
7               alpha_m = 0.1*(V+40)/(1-bp.ops.exp(-(V+40)/10)) αm(V) = (0.1(V + 40))/(1 - exp(-(V + 40)/10))
8               beta_m = 4.0*bp.ops.exp(-(V+65)/18) βm(V) = 4.0exp(-(V + 65)/18)
9               dmdt = alpha_m * (1 - m) - beta_m * m dx/dt = α(1 - x) - βx, x = m
10
11              alpha_h = 0.07*bp.ops.exp(-(V+65)/20) αh(V) = 0.07 exp(-(V + 65)/20)
12              beta_h = 1/(1+bp.ops.exp(-(V+35)/10)) βh(V) = 1/(1 + exp(-(V + 35)/10))
13              dhdt = alpha_h * (1 - h) - beta_h * h dx/dt = α(1 - x) - βx, x = h
14
15              alpha_n = 0.01*(V+55)/(1-bp.ops.exp(-(V+55)/10)) αn(V) = (0.01(V + 55))/(1 - exp(-(V + 55)/10))
16              beta_n = 0.125*bp.ops.exp(-(V+65)/80) βn(V) = 0.125exp(-(V + 65)/80)
17              dndt = alpha_n * (1 - n) - beta_n * n dx/dt = α(1 - x) - βx, x = n
18
19              I_Na = (gNa * m ** 3.0 * h) * (V - ENa)      C dV/dt = -(gNam3h(V - ENa) + gKn4(V - EK) +
20              I_K = (gK * n ** 4.0) * (V - EK)             gleak(V - Eleak)) + I(t)
21              I_leak = gL * (V - EL)                      ]
22              dVdt = (- I_Na - I_K - I_leak + Iext) / C
23
24      return dVdt, dmdt, dhdt, dnkt
25
26  def __init__(self, size, ENa=50., gNa=120., EK=-77., gK=36.,
27              EL=-54.387, gL=0.03, V_th=20., C=1.0, **kwargs):
28      # parameters
29      self.ENa = ENa
30      self.EK = EK
31      self.EL = EL
32      self.gNa = gNa
33      self.gK = gK
34      self.gL = gL
35      self.C = C
36      self.V_th = V_th
37
38      # variables
39      num = bp.size2len(size)
40      self.V = -65. * bp.ops.ones(num)
41      self.m = 0.5 * bp.ops.ones(num)
42      self.h = 0.6 * bp.ops.ones(num)
43      self.n = 0.32 * bp.ops.ones(num)
44      self.spike = bp.ops.zeros(num, dtype=bool)
45      self.input = bp.ops.zeros(num)
46
47      super(HH, self).__init__(size=size, **kwargs) → Pass 'size' and '**kwargs' to
48
49  def update(self, _t):
50      V, m, h, n = self.integral(self.V, self.m, self.h, self.n, _t,      Update variables with
51                           self.C, self.gNa, self.ENa, self.gK,      numerical integration
52                           self.EK, self.gL, self.EL, self.input)      in vector form.
53      self.spike = (self.V < self.V_th) * (V >= self.V_th) → Judge if the neuron spikes.
54      self.V = V
55      self.m = m
56      self.h = h
57      self.n = n
58      self.input[:] = 0 → Reset external input
                                         for this time step.
```

Run codes in our github repository: <https://github.com/PKU-NIP-Lab/BrainModels>

The V-t plot of HH model simulated by BrainPy is shown below. The three periods, depolarization, repolarization and refractory period of a real action potential can be seen in the V-t plot. In addition, during the depolarization period, the membrane integrates external inputs slowly at first, and increases rapidly once it grows beyond some point, which also reproduces the "shape" of action potentials.

1.1 Biological background



References

- [1] Gerstner, Wulfram, et al. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.

1.3 Reduced models

Inspired by biophysical experiments, Hodgkin-Huxley model is precise but costly. Researchers proposed the reduced models to reduce the consumption on computing resources and running time in simulation.

These models are simple and easy to compute, while they can still reproduce the main pattern of neuron behaviors. Although their representation capabilities are not as good as biophysical models, such a loss of accuracy is sometimes acceptable considering their simplicity.

1.3.1 Leaky Integrate-and-Fire model

The most typical reduced model is the **Leaky Integrate-and-Fire model (LIF model)** presented by Lapicque (1907). LIF model is a combination of integrate process represented by differential equation and spike process represented by conditional judgment:

$$\tau \frac{dV}{dt} = -(V - V_{rest}) + RI(t)$$

If $V > V_{th}$, neuron fires,

$$V \leftarrow V_{reset}$$

$\tau = RC$ is the time constant of LIF model, the larger τ is, the slower model dynamics is. The equation shown above is corresponding to a simpler equivalent circuit than HH model, for it does not model the Na^+ and K^+ ion channels any more. Actually, in LIF model, only the resistance R , capacitance C , battery E and external input I is modeled.

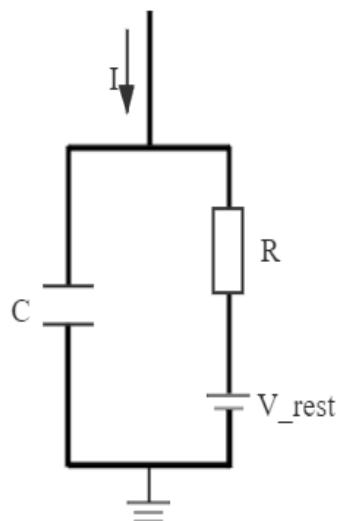


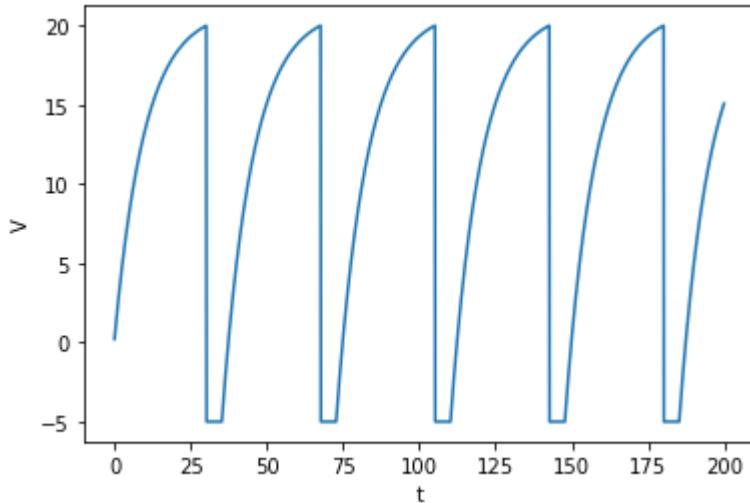
Fig.1-5 Equivalent circuit of LIF model

Compared with HH model, LIF model does not model the shape of action potentials, which means, the membrane potential does not burst before a spike. Also, the refractory period is overlooked in the original model, and in order to generate it, another conditional judgment must be added:

If

$$t - t_{lastspike} \leq refractoryperiod$$

then neuron is in refractory period, membrane potential V will not be updated.



1.3.2 Quadratic Integrate-and-Fire model

To pursue higher representation capability, Latham et al. (2000) proposed **Quadratic Integrate-and-Fire model (QuaIF model)**, in which they add a second order term in differential equation so the neurons can generate spike better.

$$\tau \frac{dV}{dt} = a_0(V - V_{rest})(V - V_c) + RI(t)$$

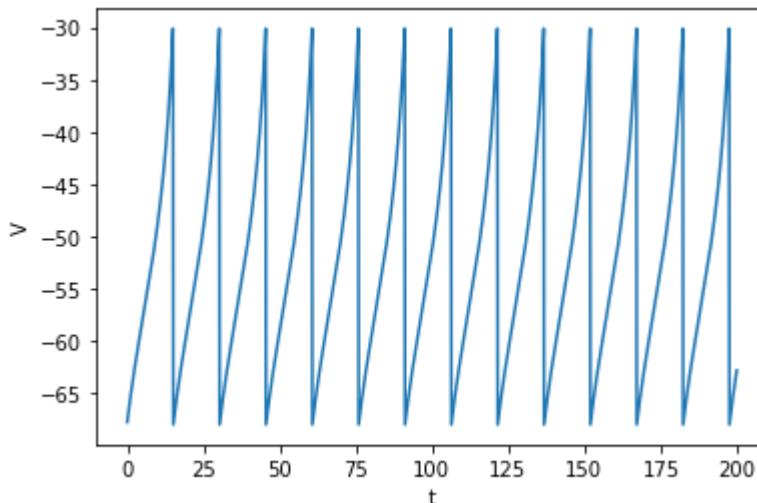
In the equation above, a_0 is a parameter controls the slope of membrane potential before a spike, and V_c is the critical potential for action potential initialization. Below V_C , membrane potential V increases slowly, once it grows beyond V_C , V turns to rapid increase.

1.1 Biological background

```

1   class QuaIF(bp.NeuGroup): → bp.NeuGroup class:
2       target_backend = 'general'
3
4       @staticmethod
5       def derivative(V, t, I_ext, V_rest, V_c, R, tau, a_0):
6           dVdt = (a_0 * (V - V_rest) * (V - V_c) + R * I_ext) / tau → τ  $\frac{dV}{dt} = a_0(V - V_{rest})(V - V_c) + RI(t)$ 
7           return dVdt
8
9       def __init__(self, size, V_rest=-65., V_reset=-68.,
10                  V_th=-30., V_c=-50.0, a_0=0.07,
11                  R=1., tau=10., t_refractory=0., **kwargs):
12           # parameters
13           self.V_rest = V_rest
14           self.V_reset = V_reset
15           self.V_th = V_th
16           self.V_c = V_c
17           self.a_0 = a_0
18           self.R = R
19           self.tau = tau
20           self.t_refractory = t_refractory
21
22           # variables
23           num = bp.size2len(size)
24           self.V = bp.ops.ones(num) * V_reset
25           self.input = bp.ops.zeros(num)
26           self.spike = bp.ops.zeros(num, dtype=bool)
27           self.refractory = bp.ops.zeros(num, dtype=bool)
28           self.t_last_spike = bp.ops.ones(num) * -1e7
29
30           self.integral = bp.odeint(f=self.derivative, method='euler') → Call 'bp.odeint' to integrate ODEs.
31           super(QuaIF, self).__init__(size=size, **kwargs) → Set parameter 'method' to choose
32
33       def update(self, _t):
34           for i in prange(self.size[0]): → For each neuron in neuron group.
35               spike = 0.
36               refractory = (_t - self.t_last_spike[i] <= self.t_refractory) → Check if neuron is
37               if not refractory:                                         in refractory period.
38                   V = self.integral(self.V[i], _t, self.input[i], → Update variables
39                               self.V_rest, self.V_c, self.R, → with numerical integration
40                               self.tau, self.a_0) → one by one.
41                   spike = (V >= self.V_th) → Check if neuron spikes.
42                   if spike:
43                       V = self.V_rest
44                       self.t_last_spike[i] = _t
45                       self.V[i] = V
46                       self.spike[i] = spike
47                       self.refractory[i] = refractory or spike
48                       self.input[i] = 0.

```



1.3.3 Exponential Integrate-and-Fire model

Exponential Integrate-and-Fire model (ExpIF model) (Fourcaud-Trocme et al., 2003) is more expressive than QualIF model. With the exponential term added to the right hand of differential equation, the dynamics of ExpIF model can now generates a more realistic action potential.

$$\tau \frac{dV}{dt} = -(V - V_{rest}) + \Delta_T e^{\frac{V-V_T}{\Delta T}} + RI(t)$$

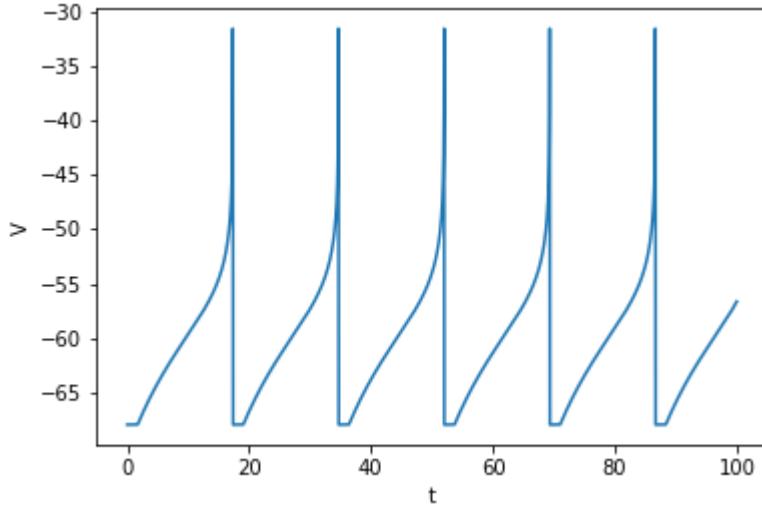
In the exponential term, V_T is the critical potential of generating action potential, below which V increases slowly and above which rapidly. Δ_T is the slope of action potentials in ExpIF model, and when $\Delta_T \rightarrow 0$, the shape of spikes in ExpIF model will be equivalent to the LIF model with $V_{th} = V_T$ (Fourcaud-Trocme et al., 2003).

$V_{th} = V_T$ (Fourcaud-Trocme et al., 2003) .

```

1 class ExpIF(bp.NeuGroup): → bp.NeuGroup class:
2     target_backend = 'general' → Group of neurons
3
4     @staticmethod
5     def derivative(V, t, I_ext, V_rest, delta_T, V_T, R, tau):
6         exp_term = bp.ops.exp((V - V_T) / delta_T)
7         dvdt = (- (V - V_rest) + delta_T * exp_term + R * I_ext) / tau → τ  $\frac{dV}{dt} = -(V - V_{rest}) + \Delta_T e^{\frac{V-V_T}{\Delta T}} + RI(t)$ 
8         return dvdt
9
10    def __init__(self, size, V_rest=65., V_reset=-68.,
11                 V_th=-30., V_T=-59.9, delta_T=3.48,
12                 R=10., C=1., tau=10., t_refractory=1.7,
13                 **kwargs):
14        # parameters
15        self.V_rest = V_rest
16        self.V_reset = V_reset
17        self.V_th = V_th
18        self.V_T = V_T
19        self.delta_T = delta_T
20        self.R = R
21        self.C = C
22        self.tau = tau
23        self.t_refractory = t_refractory → Model parameters saved as floating point numbers.
24
25        # variables
26        self.V = bp.ops.ones(size) * V_rest
27        self.input = bp.ops.zeros(size)
28        self.spike = bp.ops.zeros(size, dtype=bool)
29        self.refractory = bp.ops.zeros(size, dtype=bool)
30        self.t_last_spike = bp.ops.ones(size) * -1e7 → Model variables saved as vectors of floating point numbers.
31
32        self.integral = bp.odeint(self.derivative) → Call 'bp.odeint' to integrate ODEs.
33        super(ExpIF, self).__init__(size=size, **kwargs) → Parameter 'method' is set to default value 'euler'.
34
35    def update(self, _t):
36        for i in prange(self.num): → For each neuron in neuron group.
37            spike = 0.
38            refractory = (_t - self.t_last_spike[i] <= self.t_refractory) → Check if neuron is in refractory period.
39            if not refractory:
40                V = self.integral( → Update variables
41                    self.V[i], _t, self.input[i], self.V_rest, → with numerical integration
42                    self.delta_T, self.V_T, self.R, self.tau → one by one.
43                )
44            spike = (V >= self.V_th) → Check if neuron spikes.
45            if spike:
46                V = self.V_reset
47                self.t_last_spike[i] = _t → Reset neuron.
48                self.V[i] = V
49                self.spike[i] = spike
50                self.refractory[i] = refractory or spike
51                self.input[:] = 0.

```



1.3.4 Adaptive Exponential Integrate-and-Fire model

While facing a constant stimulus, the response generated by a single neuron will sometimes decrease over time, this phenomenon is called **adaptation** in biology.

To reproduce the adaptation behavior of neurons, researchers add a weight variable w to existing integrate-and-fire models like LIF, QualF and ExpIF models. Here we introduce a typical one: **Adaptive Exponential Integrate-and-Fire model (AdExIF model)** (Gerstner et al., 2014).

$$\tau_m \frac{dV}{dt} = -(V - V_{rest}) + \Delta_T e^{\frac{V-V_T}{\Delta_T}} - R w + RI(t)$$

$$\tau_w \frac{dw}{dt} = a(V - V_{rest}) - w + b \tau_w \sum \delta(t - t^f)$$

The first differential equation of AdExIF model, as the model's name shows, is similar to ExpIF model we introduced above, except for the term of adaptation, which is shown as $-Rw$ in the equation.

The weight term w is regulated by the second differential equation. a describes the sensitivity of the recovery variable w to the sub-threshold fluctuations of V , and b is the increment value of w generated by a spike, and w will also decay over time.

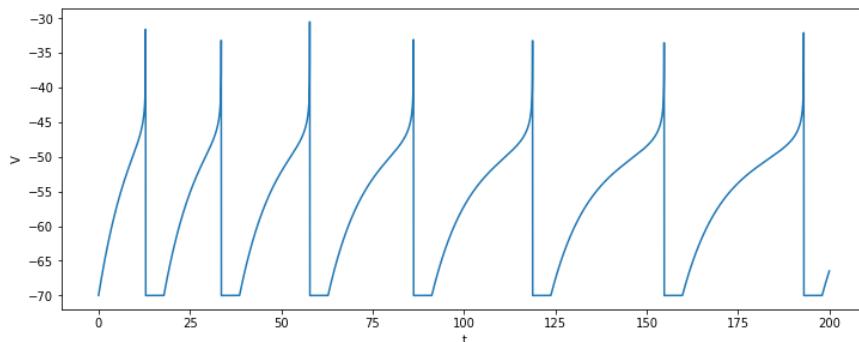
Give AdExIF neuron a constant input, after several spikes, the value of w will increase to a high value, which slows down the rising speed of V , thus reduces the neuron's firing rate.

1.1 Biological background

```

1 class AdExIF(bp.NeuGroup): _____ → bp.NeuGroup class:
2     target_backend = 'general'           Group of neurons
3
4     @staticmethod
5     def derivative(V, w, t, I_ext, V_rest, delta_T, V_T, R, tau, tau_w, a):
6         exp_term = bp.ops.exp((V-V_T)/delta_T) _____ →  $\tau \frac{dV}{dt} = -(V - V_{rest}) + \Delta_T e^{\frac{V-V_T}{\Delta_T}} + RI(t)$ 
7         dVdt = (- (V - V_rest) + delta_T * exp_term - R * w + R * I_ext) / tau
8
9         dwdt = (a * (V - V_rest) - w) / tau_w _____ →  $\tau_w \frac{dw}{dt} = a(V - V_{rest}) - w + b\tau_w \sum \delta(t - t^f)$ 
10
11     return dVdt, dwdt
12
13 def __init__(self, size, V_rest=-65., V_reset=-68.,
14             V_th=-30., V_T=-59.9, delta_T=3.48,
15             a=1., b=1., R=10., tau=10., tau_w=30.,
16             t_refractory=0., **kwargs):
17
18     # parameters _____ → Model parameters saved as floating point numbers.
19     self.V_rest = V_rest
20     self.V_reset = V_reset
21     self.V_th = V_th
22     self.V_T = V_T
23     self.delta_T = delta_T
24     self.a = a
25     self.b = b
26     self.R = R
27     self.tau = tau
28     self.tau_w = tau_w
29     self.t_refractory = t_refractory
30
31     # variables _____ → Model variables saved as vectors of floating point numbers.
32     num = bp.size2len(size)
33     self.V = bp.ops.ones(num) * V_reset
34     self.w = bp.ops.zeros(size)
35     self.input = bp.ops.zeros(num)
36     self.spike = bp.ops.zeros(num, dtype=bool)
37     self.refractory = bp.ops.zeros(num, dtype=bool)
38     self.t_last_spike = bp.ops.ones(num) * -1e7
39
40     self.integral = bp.odeint(f=self.derivative, method='euler') → Call 'bp.odeint' to integrate ODEs.
41
42     super(AdExIF, self).__init__(size=size, **kwargs) → Set parameter 'method' to choose numerical integration methods.
43
44     def update(self, _t): _____ → Pass 'size' and '**kwargs' to superclass bp.NeuGroup's constructor.
45
46         for i in prange(self.size[0]): _____ → For each neuron in neuron group.
47
48             spike = 0.
49             refractory = (_t - self.t_last_spike[i] <= self.t_refractory) → Check if neuron is in refractory period.
50             if not refractory:
51                 V, w = self.integral(self.V[i], self.w[i], _t, self.input[i], → Update variables
52                                         self.V_rest, self.delta_T, → with numerical integration
53                                         self.V_T, self.R, self.tau, self.tau_w, self.a) → one by one.
54
55             spike = (V >= self.V_th) → Check if neuron spikes.
56             if spike:
57                 V = self.V_rest
58                 w += self.b
59                 self.t_last_spike[i] = _t
60                 self.V[i] = V
61                 self.w[i] = w
62                 self.spike[i] = spike
63                 self.refractory[i] = refractory or spike
64                 self.input[i] = 0.
65
66             dwdt = (a * (V - V_rest) - w) / tau_w _____ → Reset membrane potential,
67             self.w[i] = w + bt_w ∑ δ(t - t^f) → add the Dirac δ term of w:
68
69             self.w[i] = w + bt_w ∑ δ(t - t^f)
70
71         self.V = self.V
72         self.w = self.w
73         self.t_last_spike = self.t_last_spike
74         self.refractory = self.refractory
75         self.input = self.input

```



1.3.5 Hindmarsh-Rose model

1.1 Biological background

To simulate the bursting spike pattern in neurons (i.e., continuously firing in a short time period), Hindmarsh and Rose (1984) proposed

Hindmarsh-Rose model, import a third model variable z as slow variable to control the bursting of neuron.

$$\frac{dV}{dt} = y - aV^3 + bV^2 - z + I$$

$$\frac{dy}{dt} = c - dV^2 - y$$

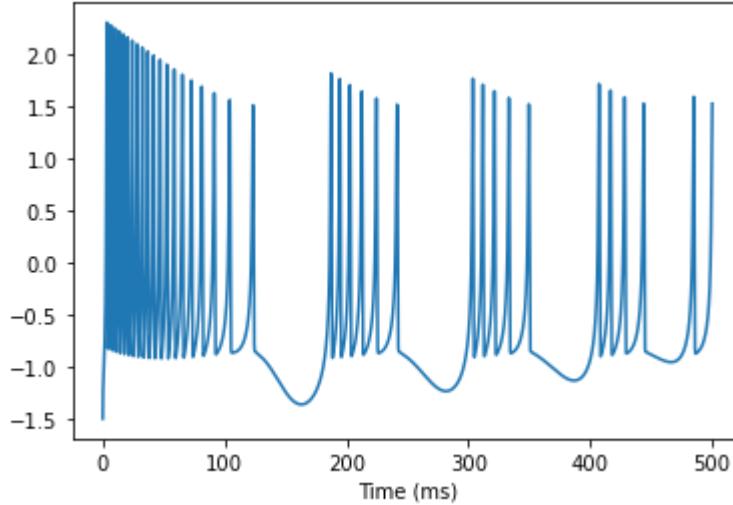
$$\frac{dz}{dt} = r(s(V - V_{rest}) - z)$$

The V variable refers to membrane potential, and y, z are two gating variables. The parameter b in $\frac{dV}{dt}$ equation allows the model to switch between spiking and bursting states, and controls the spiking frequency. r controls slow variable z 's variation speed, affects the number of spikes per burst when bursting, and governs the spiking frequency together with b . The parameter s governs adaptation, and other parameters are fitted by firing patterns.

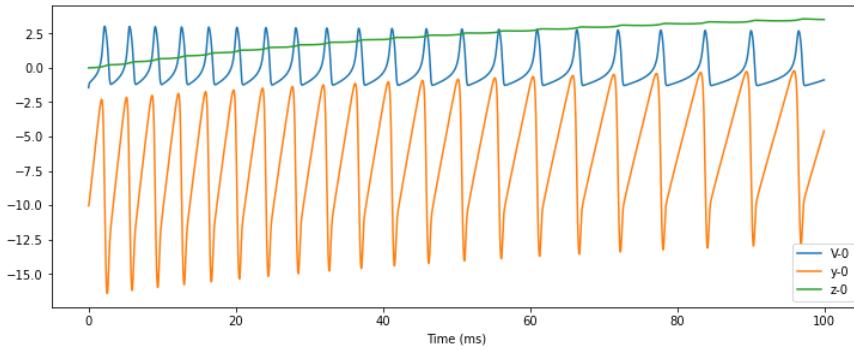
```

1 class HindmarshRose(bp.NeuGroup): → bp.NeuGroup class:
2     target_backend = 'general' → Group of neurons
3
4     @staticmethod
5     def derivative(V, y, z, t, a, b, I_ext, c, d, r, s, V_rest):
6         dVdt = y - a * V * V * V + b * V * V - z + I_ext → dV/dt = y - aV3 + bV2 - z + I
7         dydt = c - d * V * V - y → dy/dt = c - dV2 - y
8         dzdt = r * (s * (V - V_rest) - z) → dz/dt = r(s(V - Vrest) - z)
9         return dVdt, dydt, dzdt
10
11    def __init__(self, size, a=1., b=3.,
12                 c=1., d=5., r=0.01, s=4.,
13                 V_rest=-1.6, **kwargs):
14        # parameters
15        self.a = a
16        self.b = b
17        self.c = c
18        self.d = d
19        self.r = r
20        self.s = s
21        self.V_rest = V_rest
22
23        # variables
24        num = bp.size2len(size)
25        self.z = bp.ops.zeros(num)
26        self.input = bp.ops.zeros(num)
27        self.V = bp.ops.ones(num) * -1.6
28        self.y = bp.ops.ones(num) * -10.
29        self.spike = bp.ops.zeros(num, dtype=bool)
30
31        self.integral = bp.odeint(f=self.derivative) → Call 'bp.odeint' to integrate ODEs.
32        super(HindmarshRose, self).__init__(size=size, **kwargs) → Parameter 'method' is set to
33
34    def update(self, _t): → default value 'euler'.
35        for i in prange(self.num): → Pass 'size' and '**kwargs' to
36            V, self.y[i], self.z[i] = self.integral( → superclass bp.NeuGroup's constructor.
37                self.V[i], self.y[i], self.z[i], _t, → Pass 'size' and '**kwargs' to
38                self.a, self.b, self.input[i], → bp.NeuGroup's constructor.
39                self.c, self.d, self.r, self.s,
40                self.V_rest) → Update variables with
41            self.V[i] = V → numerical integration
42            self.input[i] = 0. → one by one.
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
824
825
826
827
827
828
829
829
830
831
832
833
834
835
836
837
837
838
839
839
840
841
842
843
844
845
846
846
847
848
848
849
849
850
851
852
853
854
855
856
856
857
858
858
859
859
860
861
862
863
864
865
866
866
867
868
868
869
869
870
871
872
873
874
875
876
876
877
878
878
879
879
880
881
882
883
884
885
886
886
887
888
888
889
889
890
891
892
893
894
895
895
896
896
897
897
898
898
899
899
900
901
902
903
904
905
905
906
907
907
908
908
909
909
910
911
912
913
914
914
915
915
916
916
917
917
918
918
919
919
920
921
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
931
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
1581
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1590
1591
1591
1592
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1600
1601
1601
1602
1602
1603
1603
1604
1604
1605
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1610
1611
1611
1612
1612
1613
1613
1614
1614
1615
1615
1616
1616
1617
1617
1618
1618
1619
1619
1620
1620
1621
1621
1622
1622
1623
1623
1624
1624
1625
1625
1626
1626
1627
1627
1628
1628
1629
1629
1630
1630
1631
1631
1632
1632
1633
1633
1634
1634
1635
1635
1636
1636
1637
1637
1638
1638
1639
1639
1640
1640
1641
1641
1642
1642
1643
1643
1644
1644
1645
1645
1646
1646
1647
1647
1648
1648
1649
1649
1650
1650
1651
1651
1652
1652
1653
1653
1654
1654
1655
1655
1656
1656
1657
1657
1658
1658
1659
1659
1660
1660
1661
1661
1662
1662
1663
1663
1664
1664
1665
1665
1666
1666
1667
1667
1668
1668
1669
1669
1670
1670
1671
1671
1672
1672
1673
1673
1674
1674
1675
1675
1676
1676
1677
1677
1678
1678
1679
1679
1680
1680
1681
1681
1682
1682
1683
1683
1684
1684
1685
1685
1686
1686
1687
1687
1688
1688
1689
1689
1690
1690
1691
1691
1692
1692
1693
1693
1694
1694
1695
1695
1696
1696
1697
1697
1698
1698
1699
1699
1700
1700
1701
1701
1702
1702
1703
1703
1704
1704
1705
1705
1706
1706
1707
1707
1708
1708
1709
1709
1710
1710
1711
1711
1712
1712
1713
1713
1714
1714
1715
1715
1716
1716
1717
1717
1718
1718
1719
1719
1720
1720
1721
1721
1722
1722
1723
1723
1724
1724
1725
1725
1726
1726
1727
1727
1728
1728
1729
1729
1730
1730
1731
1731
1732
1732
1733
1733
1734
1734
1735
1735
1736
1736
17
```

1.1 Biological background



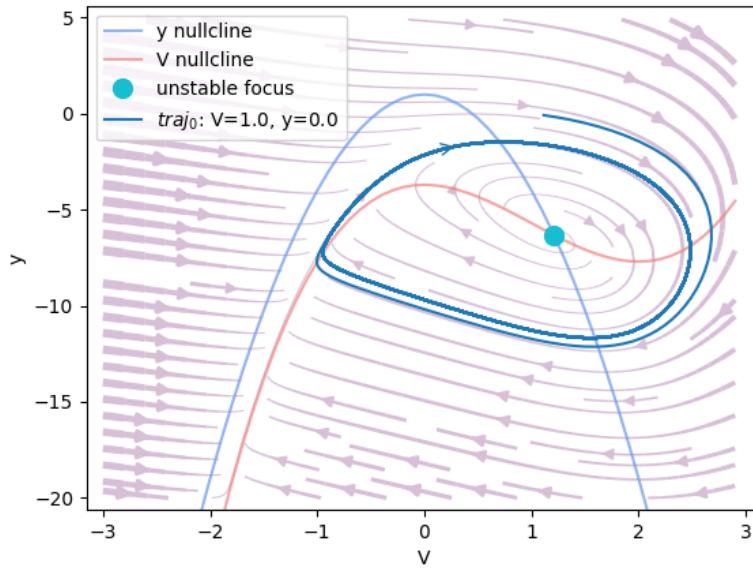
In the variable-t plot painted below, we may see that the slow variable z changes much slower than V and y . Also, V and y are changing periodically during the simulation.



With the theoretical analysis module `analysis` of BrainPy, we may explain the existence of this periodicity through theoretical analysis. In Hindmarsh-Rose model, the trajectory of V and y approaches a limit cycle in phase plane, therefore their values change periodically along the limit cycle.

```

68 # Phase plane analysis
69 phase_plane_analyzer = bp.analysis.PhasePlane(
70     neu.integral,                                ━━━━━━ Dynamic system to be analyzed.
71     target_vars={'V': [-3., 3.], 'y': [-20., 5.]}, ━━━━━━ Variables to be showed in phase plane.
72     fixed_vars={'z': 0.},                           ━━━━━━ Variables to be fixed in phase plane.
73     pars_update={'I_ext': param[mode][1], 'a': 1., 'b': 3.,           ━━━━━━ Other parameters to be fixed.
74         'c': 1., 'd': 5., 'r': 0.01, 's': 4.,
75         'V_rest': -1.6}
76 )
77 phase_plane_analyzer.plot_nullcline()          ━━━━━━ Plot nullcline.
78 phase_plane_analyzer.plot_fixed_point()        ━━━━━━ Plot fixed points.
79 phase_plane_analyzer.plot_vector_field()       ━━━━━━ Plot vector field.
80 phase_plane_analyzer.plot_trajectory(          ━━━━━━ Plot trajectory.
81     [{"V": 1., "y": 0., "z": -0.}],           Define start point of trajectory and
82     duration=100.,                            simulation duration.
83     show=True
84 )
```



1.3.6 Generalized Integrate-and-Fire model

Generalized Integrate-and-Fire model (GIF model) (Mihalaş et al., 2009) integrates several firing patterns in one model. With 4 model variables, it can generate more than 20 types of firing patterns, and is able to alternate between patterns by fitting parameters.

$$\frac{dI_j}{dt} = -k_j I_j, j = 1, 2$$

$$\tau \frac{dV}{dt} = -(V - V_{rest}) + R \sum_j I_j + RI$$

$$\frac{dV_{th}}{dt} = a(V - V_{rest}) - b(V_{th} - V_{th\infty})$$

When V meets V_{th} , Generalized IF neuron fire:

$$I_j \leftarrow R_j I_j + A_j$$

$$V \leftarrow V_{reset}$$

$$V_{th} \leftarrow \max(V_{threset}, V_{th})$$

In the $\frac{dV}{dt}$ differential equation, just like all the integrate-and-fire models, τ is time constant, V is membrane potential, V_{rest} is resting potential, R is conductance, and I is external input.

1.1 Biological background

However, in GIF model, variable amounts of internal currents are added to the equation, shown as the $\sum_j I_j$ term. Each I_j is an internal current in the neuron, with a decay rate of k_j . R_j and A_j are free parameters, R_j describes the dependence of I_j reset value on the value of I_j before spike, and A_j is a constant value added to the reset value after spike.

The variable threshold potential V_{th} is regulated by two parameters: a describes the dependence of V_{th} on the membrane potential V , and b is the rate V_{th} approaches the infinite value of threshold $V_{th\infty}$. $V_{th_{reset}}$ is the reset value of threshold potential when neuron fires.

```

1  class GeneralizedIF(bp.NeuGroup): → bp.NeuGroup class:
2      target_backend = 'general' → Group of neurons
3
4      @staticmethod
5      def derivative(I1, I2, V_th, V, t,
6                      k1, k2, a, V_rest, b, V_th_inf,
7                      R, I_ext, tau):
8          dI1dt = - k1 * I1 → dI1/dt = -k1I1
9          dI2dt = - k2 * I2 → dI2/dt = -k2I2
10         dVthdt = a * (V - V_rest) - b * (V_th - V_th_inf) → dVth/dt = a(V - Vrest) - b(Vth - Vth∞)
11         dVdt = (- (V - V_rest) + R * I_ext + R * I1 + R * I2) / tau → dV/dt = -(V - Vrest) + RΣIj + RI(t),
12         return dI1dt, dI2dt, dVthdt, dVdt → j = 1,2
13
14     def __init__(self, size, V_rest=-70., V_reset=-70.,
15                  V_th_inf=-50., V_th_reset=-60., R=20., tau=20.,
16                  a=0., b=0.01, k1=0.2, k2=0.02,
17                  R1=0., R2=1., A1=0., A2=0.,
18                  **kwargs):
19         # params
20         self.V_rest = V_rest
21         self.V_reset = V_reset
22         self.V_th_inf = V_th_inf
23         self.V_th_reset = V_th_reset
24         self.R = R
25         self.tau = tau
26         self.a = a
27         self.b = b
28         self.k1 = k1
29         self.k2 = k2
30         self.R1 = R1
31         self.R2 = R2
32         self.A1 = A1
33         self.A2 = A2
34
35         # vars
36         self.input = bp.ops.zeros(size)
37         self.spike = bp.ops.zeros(size, dtype=bool)
38         self.I1 = bp.ops.zeros(size)
39         self.I2 = bp.ops.zeros(size)
40         self.V = bp.ops.ones(size) * -70.
41         self.V_th = bp.ops.ones(size) * -50.
42
43         self.integral = bp.odeint(self.derivative) → Call 'bp.odeint' to integrate ODEs.
44         super(GeneralizedIF, self).__init__(size=size, **kwargs) → Parameter 'method' is set to
45                                         default value 'euler'. → Pass 'size' and '**kwargs' to
                                         superclass bp.NeuGroup's constructor.

```

Model parameters saved as floating point numbers.

Model variables saved as vectors of floating point numbers.

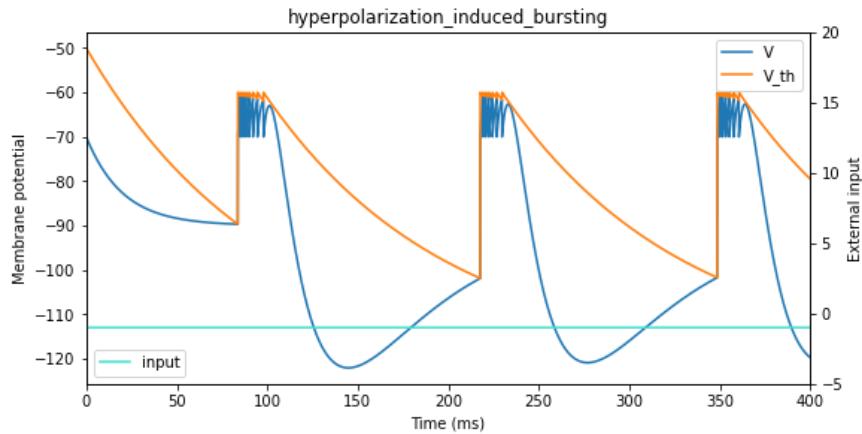
→ Pass 'size' and '**kwargs' to superclass bp.NeuGroup's constructor.

1.1 Biological background

```

46     def update(self, _t):
47         for i in prange(self.size[0]):           For each neuron in neuron group.
48             I1, I2, V_th, V = self.integral(
49                 self.I1[i], self.I2[i], self.V_th[i], self.V[i], _t,
50                 self.k1, self.k2, self.a, self.V_rest,
51                 self.b, self.V_th_inf,
52                 self.R, self.input[i], self.tau
53             )
54             self.spike[i] = self.V_th[i] < V           Check if neuron spikes.
55             if self.spike[i]:
56                 V = self.V_reset
57                 I1 = self.R1 * I1 + self.A1
58                 I2 = self.R2 * I2 + self.A2
59                 V_th = max(V_th, self.V_th_reset)
60                 self.I1[i] = I1
61                 self.I2[i] = I2
62                 self.V_th[i] = V_th
63                 self.V[i] = V
64                 self.f = 0.
65             self.input[:] = self.f           Reset external input
                                            for this time step.

```



1.4 Firing Rate models

Firing Rate models are simpler than reduced models. Typically, firing rate models are used in network models, and they are regarded as compute units, with each of them representing a neuron group. The state of a computation unit is not described by the membrane potential variable V but by the firing rate variable a (or r or ν). Here we introduce a firing rate unit in the classical Wilson and Cowan model.

1.4.1 Firing Rate Unit

Wilson and Cowan (1972) proposed this unit to represent the activities in excitatory and inhibitory cortical neuron columns. Each element of variables a_e and a_i refers to the average activity of a neuron column group contains multiple neurons.

$$\begin{aligned}\tau_e \frac{da_e(t)}{dt} &= -a_e(t) + (k_e - r_e * a_e(t)) * \mathcal{S}(c_1 a_e(t) - c_2 a_i(t) + I_{ext_e}(t)) \\ \tau_i \frac{da_i(t)}{dt} &= -a_i(t) + (k_i - r_i * a_i(t)) * \mathcal{S}(c_3 a_e(t) - c_4 a_i(t) + I_{ext_i}(t)) \\ \mathcal{S}(x) &= \frac{1}{1 + \exp(-a(x - \theta))} - \frac{1}{1 + \exp(a\theta)}\end{aligned}$$

The subscript $x \in \{e, i\}$ points out whether this parameter or variable corresponds to excitatory or inhibitory neuron group. In the differential equations, τ_x refers to the time constant of neuron columns, parameters k_x and r_x control the refractory periods, a_x and θ_x refer to the slope factors and phase parameters of sigmoid functions, and external inputs I_{ext_x} are given separately to excitatory and inhibitory neuron groups.

```

1  class FiringRateUnit(bp.NeuGroup):
2      target_backend = 'general'
3
4      @staticmethod
5      def derivative(a_e, a_i, t,
6                      k_e, r_e, c1, c2, I_ext_e, slope_e, theta_e, tau_e,
7                      k_i, r_i, c3, c4, I_ext_i, slope_i, theta_i, tau_i):
8          x_ae = c1 * a_e - c2 * a_i + I_ext_e
9          sigmoid_ae_l = 1 / (1 + bp.ops.exp(-slope_e * (x_ae - theta_e)))
10         sigmoid_ae_r = 1 / (1 + bp.ops.exp(slope_e * theta_e))
11         sigmoid_ae = sigmoid_ae_l - sigmoid_ae_r
12         daedt = (-a_e + (k_e - r_e * a_e) * sigmoid_ae) / tau_e
13         daidt = c3 * a_e - c4 * a_i + I_ext_i
14         sigmoid_ai_l = 1 / (1 + bp.ops.exp(-slope_i * (x_ai - theta_i)))
15         sigmoid_ai_r = 1 / (1 + bp.ops.exp(slope_i * theta_i))
16         sigmoid_ai = sigmoid_ai_l - sigmoid_ai_r
17         daidt = (-a_i + (k_i - r_i * a_i) * sigmoid_ai) / tau_i
18
19     return daedt, daidt
20

```

1.1 Biological background

```

21     def __init__(self, size, c1=12., c2=4., c3=13., c4=11.,
22                  k_e=1., k_i=1., tau_e=1., tau_i=1., r_e=1., r_i=1.,
23                  slope_e=1.2, slope_i=1., theta_e=2.8, theta_i=4.,
24                  **kwargs):
25         # params
26         self.c1 = c1
27         self.c2 = c2
28         self.c3 = c3
29         self.c4 = c4
30         self.k_e = k_e
31         self.k_i = k_i
32         self.tau_e = tau_e
33         self.tau_i = tau_i
34         self.r_e = r_e
35         self.r_i = r_i
36         self.slope_e = slope_e
37         self.slope_i = slope_i
38         self.theta_e = theta_e
39         self.theta_i = theta_i
40
41         # vars
42         self.input_e = bp.backend.zeros(size)
43         self.input_i = bp.backend.zeros(size)
44         self.a_e = bp.backend.ones(size) * 0.1
45         self.a_i = bp.backend.ones(size) * 0.05
46
47         self.integral = bp.odeint(self.derivative) → Call 'bp.odeint' to integrate ODEs.
48         super(FiringRateUnit, self).__init__(size=size, **kwargs) → Parameter 'method' is set to
49                                         default value 'euler'.
50
51     def update(self, _t):
52         self.a_e, self.a_i = self.integral( → Pass 'size' and '**kwargs' to
53                                             self.a_e, self.a_i, _t,
54                                             self.k_e, self.r_e, self.c1, self.c2,
55                                             self.input_e, self.slope_e,
56                                             self.theta_e, self.tau_e,
57                                             self.k_i, self.r_i, self.c3, self.c4,
58                                             self.input_i, self.slope_i,
59                                             self.theta_i, self.tau_i) → superclass bp.NeuGroup's constructor.
60
61         self.input_e[:] = 0.   → Reset external input
62         self.input_i[:] = 0.   → for this time step.

```

Model parameters saved as floating point numbers.

Model variables saved as vectors of floating point numbers.

Call 'bp.odeint' to integrate ODEs.
Parameter 'method' is set to
default value 'euler'.

Pass 'size' and '**kwargs' to
superclass bp.NeuGroup's constructor.

Update variables with numerical integration
in vector form.

Reset external input
for this time step.

2. Synapse models

Synapse mediates the communication between neurons, and it is an essential component of the formation of the network. Therefore, we need to model the synapse.

We will first introduce how to implement synaptic dynamics with BrainPy, then introduce synapse plasticity.

2.1 Synaptic Models

2.2 Plasticity Models

2.1 Synaptic Models

We have learned how to model neurons in the previous chapters, so how are the neurons connected? How are the action potentials of neurons transmitted between different neurons? Here, we will introduce how to use BrainPy to simulate the communication between neurons.

2.1.1 Chemical Synapses

Biological Background

Figure 2-1 describes the biological process of information transmission between neurons. When the action potential of a presynaptic neuron is transmitted to the terminal of the axon, the presynaptic neuron releases **neurotransmitters** (also called transmitters). Neurotransmitters bind to receptors on postsynaptic neurons to cause changes in the membrane potential of postsynaptic neurons. These changes are called post-synaptic potential (PSP). Depending on the type of neurotransmitter, postsynaptic potentials can be excitatory or inhibitory. For example, **Glutamate** is an important excitatory neurotransmitter, while **GABA** is an important inhibitory neurotransmitter.

The binding of neurotransmitters and receptors may cause the opening of ion channels (**ionotype** receptors) or change the process of chemical reactions (**metabolic** receptors).

In this section, we will introduce how to use BrainPy to implement some common synapse models, mainly:

- **AMPA** and **NMDA**: They are both ionotropic receptors of glutamate, which can open ion channels directly after being bound. But NMDA is usually blocked by magnesium ions (Mg^{2+}) and cannot respond to the glutamate. Since magnesium ions are sensitive to voltage, when the postsynaptic potential exceeds the threshold of magnesium ions, magnesium ions will leave the NMDA channel, allowing NMDA to respond to glutamate. Therefore, the dynamics of NMDA are much slower than that of AMPA.
- **GABA_A** and **GABA_B**: They are two classes of GABA receptors, of which GABA_A are ionotropic receptors that typically produce fast inhibitory potentials; while GABA_B are metabotropic receptors that typically produce slow inhibitory potentials.

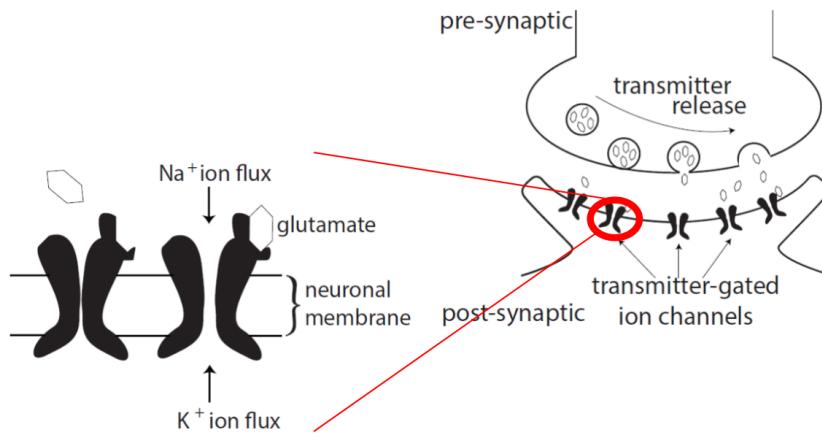


Fig. 2-1 Biological Synapse (Adapted from Gerstner et al., 2014 ¹)

In order to model the process from neurotransmitter release to the alteration of the membrane potential of postsynaptic neurons, we use a gating variable s to describe how many portion of ion channels will be opened whenever a presynaptic neuron generates an action potential. Let's start with an example of AMPA and see how to develop a synapse model and implement it with `BrainPy`.

AMPA Synapse

As we mentioned before, the AMPA receptor is an ionotropic receptor, that is, when a neurotransmitter binds to it, the ion channel will be opened immediately to allow Na⁺ and K⁺ ions flux.

We can use Markov process to describe the opening and closing process of ion channels. As shown in Fig. 2-2, s represents the probability of channel opening, $1 - s$ represents the probability of ion channel closing, and α and β are the transition probability. Because neurotransmitters can open ion channels, the transfer probability from $1 - s$ to s is affected by the concentration of neurotransmitters. We denote the concentration of neurotransmitters as [T].

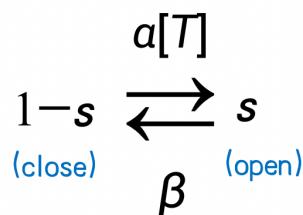


Fig. 2-2 Markov process of channel dynamics

We obtained the following formula when describing the process by a differential equation.

1.1 Biological background

$$\frac{ds}{dt} = \alpha[T](1 - s) - \beta s$$

Where $\alpha[T]$ denotes the transition probability from state $(1 - s)$ to state (s) ; and β represents the transition probability of the other direction.

Now let's see how to implement such a model with BrainPy. First of all, we need to define a class that inherits from `bp.TwoEndConn`, because synapses connect two neurons. Within the class, we can define the differential equation with `derivative` function, this is the same as the definition of neuron models. Then we use the `__init__` Function to initialize the required parameters and variables.

```

1 import brainpy as bp
2
3
4 class AMPA(bp.TwoEndConn):           → bp.TwoEndConn class:
5     target_backend = ['numpy', 'numba']   Connections between two neuron groups
6
7     @staticmethod
8     def derivative(s, t, TT, alpha, beta):    }  $\frac{ds}{dt} = \alpha[T](1 - s) - \beta s$ 
9         ds = alpha * TT * (1 - s) - beta * s
10        return ds
11
12     def __init__(self, pre, post, conn, alpha=0.98, beta=0.18, T=0.5,
13                  T_duration=0.5, **kwargs):
14         # parameters
15         self.alpha = alpha
16         self.beta = beta
17         self.T = T
18         self.T_duration = T_duration          } Regard [T] as a constant T, last for T_duration
19
20         # connections
21         self.conn = conn(pre.size, post.size)
22         self.pre_ids, self.post_ids = conn.requires('pre_ids', 'post_ids')
23         self.size = len(self.pre_ids)          } Specify how the presynaptic
24
25         # variables
26         self.s = bp.ops.zeros(self.size)
27         self.t_last_pre_spike = -1e7 * bp.ops.ones(self.size)      and postsynaptic neuron
28
29         self.int_s = bp.odeint(f=self.derivative, method='exponential_euler')
30         super(AMPA, self).__init__(pre=pre, post=post, **kwargs)      groups connect to each other
31

```

pre_ids	0	1	1	1	1	1	1	2
post_ids	2	3	3	3	3	4	4	3
syn_ids	0	1	2	3	4	5	6	7

We update s by an `update` function.

```

32     def update(self, _t):
33         for i in range(self.size):           → For each single synapse
34             pre_id = self.pre_ids[i]
35             post_id = self.post_ids[i]
36
37             if self.pre.spike[pre_id]:          } TT denotes [T], TT=T if pre neuron
38                 self.t_last_pre_spike[pre_id] = _t
39                 TT = ((_t - self.t_last_pre_spike[pre_id]) * self.T
40                         < self.T_duration) * self.T
41             self.s[i] = self.int_s(self.s[i], _t, TT, self.alpha, self.beta)
42

```

TT denotes $[T]$, $TT=T$ if pre neuron spikes within $T_{duration}$, otherwise $TT=0$.

After the implementation, we can plot the graph of s changing with time. We would first write a `run_syn` function to run and plot the graph. To run a synapse, we need neuron groups, so we use the LIF neuron provided by `brainmodels` package.

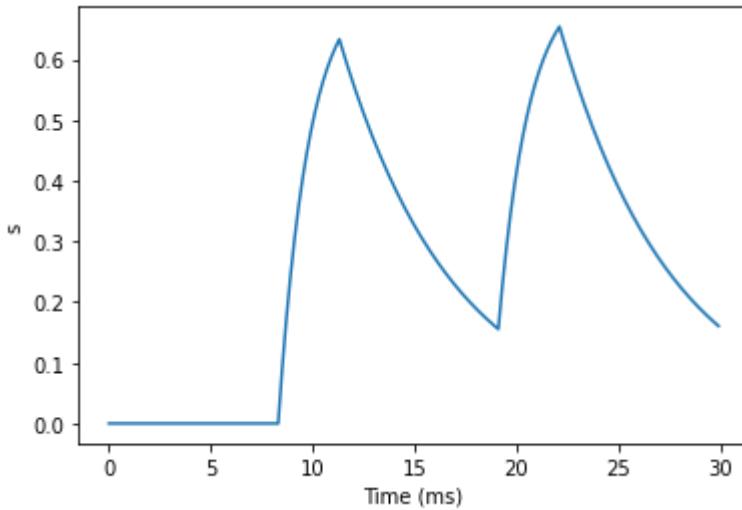
1.1 Biological background

```

43
44 import brainmodels as bm
45
46 bp.backend.set(backend='numba', dt=0.1)      -----> Set numba backend
47
48
49 def run_syn(syn_model, **kwargs):           -----> A method to run synapse
50     neu1 = bm.neurons.LIF(2, monitors=['V'])
51     neu2 = bm.neurons.LIF(3, monitors=['V'])      } Get two LIF neuron groups from
52
53     syn = syn_model(pre=neu1, post=neu2, conn=bp.connect.All2All(),
54                       monitors=['s'], **kwargs)           } brainmodels package
55
56     net = bp.Network(neu1, syn, neu2)
57     net.run(30., inputs=(neu1, 'input', 35.))
58     bp.visualize.line_plot(net.ts, syn.mon.s, ylabel='s', show=True)
59
60
61 run_syn(AMPA, T_duration=3.)      -----> Run AMPA synapse and
62                                         set parameter
63                                         T_duration to be 3.
64

```

Then we would expect to see the following result:



As can be seen from the above figure, when the presynaptic neurons fire, the value of s will first increase, and then decay.

NMDA Synapse

As we mentioned before, the NMDA receptors are blocked by Mg^{2+} , which would move away as the change of membrane potentials. We denote the concentration of Mg^{2+} as c_{Mg} , and its effect on membrane conductance g is given by,

$$g_\infty = \left(1 + e^{-\alpha V} \cdot \frac{c_{Mg}}{\beta}\right)^{-1}$$

$$g = \bar{g} \cdot g_\infty s$$

where g_∞ represents the effect of magnesium ion concentration, and its value decreases as the magnesium ion concentration increases. With the increase of the voltage V , the effect of c_{Mg} on g_∞ decreases. α, β

1.1 Biological background

and \bar{g} are some constants. The dynamic of gating variable s is similar with the AMPA synapse, which is given by,

$$\frac{ds}{dt} = -\frac{s}{\tau_{\text{decay}}} + ax(1-s)$$

$$\frac{dx}{dt} = -\frac{x}{\tau_{\text{rise}}}$$

if (pre fire), then $x \leftarrow x + 1$

where τ_{decay} and τ_{rise} are time constants of s decay and rise, respectively. a is a parameter.

Then let's implement the NMDA synapse with BrainPy. Here are the codes:

```

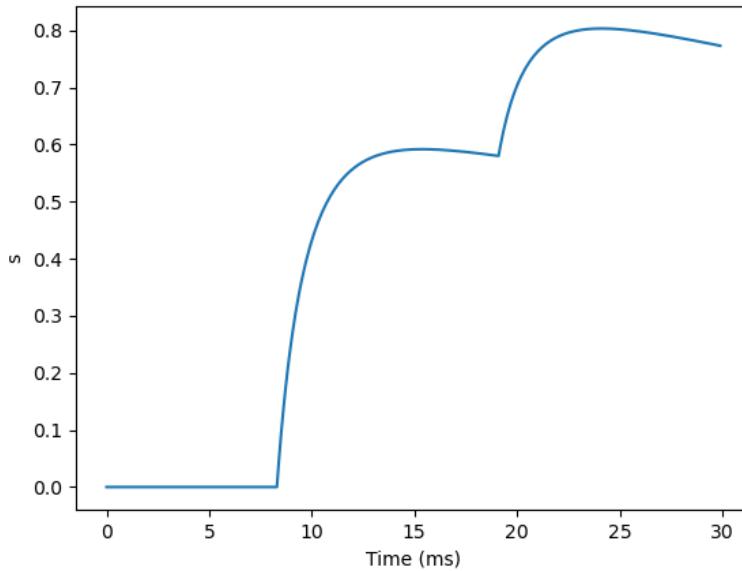
4  class NMDA(bp.TwoEndConn):
5      target_backend = ['numpy', 'numba']
6
7      @staticmethod
8      def derivative(s, x, t, tau_rise, tau_decay, a):           }    $\frac{ds}{dt} = -s/\tau_{\text{decay}} + ax(1-s)$ 
9          dsdt = -s / tau_decay + a * x * (1 - s)           }    $\frac{dx}{dt} = -x/\tau_{\text{rise}}$ 
10         dxdt = -x / tau_rise
11         return dsdt, dxdt
12
13     def __init__(self, pre, post, conn, delay=0., g_max=0.15, E=0., cc_Mg=1.2,
14                  alpha=0.062, beta=3.57, tau=100, a=0.5, tau_rise=2., **kwargs):
15         # parameters
16         self.g_max = g_max
17         self.E = E
18         self.alpha = alpha
19         self.beta = beta
20         self.cc_Mg = cc_Mg
21         self.tau = tau
22         self.tau_rise = tau_rise
23         self.a = a
24         self.delay = delay
25
26         # connections
27         self.conn = conn(pre.size, post.size)
28         self.pre_ids, self.post_ids = conn.requires('pre_ids', 'post_ids')
29         self.size = len(self.pre_ids)
30
31         # variables
32         self.s = bp.ops.zeros(self.size)
33         self.x = bp.ops.zeros(self.size)
34         self.g = self.register_constant_delay('g', size=self.size,
35                                              delay_time=delay)
36
37         self.integral = bp.odeint(f=self.derivative, method='rk4')
38
39         super(NMDA, self).__init__(pre=pre, post=post, **kwargs)
40
41     def update(self, _t):
42         for i in range(self.size):
43             pre_id = self.pre_ids[i]
44             post_id = self.post_ids[i]
45
46             self.x[i] += self.pre.spike[pre_id]           } if (pre spike), then x = x + 1
47             self.s[i], self.x[i] = self.integral(self.s[i], self.x[i], _t,
48                                                 self.tau_rise, self.tau,
49                                                 self.a)
50
51             # output
52             g_inf_exp = bp.ops.exp(-self.alpha * self.post.V[post_id])           }    $g_\infty = (1 + e^{-\alpha V} \frac{c_M g}{\beta})^{-1}$ 
53             g_inf = 1 + g_inf_exp * self.cc_Mg / self.beta
54
55             self.g.push(i, self.g_max * self.s[i] / g_inf)           }    $g = \bar{g} g_\infty s$ 
56
57             I_syn = self.g.pull(i) * (self.post.V[post_id] - self.E)           }    $I = g(V - E)$ 
58             self.post.input[post_id] -= I_syn
59

```

1.1 Biological background

As we've already defined the `run_syn` function while implementing the AMPA synapse, we can directly call it here.

```
run_syn(NMDA)
```



The result shows that the decay of NMDA is very slow and we barely see the decay from this 30ms simulation.

GABA_B Synapse

GABA_B is a metabotropic receptor, so it would not directly open the ion channels after the neurotransmitter binds to the receptor, but act through G protein as a second messenger. We denote $[R]$ as the proportions of activated receptors, and $[G]$ represents the concentration of activated G proteins. s is modulated by $[G]$, which is given by,

$$\frac{d[R]}{dt} = k_3[T](1 - [R]) - k_4[R]$$

$$\frac{d[G]}{dt} = k_1[R] - k_2[G]$$

$$s = \frac{[G]^4}{[G]^4 + K_d}$$

The kinetics of $[R]$ is similar to that of s in the AMPA model, which is affected by the neurotransmitter concentration $[T]$, and k_3, k_4 represent the transition probability. The dynamics of $[G]$ is affected by $[R]$ with parameters k_1, k_2 . K_d is a constant.

Here are the codes of the BrainPy implementation.

1.1 Biological background

```

3   class GABAB(bp.TwoEndConn):
4       target_backend = ['numpy', 'numba']
5
6       @staticmethod
7       def derivative(R, G, t, k3, TT, k4, k1, k2):
8           dRdt = k3 * TT * (1 - R) - k4 * R
9           dGdt = k1 * R - k2 * G
10          return dRdt, dGdt
11
12      def __init__(self, pre, post, conn, delay=0., g_max=0.02, E=-95.,
13                   k1=0.18, k2=0.034, k3=0.09, k4=0.0012, kd=100., T=0.5,
14                   T_duration=0.3, **kwargs):
15          # params
16          self.g_max = g_max
17          self.E = E
18          self.k1 = k1
19          self.k2 = k2
20          self.k3 = k3
21          self.k4 = k4
22          self.kd = kd
23          self.T = T
24          self.T_duration = T_duration
25
26          # connns
27          self.conn = conn(pre.size, post.size)
28          self.pre_ids, self.post_ids = conn.requires('pre_ids', 'post_ids')
29          self.size = len(self.pre_ids)
30
31          # data
32          self.R = bp.ops.zeros(self.size)
33          self.G = bp.ops.zeros(self.size)
34          self.t_last_pre_spike = bp.ops.ones(self.size) * -1e7
35          self.s = bp.ops.zeros(self.size)
36          self.g = self.register_constant_delay('g', size=self.size,
37                                              delay_time=delay)
38
39          self.integral = bp.odeint(f=self.derivative, method='rk4')
40          super(GABAB, self).__init__(pre=pre, post=post, **kwargs)

42      def update(self, _t):
43          for i in range(self.size):
44              pre_id = self.pre_ids[i]
45              post_id = self.post_ids[i]
46
47              if self.pre.spike[pre_id]:
48                  self.t_last_pre_spike[i] = _t
49                  TT = ((_t - self.t_last_pre_spike[i]) < self.T_duration) * self.T
50
51                  self.R[i], G = self.integral(self.R[i], self.G[i], _t, self.k3,
52                                              TT, self.k4, self.k1, self.k2)
53                  self.s[i] = G ** 4 / (G ** 4 + self.kd)
54                  self.G[i] = G
55
56                  self.g.push(i, self.g_max * self.s[i])
57                  I_syn = self.g.pull(i) * (self.post.V[post_id] - self.E)
58                  self.post.input[post_id] -= I_syn

```

} TT denotes [T], TT=T if pre neuron spikes within T_duration, otherwise TT=0.

} $s = [G]^4 / ([G]^4 + K_d)$

} $I = g(V - E)$

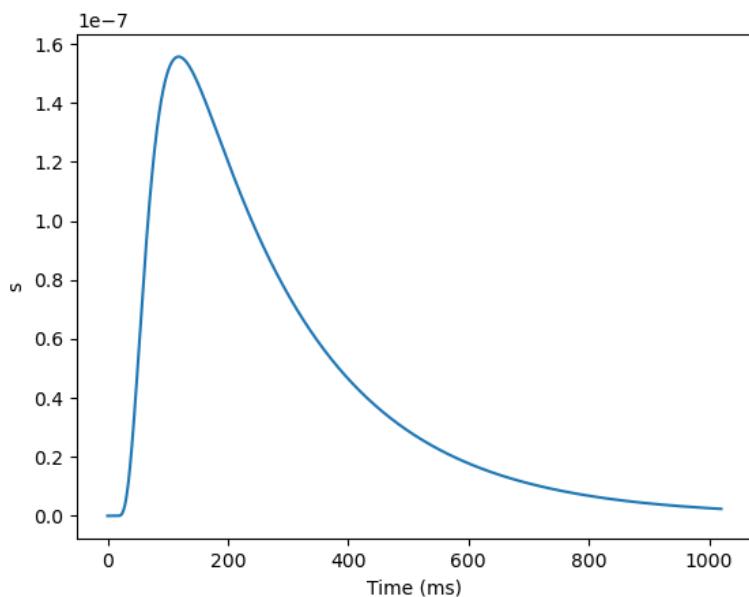
Since the dynamics of GABA_B is very slow, we no longer use the `run_syn` function here, but use `bp.inputs.constant_current` to give stimulation for 20ms, and then look at the decay at the subsequent 1000ms without external inputs.

1.1 Biological background

```
neu1 = bm.neurons.LIF(2, monitors=['V'])
neu2 = bm.neurons.LIF(3, monitors=['V'])
syn = GABAB(pre=neu1, post=neu2, conn=bp.connect.All2All(),
net = bp.Network(neu1, syn, neu2)

# input
I, dur = bp.inputs.constant_current([(25, 20), (0, 1000)])
net.run(dur, inputs=(neu1, 'input', I))

bp.visualize.line_plot(net.ts, syn.mon.s, ylabel='s', show=
```



The result shows that the decay of GABA_B lasts for hundreds of milliseconds.

Current-based and Conductance-based synapses

You may have noticed that the modeling of the gating variable s of the GABA_B synapse did not show its property of inducing inhibitory potentials. To demonstrate the excitatory and inhibitory properties, we not only need to model the gating variable s , but also the current I through the synapses (as input to the postsynaptic neurons). There are two different methods to model the relationships between s and I : **current-based** and **conductance-based**. The main difference between them is whether the synaptic current is influenced by the membrane potential of postsynaptic neurons.

(1) Current-based

The formula of the current-based model is as follow:

$$I \propto s$$

While coding, we usually multiply s by a weight w . We can implement excitatory and inhibitory synapses by adjusting the positive and negative values of the weight w .

The delay of synapses is implemented by applying the delay time to the `I_syn` variable using the `register_constant_delay` function provided by BrainPy.

```

1  def __init__(self, pre, post, conn, delay, **kwargs):
2      # ...
3      self.s = bp.ops.zeros(self.size)
4      self.w = bp.ops.ones(self.size) * .2
5      self.I_syn = self.register_constant_delay('I_syn', size=self.size,
6                                                 delay_time=delay)           ──> set delay time before
7                                              changing synaptic current
8                                              by using the
9                                              register_constant_delay
10                                             method
11                                             # ...
12                                             self.I_syn.push(i, self.w[i] * self.s[i])    ] use push and pull to set delay before applying
13                                             self.post.input[post_id] += self.I_syn.pull(i)  synaptic current into postsynaptic input.

```

(2) Conductance-based

In the conductance-based model, the conductance is $g = \bar{g}s$. Therefore, according to Ohm's law, the formula is given by:

$$I = \bar{g}s(V - E)$$

Here E is a reverse potential, which can determine whether the effect of I is inhibition or excitation. For example, when the resting potential is about -65, subtracting a lower E , such as -75, will become positive, thus will change the direction of the current in the formula and produce the suppression current. The E value of excitatory synapses is relatively high, such as 0.

In terms of implementation, you can apply a synaptic delay to the variable `g`.

```

1  def __init__(self, pre, post, conn, g_max, E, delay, **kwargs):
2      self.g_max = g_max
3      self.E = E
4      # ...
5      self.s = bp.ops.zeros(self.size)
6      self.g = self.register_constant_delay('g', size=self.size, delay_time=delay)   ──> set delay time
7                                              before changing the
8                                              conductance
9      def update(self, _t):
10         for i in range(self.size):
11             # ...
12             self.g.push(i, self.g_max * self.s[i])       ──> g = \bar{g}s
13             self.post.input[post_id] -= self.g.pull(i) * (self.post.V[post_id] - self.E)   ──> I = g(V - E)
14

```

Now let's review the NMDA and GABA_B synapses we just implemented. They are both conductance-based models. Excitatory currents are induced by $E = 0$ in the NMDA synapse; while inhibitory currents are

produced by $E = -95$ in the GABA_B synapse. You can find the complete code to the above models from our [BrainModels](#) GitHub repository.

Abstract reduced models

The dynamics of the gating variables s of the above chemical synapses sharing the same characteristic of rising first and then decay. Sometimes we don't need to use models that specifically correspond to biological synapses. Therefore, some abstract synaptic models have been proposed. Here, we will introduce the implementation of four abstract models on BrainPy, they can be either current-based or conductance-based. These models can also be found in the [BrainModels](#) repository.

(1) Differences of two exponentials

Let's first introduce the `Differences of two exponentials` model, its dynamics is given by,

$$s = \frac{\tau_1 \tau_2}{\tau_1 - \tau_2} \left(\exp\left(-\frac{t - t_s}{\tau_1}\right) - \exp\left(-\frac{t - t_s}{\tau_2}\right) \right)$$

Where t_s denotes the spike timing of the presynaptic neuron, τ_1 and τ_2 are time constants.

While implementing with BrainPy, we use the following differential equation form,

$$\begin{aligned} \frac{ds}{dt} &= x \\ \frac{dx}{dt} &= -\frac{\tau_1 + \tau_2}{\tau_1 \tau_2} x - \frac{s}{\tau_1 \tau_2} \\ \text{if (fire), then } x &\leftarrow x + 1 \end{aligned}$$

Here we specify the logic of increment of x in the `update` function when the presynaptic neurons fire. The code is as follows:

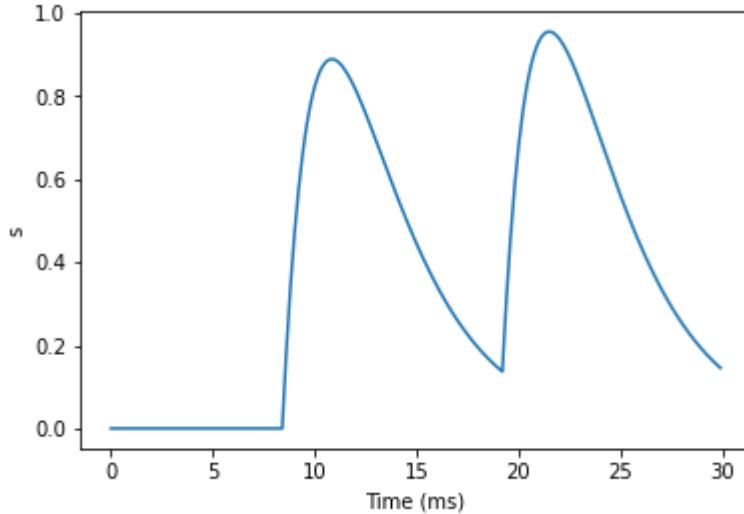
1.1 Biological background

```

1  class Two_exponentials(bp.TwoEndConn):
2      target_backend = ['numpy', 'numba']
3
4      @staticmethod
5      def derivative(s, x, t, tau1, tau2):
6          dxdt = (-(tau1 + tau2) * x - s) / (tau1 * tau2)
7          dsdt = x
8          return dsdt, dxdt
9
10     def __init__(self, pre, post, conn, tau1=1.0, tau2=3.0, **kwargs):
11         # parameters
12         self.tau1 = tau1
13         self.tau2 = tau2
14
15         # connections
16         self.conn = conn(pre.size, post.size)
17         self.pre_ids, self.post_ids = conn.requires('pre_ids', 'post_ids')
18         self.size = len(self.pre_ids)
19
20         # variables
21         self.s = bp.ops.zeros(self.size)
22         self.x = bp.ops.zeros(self.size)
23
24         self.integral = bp.odeint(f=self.derivative, method='rk4')
25
26     super(Two_exponentials, self).__init__(pre=pre, post=post, **kwargs)
27
28     def update(self, _t):
29         for i in range(self.size):
30             pre_id = self.pre_ids[i]
31
32             self.s[i], self.x[i] = self.integral(self.s[i], self.x[i], _t,
33                                                 self.tau1, self.tau2)
34             self.x[i] += self.pre.spike[pre_id]  → if (pre spike), then x = x + 1
35
36
37 run_syn(Two_exponentials, tau1=2.)

```

Then we expect to see the following result:



(2) Alpha synapse

Dynamics of `Alpha synapse` is given by,

$$s = \frac{t - t_s}{\tau} \exp\left(-\frac{t - t_s}{\tau}\right)$$

As the dual exponential synapse, t_s denotes the spike timing of the presynaptic neuron, with a time constant τ .

1.1 Biological background

The differential equation form of alpha synapse is also very similar with the dual exponential synapses, with $\tau = \tau_1 = \tau_2$, as shown below:

$$\frac{ds}{dt} = x$$

$$\frac{dx}{dt} = -\frac{2x}{\tau} - \frac{s}{\tau^2}$$

if (fire), then $x \leftarrow x + 1$

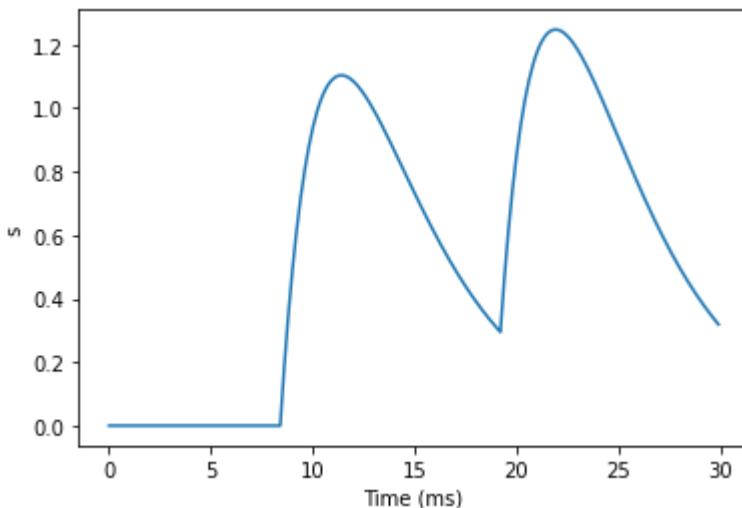
Code implementation is similar:

```

1  class Alpha(bp.TwoEndConn):
2      target_backend = ['numpy', 'numba']
3
4      @staticmethod
5      def derivative(s, x, t, tau):
6          dxdt = (-2 * tau * x - s) / (tau ** 2)
7          dsdt = x
8          return dsdt, dxdt
9
10     def __init__(self, pre, post, conn, tau=3.0, **kwargs):
11         # parameters
12         self.tau = tau
13
14         # connections
15         self.conn = conn(pre.size, post.size)
16         self.pre_ids, self.post_ids = conn.requires('pre_ids', 'post_ids')
17         self.size = len(self.pre_ids)
18
19         # variables
20         self.s = bp.ops.zeros(self.size)
21         self.x = bp.ops.zeros(self.size)
22
23         self.integral = bp.odeint(f=self.derivative, method='rk4')
24
25     super(Alpha, self).__init__(pre=pre, post=post, **kwargs)
26
27     def update(self, _t):
28         for i in range(self.size):
29             pre_id = self.pre_ids[i]
30
31             self.s[i], self.x[i] = self.integral(self.s[i], self.x[i], _t,
32                                                 self.tau)
33             self.x[i] += self.pre.spike[pre_id]  ────────── if (pre spike), then x=x+1
34
35
36 run_syn(Alpha)

```

Then we expect to see the following result:



(3) Single exponential decay

Sometimes we can ignore the rising process in modeling, and only need to model the decay process. Therefore, the formula of `single exponential decay` model is more simplified:

$$\frac{ds}{dt} = -\frac{s}{\tau_{decay}}$$

if (fire), then $s \leftarrow s + 1$

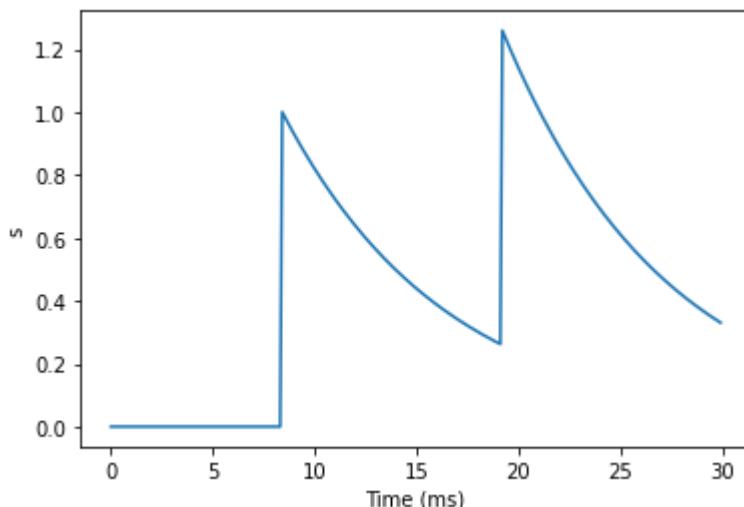
The implementing code is given by:

```

1  class Exponential(bp.TwoEndConn):
2      target_backend = ['numpy', 'numba']
3
4      @staticmethod
5      def derivative(s, t, tau):
6          ds = -s / tau      ——————>  $\frac{ds}{dt} = -s/\tau$ 
7          return ds
8
9      def __init__(self, pre, post, conn, tau=8.0, **kwargs):
10         # parameters
11         self.tau = tau
12
13         # connections
14         self.conn = conn(pre.size, post.size)
15         self.pre_ids, self.post_ids = conn.requires('pre_ids', 'post_ids')
16         self.size = len(self.pre_ids)
17
18         # variables
19         self.s = bp.ops.zeros(self.size)
20
21         self.integral = bp.odeint(f=self.derivative, method='exponential_euler')
22
23     super(Exponential, self).__init__(pre=pre, post=post, **kwargs)
24
25    def update(self, _t):
26        for i in range(self.size):
27            pre_id = self.pre_ids[i]
28
29            self.s[i] = self.integral(self.s[i], _t, self.tau)
30            self.s[i] += self.pre.spike[pre_id]      ——————> if (pre spike), then  $s = s + 1$ 
31
32
33 run_syn(Exponential)

```

Then we expect to see the following result:



(4) Voltage jump

1.1 Biological background

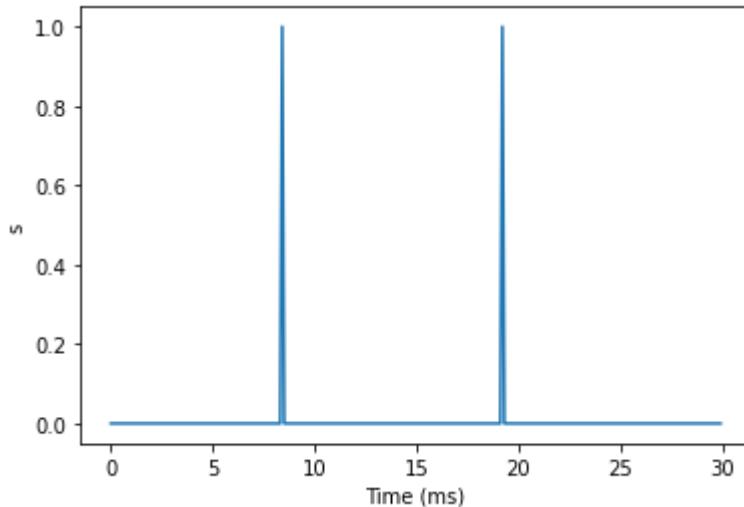
Sometimes even the decay process can be ignored, so there is a **voltage jump** model, which is given by:

if (fire), then $s \leftarrow s + 1$

The code is as follows:

```
1  class Voltage_jump(bp.TwoEndConn):
2      target_backend = ['numpy', 'numba']
3
4      def __init__(self, pre, post, conn, **kwargs):
5          # connections
6          self.conn = conn(pre.size, post.size)
7          self.pre_ids, self.post_ids = conn.requires('pre_ids', 'post_ids')
8          self.size = len(self.pre_ids)
9
10         # variables
11         self.s = bp.ops.zeros(self.size)
12
13     super(Voltage_jump, self).__init__(pre=pre, post=post, **kwargs)
14
15    def update(self, _t):
16        for i in range(self.size):
17            pre_id = self.pre_ids[i]
18            self.s[i] = self.pre.spike[pre_id]  ──────────────────> if (pre spike), then s = s + 1
19
20
21 run_syn(Voltage_jump)
```

Then we expect to see the following result:



2.1.2 Electrical Synapses

In addition to the chemical synapses described earlier, electrical synapses are also common in our neural system.

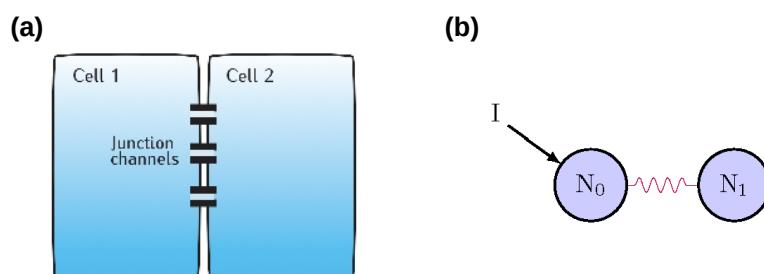


Fig. 2-3 (a) Gap junction connection between two cells. **(b)** An equivalent diagram.(From Starrant et al., 2011²)

As shown in the Fig. 2-3a, two neurons are connected by junction channels and can conduct electricity directly. Therefore, it can be seen that two neurons are connected by a constant resistance, as shown in the Fig. 2-3b.

According to Ohm's law, the current of one neuron is given by,

$$I_1 = w(V_0 - V_1)$$

where V_0 and V_1 are the membrane potentials of the two neurons, and the synaptic weight w is equivalent with the conductance.

While implementing with BrainPy, you only need to specify the equation in the `update` function.

```

1  class Gap_junction(bp.TwoEndConn):
2      target_backend = ['numpy', 'numba']
3
4      def __init__(self, pre, post, conn, delay=0., k_spikelet=0.1,
5                   post_refractory=False, **kwargs):
6          self.delay = delay
7          self.k_spikelet = k_spikelet
8          self.post_has_refractory = post_refractory
9
10         # connections
11         self.conn = conn(pre.size, post.size)
12         self.pre_ids, self.post_ids = conn.requires('pre_ids', 'post_ids')
13         self.size = len(self.pre_ids)
14
15         # variables
16         self.w = bp.ops.ones(self.size)
17         self.spikelet = self.register_constant_delay('spikelet', size=self.size,
18                                                       delay_time=self.delay)
19
20     super(Gap_junction, self).__init__(pre=pre, post=post, **kwargs)
21
22     def update(self, _t):
23         for i in range(self.size):
24             pre_id = self.pre_ids[i]
25             post_id = self.post_ids[i]
26
27             self.post.input[post_id] += self.w[i] * (self.pre.V[pre_id] -
28                                                 self.post.V[post_id])           ] } Ipost = w (Vpre - Vpost)
29
30             self.spikelet.push(i, self.w[i] * self.k_spikelet *
31                               self.pre.spike[pre_id])
32
33             out = self.spikelet.pull(i)
34             if self.post_has_refractory:
35                 self.post.V[post_id] += out * (1. - self.post.refractory[post_id])
36             else:
37                 self.post.V[post_id] += out
38

```

Then we can run a simulation.

1.1 Biological background

```
import matplotlib.pyplot as plt

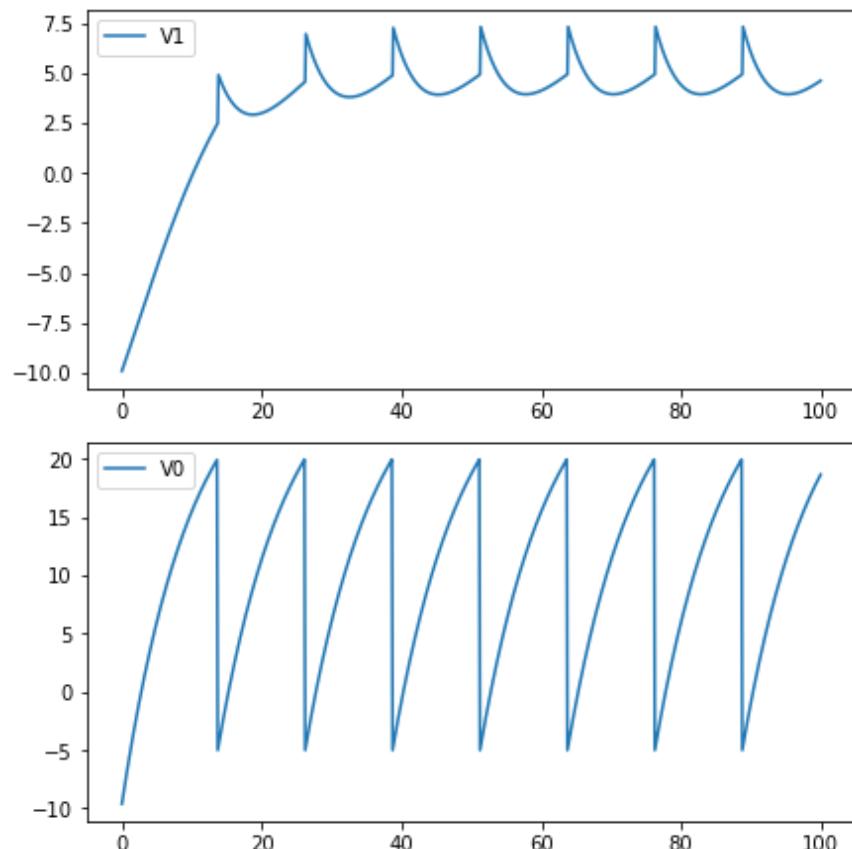
neu0 = bm.neurons.LIF(1, monitors=['V'], t_refractory=0)
neu0.V = bp.ops.ones(neu0.V.shape) * -10.
neu1 = bm.neurons.LIF(1, monitors=['V'], t_refractory=0)
neu1.V = bp.ops.ones(neu1.V.shape) * -10.
syn = Gap_junction(pre=neu0, post=neu1, conn=bp.connect.All
                     k_spikelet=5.)
syn.w = bp.ops.ones(syn.w.shape) * .5

net = bp.Network(neu0, neu1, syn)
net.run(100., inputs=(neu0, 'input', 30.))

fig, gs = bp.visualize.get_figure(row_num=2, col_num=1, )

fig.add_subplot(gs[1, 0])
plt.plot(net.ts, neu0.mon.V[:, 0], label='V0')
plt.legend()

fig.add_subplot(gs[0, 0])
plt.plot(net.ts, neu1.mon.V[:, 0], label='V1')
plt.legend()
plt.show()
```



References

- [1] Gerstner, Wulfram, et al. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.
- [2] Sterratt, David, et al. *Principles of computational modelling in neuroscience*. Cambridge University Press, 2011.

2.2 Plasticity Models

We just talked about synaptic dynamics, but we haven't talked about synaptic plasticity. Next, let's see how to use BrainPy to implement synaptic plasticity.

Plasticity mainly distinguishes short-term plasticity from long-term plasticity. We will first introduce short-term plasticity (STP), and then introduce several different models of long-term synaptic plasticity (also known as learning rules).

The introduction is as follows:

- Short-term plasticity
- Long-term plasticity
 - Spike-timing dependent plasticity
 - Rate-based Hebb rule
 - Oja's rule
 - BCM rule

2.2.1 Short-term plasticity (STP)

Let's first look at short-term plasticity. We will start with the results of the experiment. Fig. 2-4 shows the changes of the membrane potential of postsynaptic neurons as the firing of presynaptic neurons. We can see that when the presynaptic neurons repeatedly firing with short intervals, the response of the postsynaptic neurons becomes weaker and weaker, showing a short term depression. But the response recovers after a short period of time, so this plasticity is short-term.

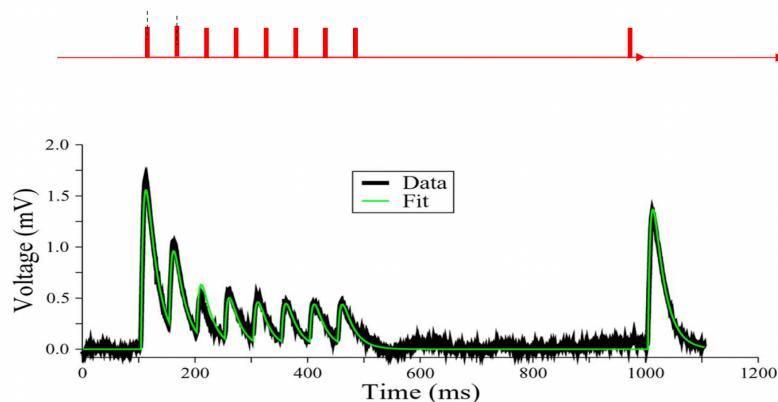


Fig. 2-4 Short-term plasticity. (Adapted from Gerstner et al., 2014 [1](#))

1.1 Biological background

Now let's turn to the model [2](#). The short term plasticity can be described by two variables, u and x . Where u represents the probability of neurotransmitter release, and x represents the residual amount of neurotransmitters. The dynamic of a synapse with short term plasticity is given by,

$$\begin{aligned}\frac{dI}{dt} &= -\frac{I}{\tau} \\ \frac{du}{dt} &= -\frac{u}{\tau_f} \\ \frac{dx}{dt} &= \frac{1-x}{\tau_d} \\ \text{if (pre fire), then } &\begin{cases} u^+ = u^- + U(1-u^-) \\ I^+ = I^- + Au^+x^- \\ x^+ = x^- - u^+x^- \end{cases}\end{aligned}$$

where the dynamics of the synaptic current I can be one of the dynamics we introduced in the previous section (i.e., the dynamic of gating variable s under current-based condition). U and A are two constants representing the increments of u and I after a presynaptic spike, respectively. τ_f and τ_d are time constants of u and x , respectively.

In this model, u contributes to the short-term facilitation (STF) by increasing from 0 whenever there is a spike on the presynaptic neuron; while x contributes to the short-term depression (STD) by decreasing from 1 after the presynaptic spike. The two directions of facilitation and depression occur simultaneously, and the value of τ_f and τ_d determines which direction of plasticity plays a dominant role.

Now let's see how to implement the STP model with BrainPy. We can see that the plasticity also occurs in synapses, so we will define the class by inheriting from the `bp.TwoEndConn` class like synaptic models. The code is as follows:

1.1 Biological background

```

9  class STP(bp.TwoEndConn):
10    target_backend = ['numpy', 'numba']
11
12    @staticmethod
13    def derivative(s, u, x, t, tau, tau_d, tau_f):
14        dsdt = - s / tau
15        dudt = - u / tau_f
16        dxdt = (1 - x) / tau_d
17        return dsdt, dudt, dxdt
18
19    def __init__(self, pre, post, conn, delay=0., U=0.15, tau_f=1500.,
20                 tau_d=200., tau=8., **kwargs):
21        # parameters
22        self.tau_d = tau_d
23        self.tau_f = tau_f
24        self.tau = tau
25        self.U = U
26        self.delay = delay
27
28        # connections
29        self.conn = conn(pre.size, post.size)
30        self.pre_ids, self.post_ids = conn.requires('pre_ids', 'post_ids')
31        self.size = len(self.pre_ids)
32
33        # variables
34        self.s = bp.ops.zeros(self.size)
35        self.x = bp.ops.ones(self.size)
36        self.u = bp.ops.zeros(self.size)
37        self.w = bp.ops.ones(self.size)
38        self.I_syn = self.register_constant_delay('I_syn', size=self.size,
39                                               delay_time=delay)
40
41        self.integral = bp.odeint(f=self.derivative, method='exponential_euler')
42
43        super(STP, self).__init__(pre=pre, post=post, **kwargs)
44
45
46    def update(self, _t):
47        for i in range(self.size):
48            pre_id = self.pre_ids[i]
49
50            self.s[i], u, x = self.integral(self.s[i], self.u[i], self.x[i], _t,
51                                             self.tau, self.tau_d, self.tau_f)
52
53            if self.pre.spike[pre_id] > 0:
54                u += self.U * (1 - self.u[i])
55                self.s[i] += self.w[i] * u * self.x[i]
56                x -= u * self.x[i]
57                self.u[i] = u
58                self.x[i] = x
59
60            # output
61            post_id = self.post_ids[i]
62            self.I_syn.push(i, self.s[i])
63            self.post.input[post_id] += self.I_syn.pull(i)
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93

```

Then let's define a function to run the code. Like synapse models, we need two neuron groups to be connected. Besides the dynamic of s , we also want to see how u and x changes over time, so we monitor 's', 'u' and 'x' and plot them.

1.1 Biological background

```
def run_stp(**kwargs):
    neu1 = bm.neurons.LIF(1, monitors=['V'])
    neu2 = bm.neurons.LIF(1, monitors=['V'])

    syn = STP(pre=neu1, post=neu2, conn=bp.connect.All2All(
        monitors=['s', 'u', 'x'], **kwargs)
    net = bp.Network(neu1, syn, neu2)
    net.run(100., inputs=(neu1, 'input', 28.))

    # plot
    fig, gs = bp.visualize.get_figure(2, 1, 3, 7)

    fig.add_subplot(gs[0, 0])
    plt.plot(net.ts, syn.mon.u[:, 0], label='u')
    plt.plot(net.ts, syn.mon.x[:, 0], label='x')
    plt.legend()

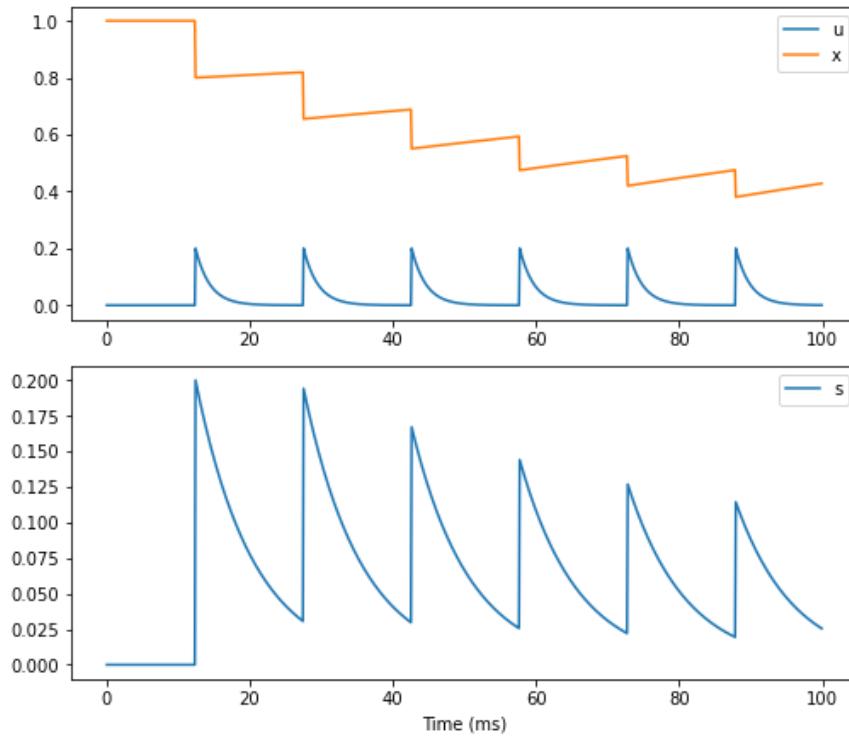
    fig.add_subplot(gs[1, 0])
    plt.plot(net.ts, syn.mon.s[:, 0], label='s')
    plt.legend()

    plt.xlabel('Time (ms)')
    plt.show()
```

Let's first set `tau_d > tau_f`.

```
run_stp(U=0.2, tau_d=150., tau_f=2.)
```

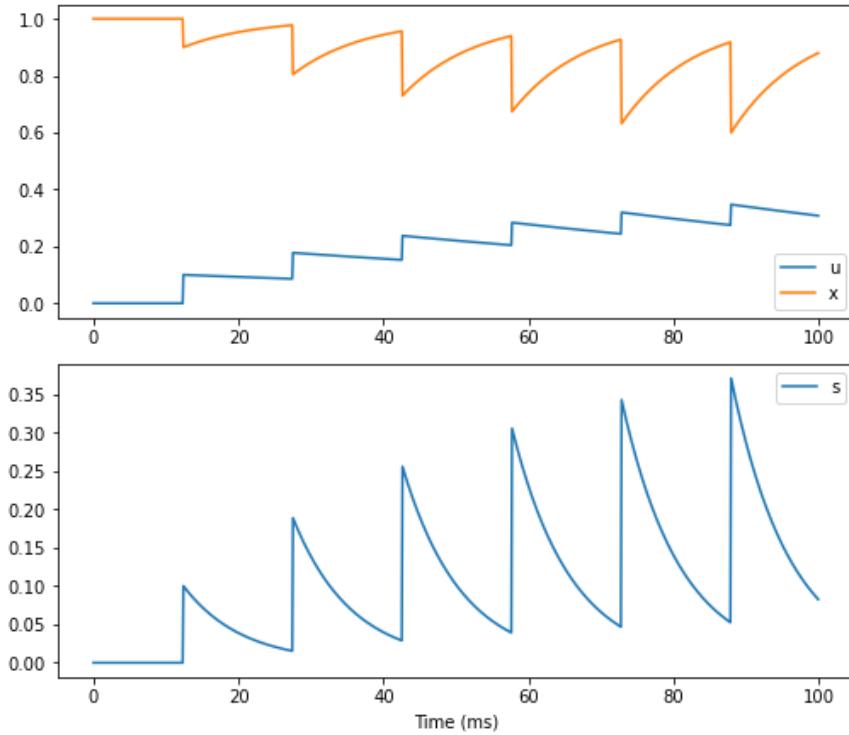
1.1 Biological background



The plots show that when we set the parameters $\tau_d > \tau_f$, x recovers very slowly, and u decays very quickly, so in the end, the transmitter is not enough to open the receptors, showing STD dominants.

Then let's set `tau_f > tau_d`.

```
run_stp(U=0.1, tau_d=10, tau_f=100.)
```



We can see from the figure that when we set $\tau_f > \tau_d$, on the contrary, every time x is used, it will be added back quickly. There are always enough transmitters available. At the same time, the decay of u is very slow, so the probability of releasing transmitters is getting higher and higher, showing STF dominants.

2.2.2 Long-term Plasticity

Spike-timing dependent plasticity (STDP)

Fig. 2-5 shows the spiking timing dependent plasticity (STDP) of experimental results. The x-axis is the time difference between the spike of the presynaptic neuron and the postsynaptic neuron. The left part of the zero represents the spike timing of the presynaptic neuron earlier than that of the postsynaptic neuron, which shows long term potentiation (LTP); and the right side of the zero represents the postsynaptic neuron fires before the presynaptic neuron does, showing long term depression (LTD).

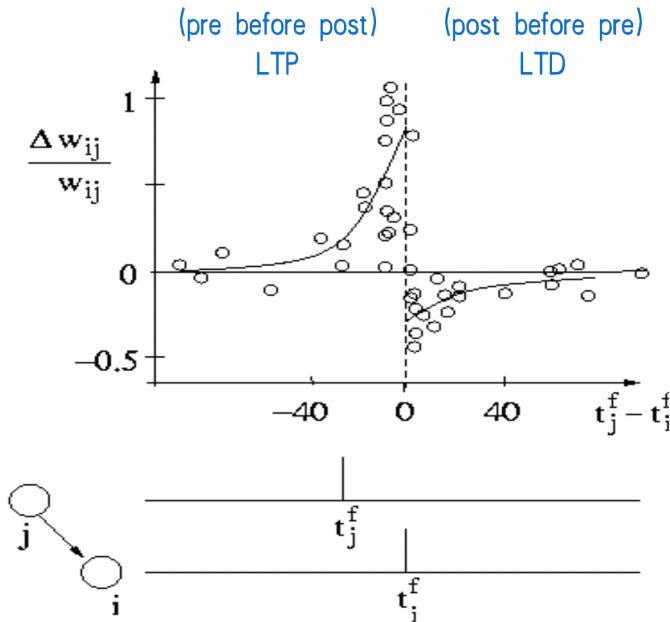


Fig. 2-5 Spike timing dependent plasticity. (Adapted from Bi & Poo, 2001³)

The computational model of STDP is given by,

$$\frac{dA_s}{dt} = -\frac{A_s}{\tau_s}$$

$$\frac{dA_t}{dt} = -\frac{A_t}{\tau_t}$$

if (pre fire), then $\begin{cases} s \leftarrow s + w \\ A_s \leftarrow A_s + \Delta A_s \\ w \leftarrow w - A_t \end{cases}$

if (post fire), then $\begin{cases} A_t \leftarrow A_t + \Delta A_t \\ w \leftarrow w + A_s \end{cases}$

Where w is the synaptic weight, and s is the same gating variable as we mentioned in the previous section. Like the STP model, LTD and LTP are controlled by two variables A_s and A_t , respectively. ΔA_s and ΔA_t are the increments of A_s and A_t , respectively. τ_s and τ_t are time constants.

According to this model, when a presynaptic neuron fire before the postsynaptic neuron, A_s increases everytime when there is a spike on the presynaptic neuron, and A_t will stay on 0 until the postsynaptic neuron fire, so w will not change for the time being. When there is a spike in the postsynaptic neuron, the increment of w will be an amount of $A_s - A_t$, since $A_s > A_t$ in this situation, LTP will be presented, and vice versa.

1.1 Biological background

Now let's see how to use BrainPy to implement this model. Here we use the single exponential decay model to implement the dynamics of s .

```

9   class STDP(bp.TwoEndConn):
10      target_backend = ['numpy', 'numba']
11
12      @staticmethod
13      def derivative(s, A_s, A_t, t, tau, tau_s, tau_t):
14          dsdt = -s / tau
15          dAsdt = -A_s / tau_s
16          dAtdt = -A_t / tau_t
17          return dsdt, dAsdt, dAtdt
18
19      def __init__(self, pre, post, conn, delay=0., delta_A_s=0.5,
20                  delta_A_t=0.5, w_min=0., w_max=20., tau_s=10.,
21                  tau_t=10., **kwargs):
22          # parameters
23          self.tau_s = tau_s
24          self.tau_t = tau_t
25          self.tau = tau
26          self.delta_A_s = delta_A_s
27          self.delta_A_t = delta_A_t
28          self.w_min = w_min
29          self.w_max = w_max
30          self.delay = delay
31
32          # connections
33          self.conn = conn(pre.size, post.size)
34          self.pre_ids, self.post_ids = self.conn.requires('pre_ids', 'post_ids')
35          self.size = len(self.pre_ids)
36
37          # variables
38          self.s = bp.ops.zeros(self.size)
39          self.A_s = bp.ops.zeros(self.size)
40          self.A_t = bp.ops.zeros(self.size)
41          self.w = bp.ops.ones(self.size) * 1.
42          self.I_syn = self.register_constant_delay('I_syn', size=self.size,
43                                                 delay_time=delay)
44          self.integral = bp.odeint(f=self.derivative, method='exponential_euler')
45
46          super(STDP, self).__init__(pre=pre, post=post, **kwargs)
47
48      def update(self, _t):
49          for i in range(self.size):
50              pre_id = self.pre_ids[i]
51              post_id = self.post_ids[i]
52
53              self.s[i], A_s, A_t = self.integral(self.s[i], self.A_s[i],
54                                                self.A_t[i], _t, self.tau,
55                                                self.tau_s, self.tau_t)
56
57              w = self.w[i]
58              if self.pre.spike[pre_id] > 0:
59                  self.s[i] += w
60                  A_s += self.delta_A_s
61                  w -= A_t
62
63              if self.post.spike[post_id] > 0:
64                  A_t += self.delta_A_t
65                  w += A_s
66
67              self.A_s[i] = A_s
68              self.A_t[i] = A_t
69
70              self.w[i] = bp.ops.clip(w, self.w_min, self.w_max)           ————— Limit w within the boundary
71
72          # output
73          self.I_syn.push(i, self.s[i])
74          self.post.input[post_id] += self.I_syn.pull(i)
75

```

We control the spike timing by varying the input current of the presynaptic group and postsynaptic group. We apply the first input to the presynaptic group starting at $t = 5\text{ms}$ (with amplitude of $30 \mu\text{A}$, lasts for 15 ms to ensure to induce a spike with LIF neuron model), then start to stimulate the postsynaptic group at $t = 10\text{ms}$. The intervals between

1.1 Biological background

each two inputs are 15ms. We keep those $t_{post} = t_{pre} + 5$ during the first 3 spike-pairs. Then we set a long interval before switching the stimulating order to be $t_{post} = t_{pre} - 3$ since the 4th spike.

```
duration = 300.  
(I_pre, _) = bp.inputs.constant_current([(0, 5), (30, 15),  
                                         (0, 15), (30, 15),  
                                         (0, 15), (30, 15),  
                                         (0, 98), (30, 15), # switch order: t_inte  
                                         (0, 15), (30, 15),  
                                         (0, 15), (30, 15),  
                                         (0, duration-155-98)])  
(I_post, _) = bp.inputs.constant_current([(0, 10), (30, 15)  
                                         (0, 15), (30, 15),  
                                         (0, 15), (30, 15),  
                                         (0, 90), (30, 15), # switch order: t_inte  
                                         (0, 15), (30, 15),  
                                         (0, 15), (30, 15),  
                                         (0, duration-160-90)])
```



Then let's run the simulation.

1.1 Biological background

```
pre = bm.neurons.LIF(1, monitors=['spike'])
post = bm.neurons.LIF(1, monitors=['spike'])

syn = STDP(pre=pre, post=post, conn=bp.connect.All2All(),
           monitors=['s', 'w'])
net = bp.Network(pre, syn, post)
net.run(duration, inputs=[(pre, 'input', I_pre), (post, 'ir

# plot
fig, gs = bp.visualize.get_figure(4, 1, 2, 7)

def hide_spines(my_ax):
    plt.legend()
    plt.xticks([])
    plt.yticks([])
    my_ax.spines['left'].set_visible(False)
    my_ax.spines['right'].set_visible(False)
    my_ax.spines['bottom'].set_visible(False)
    my_ax.spines['top'].set_visible(False)

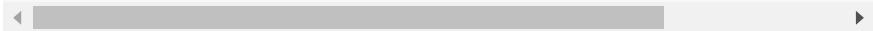
ax=fig.add_subplot(gs[0, 0])
plt.plot(net.ts, syn.mon.s[:, 0], label="s")
hide_spines(ax)

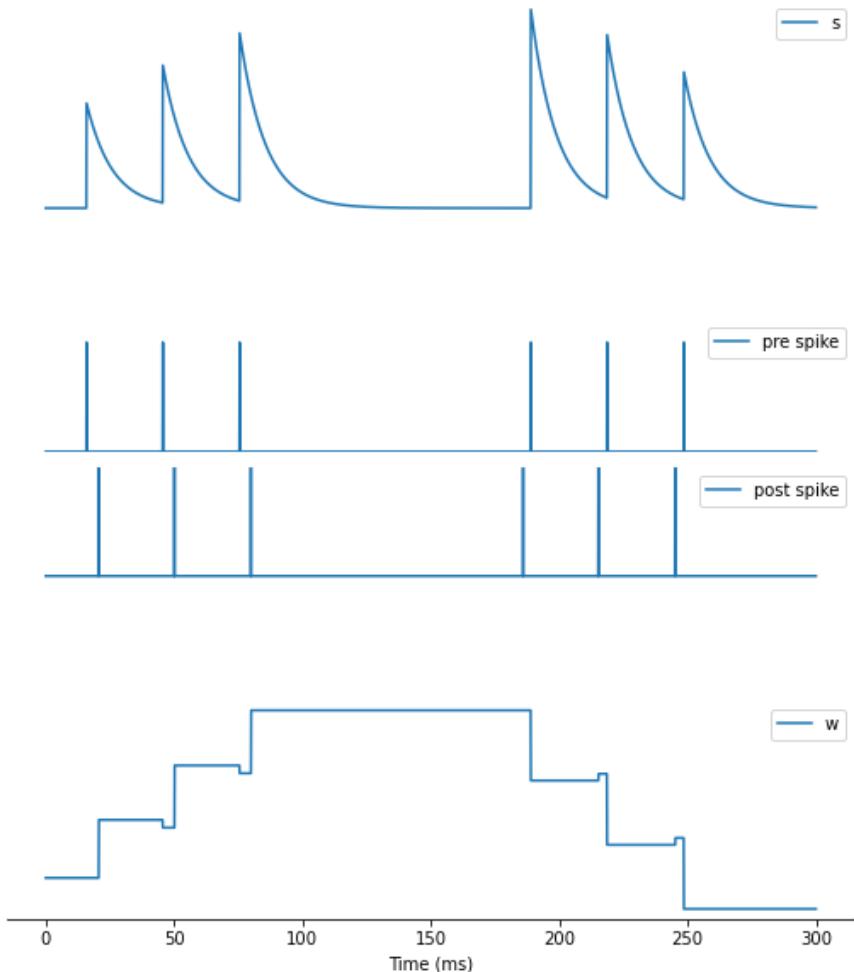
ax1=fig.add_subplot(gs[1, 0])
plt.plot(net.ts, pre.mon.spike[:, 0], label="pre spike")
plt.ylim(0, 2)
hide_spines(ax1)
plt.legend(loc = 'center right')

ax2=fig.add_subplot(gs[2, 0])
plt.plot(net.ts, post.mon.spike[:, 0], label="post spike")
plt.ylim(-1, 1)
hide_spines(ax2)

ax3=fig.add_subplot(gs[3, 0])
plt.plot(net.ts, syn.mon.w[:, 0], label="w")
plt.legend()
# hide spines
plt.yticks([])
ax3.spines['left'].set_visible(False)
ax3.spines['right'].set_visible(False)
ax3.spines['top'].set_visible(False)

plt.xlabel('Time (ms)')
plt.show()
```





The simulation result shows that weights w increase when the presynaptic neuron fire before the postsynaptic neuron (before 150ms); and decrease when the order switched (after 150ms).

Oja's rule

Next, let's look at the rate model based on Hebbian learning. Because Hebbian learning is "fire together, wire together", regardless of the order before and after, spiking time can be ignored, so it can be simplified as a rate-based model. Let's first look at the general form of Hebbian learning. For the j to i connection, r_j, r_i denotes the firing rate of pre- and post-neuron groups, respectively. According to the locality characteristic of Hebbian learning, The change of w_{ij} is affected by w itself and r_j, r_i , we get the following differential equation.

$$\frac{d}{dt}w_{ij} = F(w_{ij}; r_i, r_j)$$

The following formula is obtained by Taylor expansion on the right side of the above formula.

1.1 Biological background

$$\frac{d}{dt}w_{ij} = c_{00}w_{ij} + c_{10}w_{ij}r_j + c_{01}w_{ij}r_i + c_{20}w_{ij}r_j^2 + c_{02}w_{ij}r_i^2 + c_{11}w_{ij}r_i r_j + O(r)$$

The 6th term contains $r_i r_j$, only if c_{11} is not zero can the "fire together" of Hebbian learning be satisfied. For example, the formula of Oja's rule is as follows, which corresponds to the 5th and 6th terms of the above formula.

$$\frac{d}{dt}w_{ij} = \gamma[r_i r_j - w_{ij}r_i^2]$$

γ represents the learning rate.

Now let's see how to use BrainPy to implement Oja's rule.

```

1  class Oja(bp.TwoEndConn):           bp.TwoEndConn class:
2      target_backend = 'numpy'          Learning rule occur between two neuron
3
4      @staticmethod
5      def derivative(w, t, gamma, r_pre, r_post):
6          dwdt = gamma * (r_post * r_pre - r_post * r_post * w)    ]
7          return dwdt
8
9      def __init__(self, pre, post, conn, gamma=.005, w_max=1., w_min=0., **kwargs):
10         # params
11         self.gamma = gamma
12         self.w_max = w_max
13         self.w_min = w_min
14         # no delay in firing rate models
15
16         # conns
17         self.conn = conn(pre.size, post.size)
18         self.conn_mat = conn.requires('conn_mat')
19         self.size = bp.ops.shape(self.conn_mat)    }
20
21         # data
22         self.w = bp.ops.ones(self.size) * 0.05
23
24         self.integral = bp.odeint(f=self.derivative)
25         super(Oja, self).__init__(pre=pre, post=post, **kwargs)
26
27     def update(self, _t):
28         w = self.conn_mat * self.w
29         self.post.r = np.sum(w.T * self.pre.r, axis=1)   Rate model:  $r_i = \sum_j w_{ij} r_j$ 
30
31         # resize to matrix
32         dim = self.size
33         r_post = np.vstack((self.post.r,) * dim[0])    }
34         r_pre = np.vstack((self.pre.r,) * dim[1]).T
35
36         self.w = self.integral(w, _t, self.gamma, r_pre, r_post)
37

```

Use matrix based for rate-based learning rule.
conn_mat:

	post_id								
pre_id	0	1	2	3	4	5	6	7	...
0	0	0	0	1	0	1	0	1	...
1	1	0	1	0	1	0	1	0	...
2	0	1	0	0	0	0	0	0	...
3	0	0	1	0	0	0	0	0	...
4	0	0	0	1	0	0	0	0	...
5	0	0	0	0	1	0	0	0	...
6	0	0	0	0	0	1	0	0	...
7	0	0	0	0	0	0	1	0	...

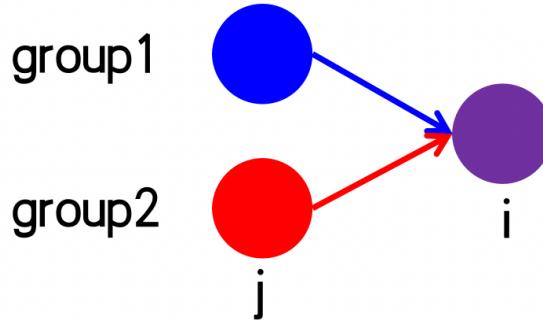
Expand post_r and pre_r (vectors) to be
matrix of the same size as w.

Since Oja's rule is a rate-based model, we need a rate-based neuron model to see this learning rule of two groups of neurons.

```

39  class neu(bp.NeuGroup):
40      target_backend = 'numpy'
41
42      def __init__(self, size, **kwargs):
43          self.r = bp.ops.zeros(size)           We need a firing rate neuron with
44          super(neu, self).__init__(size=size, **kwargs) parameter r
45
46      def update(self, _t):
47          self.r = self.r                  We only need a simple unit to test oja's rule.

```

**Fig. 2-6 Connection of neuron groups.**

We aim to implement the connection as shown in Fig. 2-6. The purple neuron group receives inputs from the blue and red groups. The external input to the post group is exactly the same as the red one, while the blue one is the same at first, but not later.

The simulation code is as follows.

```

50  # create input
51  current1, _ = bp.inputs.constant_current([(2., 20.), (0., 20.)] * 3 +
52  [ (0., 20.), (0., 20.)] * 2)
53  current2, _ = bp.inputs.constant_current([(2., 20.), (0., 20.)] * 5)
54  current3, _ = bp.inputs.constant_current([(2., 20.), (0., 20.)] * 5)
55  current_pre = np.vstack((current1, current2))
56  current_post = np.vstack((current3, current3))

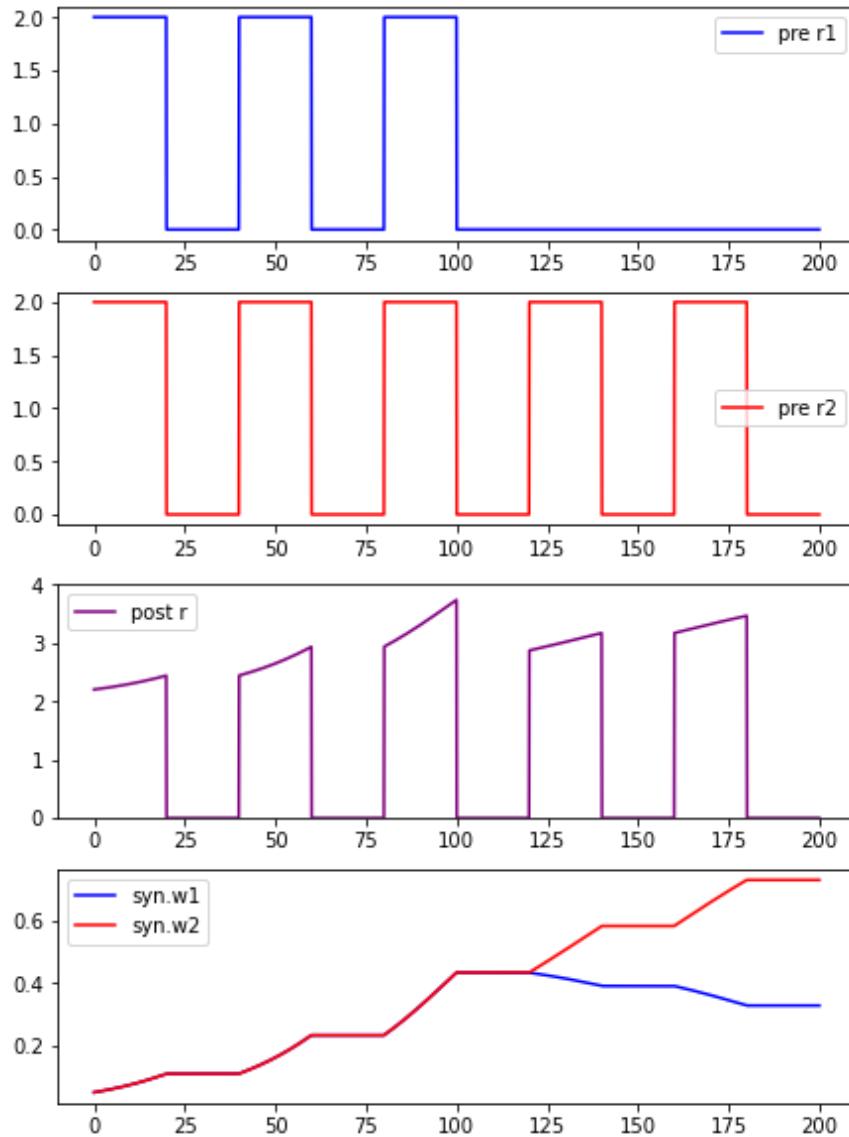
57
58  # simulate
59  neu_pre = neu(2, monitors=['r'])
60  neu_post = neu(2, monitors=['r'])
61  syn = Oja(pre=neu_pre, post=neu_post, conn=bp.connect.All2All(), monitors=['w'])
62  net = bp.Network(neu_pre, syn, neu_post)
63  net.run(duration=200., inputs=[(neu_pre, 'r', current_pre.T, '='),
64  (neu_post, 'r', current_post.T)])
65
66  # plot
67  fig, gs = bp.visualize.get_figure(4, 1, 2, 6)
68
69  fig.add_subplot(gs[0, 0])
70  plt.plot(net.ts, neu_pre.mon.r[:, 0], 'b', label='pre r1')    ----- index[:, 0], group 1
71  plt.legend()
72
73  fig.add_subplot(gs[1, 0])
74  plt.plot(net.ts, neu_pre.mon.r[:, 1], 'r', label='pre r2')    ----- index[:, 1], group 2
75  plt.legend()
76
77  fig.add_subplot(gs[2, 0])
78  plt.plot(net.ts, neu_post.mon.r[:, 0], color='purple', label='post r')
79  plt.ylim([0, 4])
80  plt.legend()
81
82  fig.add_subplot(gs[3, 0])
83  plt.plot(net.ts, syn.mon.w[:, 0, 0], 'b', label='syn.w1')
84  plt.plot(net.ts, syn.mon.w[:, 1, 0], 'r', label='syn.w2')
85  plt.legend()
86  plt.show()

```

} current1: input to pre-group1
current2: input to pre-group2
current3: input to post group,
the same as current2.(fire
together)

} Simulation is like
synapse model.
Here we give inputs
to both pre-group
and post-group.

1.1 Biological background



It can be seen from the results that at the beginning, when the two groups of neurons were given input at the same time, their weights increased simultaneously, and the response of post became stronger and stronger, showing LTP. After 100ms, the blue group is no longer fire together, only the red group still fire together, and only the weights of the red group are increased. The results accord with the "fire together, wire together" of Hebbian learning.

BCM rule

Now let's see other example of Hebbian learning, the BCM rule. It's given by,

$$\frac{d}{dt}w_{ij} = \eta r_i(r_i - r_\theta)r_j$$

1.1 Biological background

where η represents the learning rate, and r_θ represents the threshold of learning (see Fig. 2-7). Fig. 2-7 shows the right side of the formula.

When the firing rate is greater than the threshold, there is LTP, and when the firing rate is lower than the threshold, there is LTD. Therefore, the

selectivity can be achieved by adjusting the threshold r_θ .

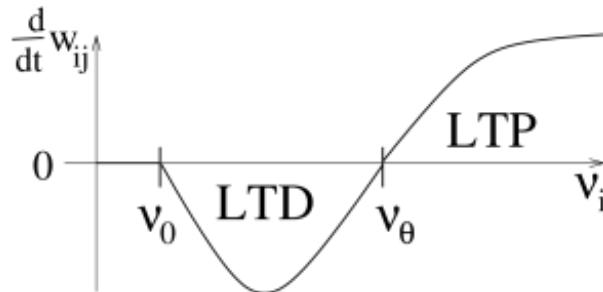


Fig. 2-7 BCM rule (From Gerstner et al., 2014 [1](#))

We will implement the same connections as the previous Oja's rule (Fig. 2-6), with different firing rates. Here the two groups of neurons are alternately firing. Among them, the blue group is always stronger than the red one. We adjust the threshold by setting it as the time average of r_i , that is $r_\theta = f(r_i)$. The code implemented by BrainPy is as follows.

```

1  class BCM(bp.TwoEndConn):
2      target_backend = ['numpy', 'numba']
3
4      @staticmethod
5      def derivative(w, t, lr, r_pre, r_post, r_th):
6          dwdt = lr * r_post * (r_post - r_th) * r_pre
7          return dwdt
8
9      def __init__(self, pre, post, conn, lr=0.005, w_max=1., w_min=0., **kwargs):
10         # parameters
11         self.lr = lr
12         self.w_max = w_max
13         self.w_min = w_min
14         self.dt = bp.backend.get_dt()
15
16         # connections
17         self.conn = conn(pre.size, post.size)
18         self.conn_mat = conn.requires('conn_mat')
19         self.size = bp.ops.shape(self.conn_mat)
20
21         # variables
22         self.w = bp.ops.ones(self.size) * .5
23         self.sum_post_r = np.zeros(post.size[0])
24
25         self.int_w = bp.odeint(f=self.derivative, method='rk4')
26
27         super(BCM, self).__init__(pre=pre, post=post, **kwargs)
28
29     def update(self, _t):
30         # update threshold
31         self.sum_post_r += self.post.r
32         r_th = self.sum_post_r / (_t / self.dt + 1)           }   rθ = Σt dt ri
33                                         self.dt + 1 since self.dt begin with 0
34
35         # resize to matrix
36         dim = self.size
37         r_th = np.vstack((r_th,) * dim[0])
38         r_post = np.vstack((self.post.r,) * dim[0])
39         r_pre = np.vstack((self.pre.r,) * dim[1]).T
40
41         # update w
42         w = self.int_w(self.w * self.conn_mat, _t, self.lr, r_pre, r_post, r_th)
43         self.w = np.clip(w, self.w_min, self.w_max)           ————— Limit w within the boundary
44
45         # output
46         self.post.r = np.sum(w.T * self.pre.r, axis=1)    ————— Rate model: ri = Σj wij rj
```

1.1 Biological background

Then we can run the simulation with the following code.

```
n_post = 1
n_pre = 20

# group selection
group1, duration = bp.inputs.constant_current(([1.5, 1], [0, 0]))
group2, duration = bp.inputs.constant_current(([0, 1], [1., 0]))
group1 = bp.ops.vstack(((group1,)*10))
group2 = bp.ops.vstack(((group2,)*10))
input_r = bp.ops.vstack((group1, group2))

pre = neu(n_pre, monitors=['r'])
post = neu(n_post, monitors=['r'])
bcm = BCM(pre=pre, post=post, conn=bp.connect.All2All(),
           monitors=['w'])

net = bp.Network(pre, bcm, post)
net.run(duration, inputs=(pre, 'r', input_r.T, "="))

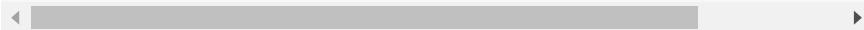
w1 = bp.ops.mean(bcm.mon.w[:, :10, 0], 1)
w2 = bp.ops.mean(bcm.mon.w[:, 10:, 0], 1)

r1 = bp.ops.mean(pre.mon.r[:, :10], 1)
r2 = bp.ops.mean(pre.mon.r[:, 10:], 1)

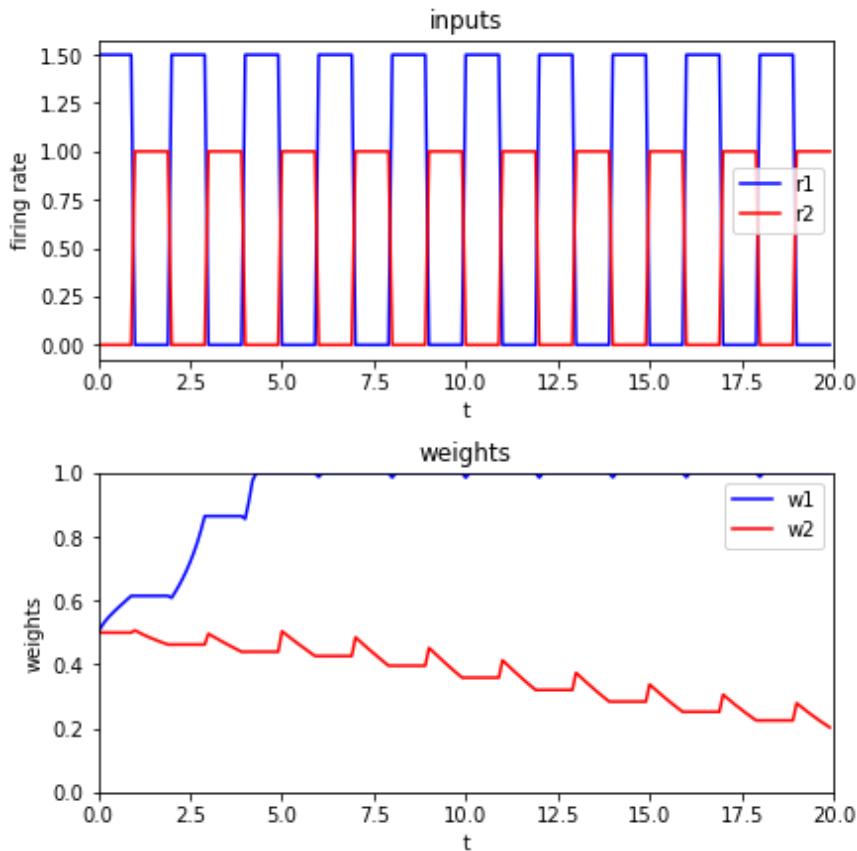
fig, gs = bp.visualize.get_figure(2, 1, 3, 12)
fig.add_subplot(gs[1, 0], xlim=(0, duration), ylim=(0, w_max))
plt.plot(net.ts, w1, 'b', label='w1')
plt.plot(net.ts, w2, 'r', label='w2')
plt.title("weights")
plt.ylabel("weights")
plt.xlabel("t")
plt.legend()

fig.add_subplot(gs[0, 0], xlim=(0, duration))
plt.plot(net.ts, r1, 'b', label='r1')
plt.plot(net.ts, r2, 'r', label='r2')
plt.title("inputs")
plt.ylabel("firing rate")
plt.xlabel("t")
plt.legend()

plt.show()
```



1.1 Biological background



The results show that the blue group with stronger input demonstrating LTP, while the red group with weaker input showing LTD, so the blue group is being chosen.

References

- [1] Gerstner, Wulfram, et al. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.
- [2] Tsodyks, Misha, and Si Wu. "Short-term synaptic plasticity." *Scholarpedia* 8.10 (2013): 3153.
- [3] Bi, Guo-qiang, and Mu-ming Poo. "Synaptic modification by correlated activity: Hebb's postulate revisited." *Annual review of neuroscience* 24.1 (2001): 139-166.

3. Network models

With the neuron and synapse models we have realized with BrainPy, users can now build networks of their own. In this section, we will introduce two main types of network models as examples:

- 1) spiking neural networks that model and compute each neuron or synapse separately;
- 2) firing rate networks that simplify neuron groups in the network as firing rate units and compute each neuron group as one unit.

In this chapter, we will introduce networks belonging to these two types of network models: E-I balanced network, decision making networks and continuous attractor neural network.

3.1 E-I balanced network

3.2 Decision making networks

3.3 Continuous attractor neural network

3.1.1 E/I balanced network

In 1990s, biologists found in experiments that neuron activities in brain cortex show a temporal irregular spiking pattern. This pattern exists widely in brain areas, but researchers knew few about its mechanism or function.

Vreeswijk and Sompolinsky (1996) proposed **E/I balanced network** to explain this irregular spiking pattern. The feature of this network is the strong, random and sparse synapse connections between neurons. Because of this feature and corresponding parameter settings, each neuron in the network will receive great excitatory and inhibitory input from within the network. However, these two types of inputs will cancel each other, and maintain the total internal input at a relatively small order of magnitude, which is only enough to generate action potentials.

The randomness and noise in E/I balanced network give each neuron in the network an internal input which varies with time and space at the order of threshold potential. Therefore, the firing of neurons also has randomness, ensures that E/I balanced network can generate temporal irregular firing pattern spontaneously.

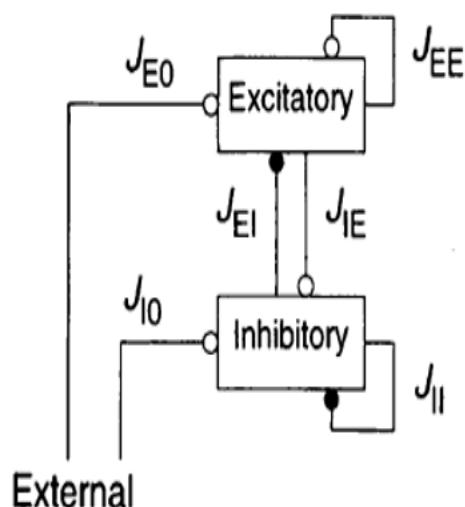


Fig. 3-1 Structure of E/I balanced network (Vreeswijk and Sompolinsky, 1996 [1](#))

Vreeswijk and Sompolinsky also suggested a possible function of this irregular firing pattern: E/I balanced network can respond to the changes of external stimulus quickly.

1.1 Biological background

As shown in Fig. 3-3, when there is no external input, the distribution of neurons' membrane potentials in E/I balanced network follows a relatively uniform random distribution between resting potential V_0 and threshold potential θ .

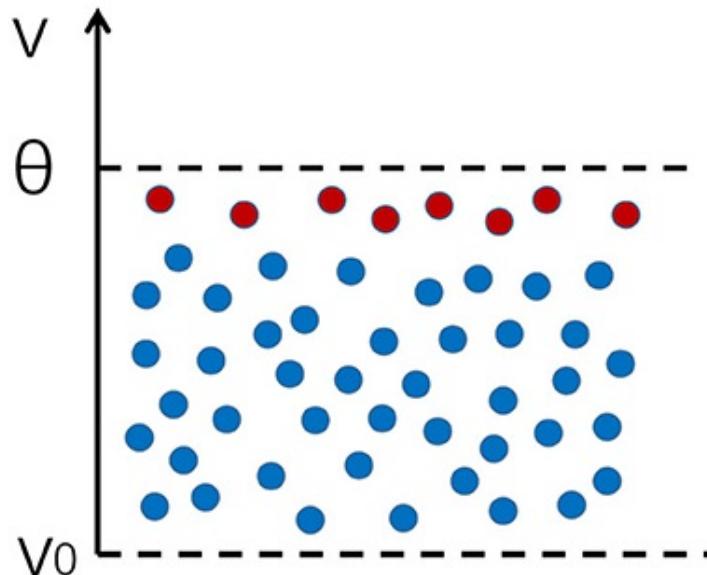


Fig. 3-2 Distribution of neuron membrane potentials in E/I balanced network (Tian et al., 2020²)

When we give the network a small constant external stimulus, those neurons whose membrane potentials fall near the threshold potential will soon meet the threshold, therefore spike rapidly. On the network scale, the firing rate of the network can adjust rapidly once the input changes.

Simulation suggests that the delay of network response to input and the delay of synapses have the same time scale, and both are significantly smaller than the delay of a single neuron from resting potential to generating a spike. So E/I balanced network may provide a fast response mechanism for neural networks.

Fig. 3-1 shows the structure of E/I balanced network:

- 1) Neurons: Neurons are realized with LIF neuron model. The neurons can be divided into excitatory neurons and inhibitory neurons, the ratio of the two types of neurons is $N_E : N_I = 4:1$.
- 2) Synapses: Synapses are realized with exponential synapse model. 4 groups of synapse connections are generated between the two groups of neurons, that is, excitatory-excitatory connection (E2E conn), excitatory-inhibitory connection (E2I conn), inhibitory-excitatory connection (I2E conn) and inhibitory-inhibitory connection (I2I conn). For excitatory or inhibitory synapse connections, we define synapse weights with different signal.

1.1 Biological background

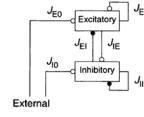
```

9   N_E = 500
10  N_I = 500
11  prob = 0.1
53
54  neu_E = brainmodels.neurons.LIF(N_E, monitors=['spike'])
55  neu_I = brainmodels.neurons.LIF(N_I, monitors=['spike'])
56  neu_E.V = V_rest + np.random.random(N_E) * (V_th - V_rest)
57  neu_I.V = V_rest + np.random.random(N_I) * (V_th - V_rest)
58
59  syn_E2E = brainmodels.synapses.Exponential(
60      pre=neu_E, post=neu_E,
61      conn=bp.connect.FixedProb(prob=prob))
62  syn_E2I = brainmodels.synapses.Exponential(
63      pre=neu_E, post=neu_I,
64      conn=bp.connect.FixedProb(prob=prob))
65  syn_I2E = brainmodels.synapses.Exponential(
66      pre=neu_I, post=neu_E,
67      conn=bp.connect.FixedProb(prob=prob))
68  syn_I2I = brainmodels.synapses.Exponential(
69      pre=neu_I, post=neu_I,
70      conn=bp.connect.FixedProb(prob=prob))
71
72  JE = 1 / np.sqrt(prob * N_E)
73  JI = 1 / np.sqrt(prob * N_I)
74  syn_E2E.w = JE
75  syn_E2I.w = JE
76  syn_I2E.w = -JI
77  syn_I2I.w = -JI
    
```

} Import LIF neuron model from brainmodels.
Build excitatory neuron group and inhibitory neuron group.

} Import exponential synapse model from brainmodels.
Build E2E, E2I, I2E, I2I synapse connections between neuron groups.

} weight of connection = $J_E = J_I = \frac{1}{\sqrt{p N_E}} = \frac{1}{\sqrt{p N_I}}$



3) Inputs: All neurons in the network receive a constant external input current.

```

79  net = bp.Network(neu_E, neu_I,
80                     syn_E2E, syn_E2I,
81                     syn_I2E, syn_I2I)
82  net.run(500., inputs=[(neu_E, 'input', 3.), (neu_I, 'input', 3.)], report=True)
83
    
```

} Simulate the network for 500ms,
give all neurons constant external
inputs with amplitude 3.

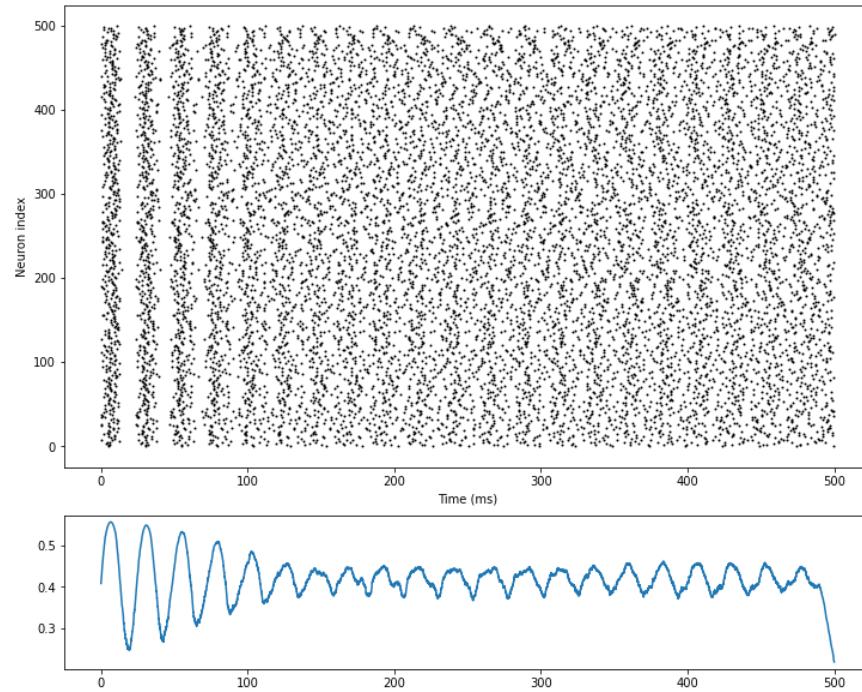
See above section 1 and 2 for definition of LIF neuron and exponential synapse. After simulation, we visualize the raster plot and firing rate-t plot of E/I balanced network. the network firing rate changes from strong synchronization to irregular fluctuation.

```

84  fig, gs = bp.visualize.get_figure(4, 1, 2, 10)
85  fig.add_subplot(gs[3, 0])
86  bp.visualization.raster_plot(net.ts, neu_E.mon.spike)
87
88  fig.add_subplot(gs[3, 0])
89  rate = bp.measure.firing_rate(neu_E.mon.spike, 5.)
90  plt.plot(net.ts, rate)
91  plt.show()
92
    
```

} Paint the raster plot and firing_rate-t plot of
excitatory neurons in E/I balanced network.

1.1 Biological background



References

- [1] Van Vreeswijk, Carl, and Haim Sompolinsky. "Chaos in neuronal networks with balanced excitatory and inhibitory activity." *Science* 274.5293 (1996): 1724-1726.
- [2] Tian, Gengshuo, et al. "Excitation-Inhibition Balanced Neural Networks for Fast Signal Detection." *Frontiers in Computational Neuroscience* 14 (2020): 79.

3.2 Decision Making Network

The modeling of computational neuroscience networks can correspond to specific physiological tasks.

For example, in the visual motion discrimination task (Roitman and Shadlen, 2002), rhesus watch a video in which random dots move towards left or right with definite coherence. Rhesus are required to choose the direction that most dots move to and give their answer by saccade. At the meantime, researchers record the activity of their LIP neurons by implanted electrode.

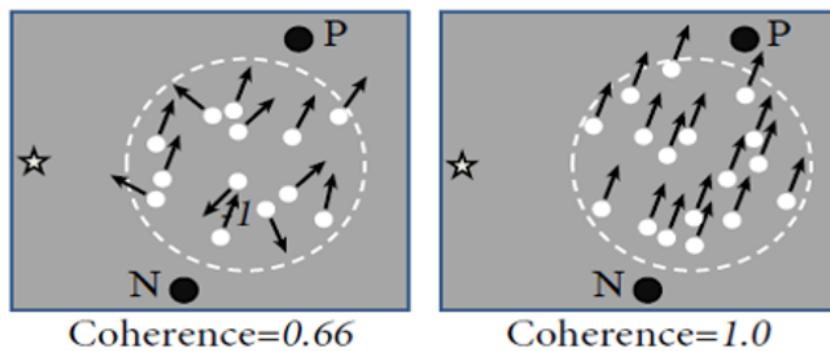


Fig. 3-3 Experimental Diagram (Gerstner et al., 2014¹)

Wang et al. (2002; 2006) proposed the spiking neural network and firing rate neural network introduced in this section, aimed to model the activity of rhesus LIP neurons during decision making period in the visual motion discrimination task.

3.2.1 Spiking Neural Network

Wang (2002) first proposed the decision making spiking network. As shown in Fig. 3-4, this network is based on E/I balanced network. The ratio of excitatory neurons and inhibitory neurons is $N_E : N_I = 4 : 1$, and parameters are adjusted to maintain the balanced state.

To accomplish the decision making task, among the excitatory neuron group, two selective subgroup A and B are chosen, both with a size of $N_A = N_B = 0.15N_E$. These two subgroups are marked as A and B in Fig. 3-4, and we call other excitatory neurons as non-selective neurons,

$$N_{non} = (1 - 2 * 0.15)N_E.$$

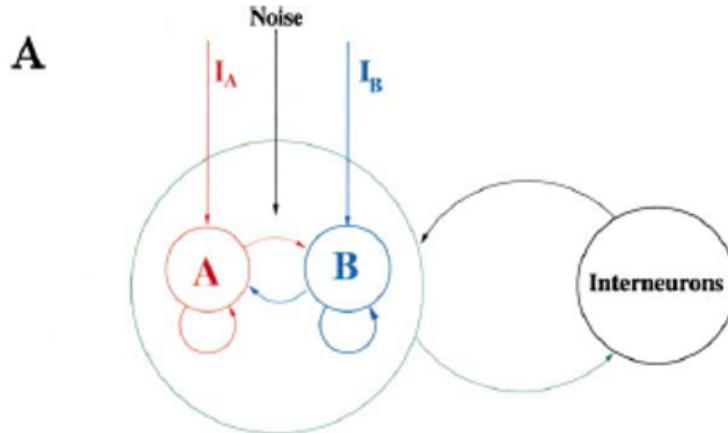
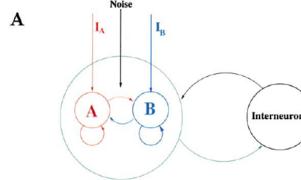


Fig. 3-4 structure of decision making network (Wang, 2002²)

```

279 # def neurons
280 # def E neurons/pyramid neurons
281 neu_A = LIF(N_A, monitors=['spike', 'input', 'V'])
282 neu_A.V_rest = V_rest_E
283 neu_A.V_reset = V_reset_E
284 neu_A.V_th = V_th_E
285 neu_A.R = R_E
286 neu_A.tau = tau_E
287 neu_A.t_refractory = t_refractory_E
288 neu_A.V = bp.ops.ones(N_A) * V_rest_E
289
290 neu_B = LIF(N_B, monitors=['spike', 'input', 'V'])
291 neu_B.V_rest = V_rest_E
292 neu_B.V_reset = V_reset_E
293 neu_B.V_th = V_th_E
294 neu_B.R = R_E
295 neu_B.tau = tau_E
296 neu_B.t_refractory = t_refractory_E
297 neu_B.V = bp.ops.ones(N_B) * V_rest_E
298
299 neu_non = LIF(N_non, monitors=['spike', 'input', 'V'])
300 neu_non.V_rest = V_rest_E
301 neu_non.V_reset = V_reset_E
302 neu_non.V_th = V_th_E
303 neu_non.R = R_E
304 neu_non.tau = tau_E
305 neu_non.t_refractory = t_refractory_E
306 neu_non.V = bp.ops.ones(N_non) * V_rest_E
307
308 # def I neurons/interneurons
309 neu_I = LIF(N_I, monitors=['input', 'V'])
310 neu_I.V_rest = V_rest_I
311 neu_I.V_reset = V_reset_I
312 neu_I.V_th = V_th_I
313 neu_I.R = R_I
314 neu_I.tau = tau_I
315 neu_I.t_refractory = t_refractory_I
316 neu_I.V = bp.ops.ones(N_I) * V_rest_I
317

```



As it is in E/I balanced network, 4 groups of synapses ---- E2E connection, E2I connection, I2E connection and I2I connection ---- are built in decision making network. Excitatory connections are realized with AMPA synapse, inhibitory connections are realized with GABAa synapse.

Decision making network needs to make a decision among the two choice, i.e., among the two subgroups A and B in this task. To achieve this, network must discriminate between these two groups. Excitatory

1.1 Biological background

neurons in the same subgroup should self-activate, and inhibit neurons in another selective subgroup.

Therefore, E2E connections are structured in the network. As shown in Sheet 3-1, $w+ > 1 > w-$. In this way, a relative activation is established within the subgroups by stronger excitatory synapse connections, and relative inhibition is established between two subgroups or between selective and non-selective subgroups by weaker excitatory synapse connections.

Sheet 3-1 Weight of synapse connections between E-neurons

w	A	B	non
A	w+	w-	1.
B	w-	w+	1.
non	w-	w-	1.

```

249 # set syn weights (only used in recurrent E connections)
250 w_pos = 1.7
251 w_neg = 1. - f * (w_pos - 1.) / (1. - f) } w+ = 1.7
252 print(f"the structured weight is: w_pos = {w_pos}, w_neg = {w_neg}")
253 # inside select group: w = w+
254 # between group / from non-select group to select group: w = w-
255 # A2A B2B w+, A2B B2A w-, non2A non2B w-
256 weight = np.ones((N_E, N_E), dtype=np.float)
257 for i in range(N_A):
258     weight[i, 0:N_A] = w_pos
259     weight[i, N_A:N_A + N_B] = w_neg
260 for i in range(N_A, N_A + N_B):
261     weight[i, N_A:N_A + N_B] = w_pos
262     weight[i, 0:N_A] = w_neg
263 for i in range(N_A + N_B, N_E):
264     weight[i, 0:N_A + N_B] = w_neg
265 print(f"Check constraints: Weight sum {weight.sum(axis=0)[0]} \
266      should be equal to N_E = {N_E}")

```

Define structured weights for E2E connns.
Strong connection inside selective group,
weak connection for other synapses.

w	A	B	non
A	w+	w-	1.
B	w-	w+	1.
non	w-	w-	1.

We give two types of external inputs to the decision making network:

- 1) Background inputs from other brain areas without specific meaning.
Represented as high frequency Poisson input mediated by AMPA synapse.

1.1 Biological background

```

488 # def background poisson input
489 class PoissonInput(bp.NeuGroup):
490     target_backend = 'general'
491
492     def __init__(self, size, freqs, dt, **kwargs):
493         self.freqs = freqs
494         self.dt = dt
495
496         self.spike = bp.ops.zeros(size, dtype=bool)
497
498     super(PoissonInput, self).__init__(size=size, **kwargs)
499
500     def update(self, _t):
501         self.spike = np.random.random(self.size) \
502             < self.freqs * self.dt / 1000. ] Generate spikes with a given Poisson frequency.
503
504 # poisson_freq = 2400Hz
505 neu_poisson_A = PoissonInput(N_A, freqs=poisson_freq, dt=dt)
506 neu_poisson_B = PoissonInput(N_B, freqs=poisson_freq, dt=dt)
507 neu_poisson_non = PoissonInput(N_non, freqs=poisson_freq, dt=dt)
508 neu_poisson_I = PoissonInput(N_I, freqs=poisson_freq, dt=dt) ] Define neurons that generate Poisson background inputs for every neuron in the network.
509
510 syn_back2A_AMPA = AMPA(pre=neu_poisson_A, post=neu_A,
511                         conn=bp.connect.OneToOne())
512 syn_back2B_AMPA = AMPA(pre=neu_poisson_B, post=neu_B,
513                         conn=bp.connect.OneToOne())
514 syn_back2non_AMPA = AMPA(pre=neu_poisson_non, post=neu_non,
515                         conn=bp.connect.OneToOne())
516
517 syn_back2I_AMPA = AMPA(pre=neu_poisson_I, post=neu_I,
518                         conn=bp.connect.OneToOne()) ] Define AMPA synapses that send the Poisson background inputs to the neurons.
519

```

2) Stimulus inputs from outside the brain, which are given only to the two selective subgroup A and B. Represented as lower frequency Poisson input mediated by AMPA synapse.

The frequency of Poisson input given to A and B subgroup have a certain difference, simulate the difference in the number of dots moving to left and right in physiological experiments, induce the network to make a decision among these two subgroups.

$$\rho_A = \rho_B = \mu_0/100$$

$$\mu_A = \mu_0 + \rho_A * c$$

$$\mu_B = \mu_0 + \rho_B * c$$

Every 50ms, the Poisson frequencies f_x change once, follows a Gaussian distribution defined by mean μ_x and variance δ^2 .

$$f_A \sim N(\mu_A, \delta^2)$$

$$f_B \sim N(\mu_B, \delta^2)$$

1.1 Biological background

```

529 ## def stimulus input
530 # Note: inputs only given to A and B group
531 mu_0 = 40.
532 coherence = 25.6
533 rou_A = mu_0 / 100.
534 rou_B = mu_0 / 100.
535 mu_A = mu_0 + rou_A * coherence
536 mu_B = mu_0 - rou_B * coherence
537 print(f"coherence = {coherence}, mu_A = {mu_A}, mu_B = {mu_B}")
538
539 class PoissonStim(bp.NeuGroup):
540     """
541         from time <t_start> to <t_end> during the simulation, the neuron
542         generates a poisson spike with frequency <self.freq>. however,
543         the value of <self.freq> changes every <t_interval> ms and obey
544         a Gaussian distribution defined by <mean_freq> and <var_freq>.
545     """
546     target_backend = 'general'
547
548     def __init__(self, size, dt=0., t_start=0., t_end=0., t_interval=0.,
549                  mean_freq=0., var_freq=20., **kwargs):
550         self.dt = dt
551         self.stim_start_t = t_start
552         self.stim_end_t = t_end
553         self.stim_change_freq_interval = t_interval
554         self.mean_freq = mean_freq
555         self.var_freq = var_freq
556
557         self.freq = 0.
558         self.t_last_change_freq = -1e7
559         self.spike = bp.ops.zeros(size, dtype=bool)
560
561     super(PoissonStim, self).__init__(size=size, **kwargs)
562
563     def update(self, _t):
564         if self.stim_start_t < _t < self.stim_end_t:
565             if self.stim_change_freq_interval <= _t - self.t_last_change_freq:
566                 self.freq = np.random.normal(self.mean_freq, self.var_freq)
567                 self.freq = max(self.freq, 0)
568                 self.t_last_change_freq = _t
569             self.spike = np.random.random(self.size) < (self.freq * self.dt / 1000)
570         else:
571             self.freq = 0.
572             self.spike[:] = False
573
574
575     neu_input2A = PoissonStim(N_A, dt=dt, t_start=pre_period,
576                               t_end=pre_period + stim_period,
577                               t_interval=50., mean_freq=mu_A, var_freq=10.,
578                               monitors=['freq'])
579     neu_input2B = PoissonStim(N_B, dt=dt, t_start=pre_period,
580                               t_end=pre_period + stim_period,
581                               t_interval=50., mean_freq=mu_B, var_freq=10.,
582                               monitors=['freq'])
583
584     syn_input2A_AMPA = AMPA(pre=neu_input2A, post=neu_A,
585                             conn=bp.connect.OneToOne())
586
587     syn_input2B_AMPA = AMPA(pre=neu_input2B, post=neu_B,
588                             conn=bp.connect.OneToOne())
589

```

Compute basic Poisson frequencies for stimuli given to neuron group A and B.
 $\mu_A = \mu_0 + \rho_A * c$
 $\mu_B = \mu_0 - \rho_B * c$

Save model parameters and variables.

Stimulus are only generated in simulation between time period [stim_start_t, stim_end_t].

Generate new Poisson frequency obeys Gaussian distribution for this neuron every 50ms.
 $f_A \sim N(\mu_A, \delta^2)$
 $f_B \sim N(\mu_B, \delta^2)$

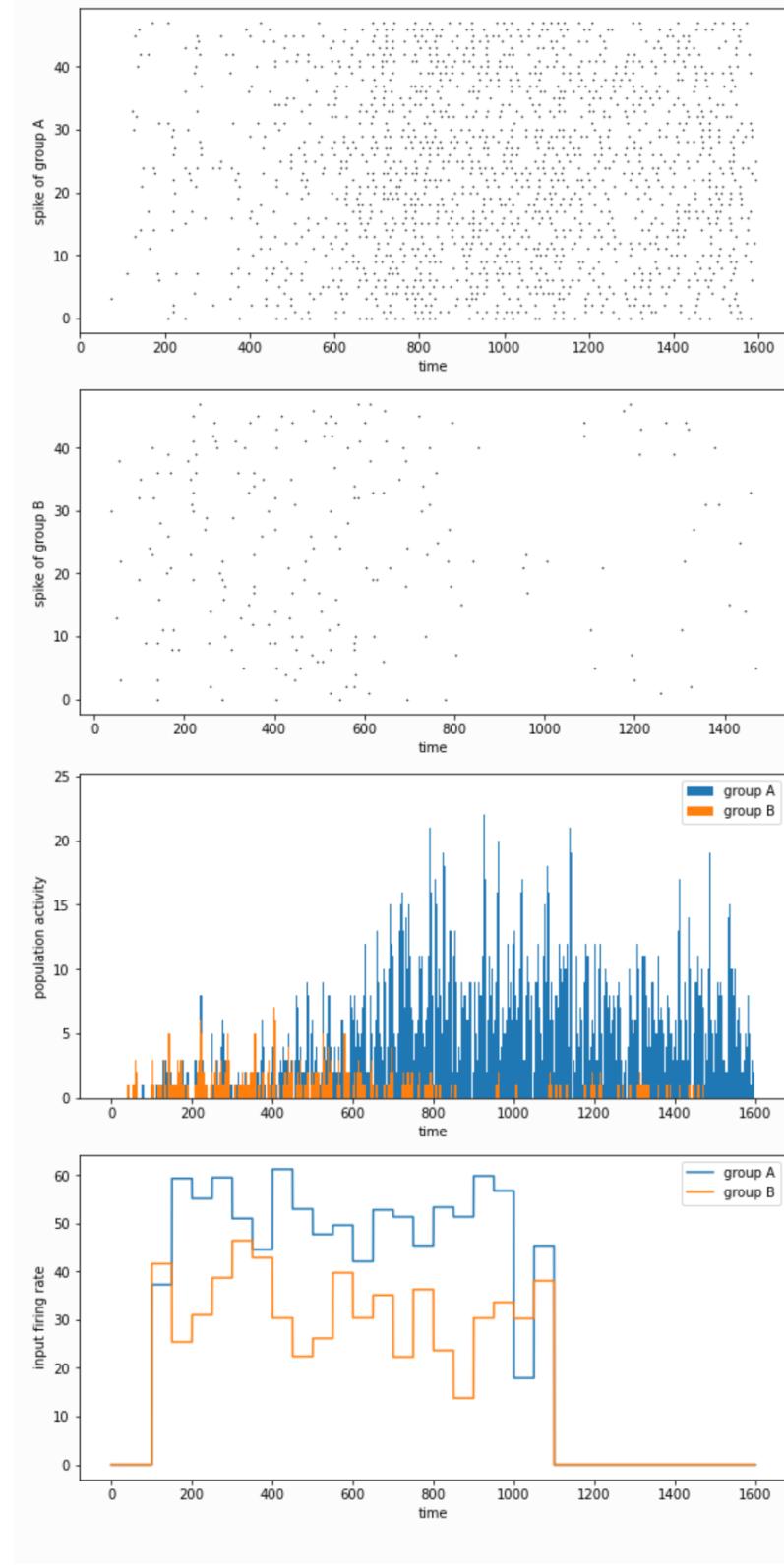
Generate Poisson inputs with the Poisson frequency of this neuron.

Define neurons that generate Poisson stimulus inputs for every neuron in the network.

Define AMPA synapses that send the Poisson stimulus inputs to the neurons.

During the simulation, subgroup A receives a larger stimulus input than B, after a definite delay period, the activity of group A is significantly higher than group B, which means, the network chooses the right direction.

1.1 Biological background



3.2.2 Firing rate model

1.1 Biological background

In addition to spiking models, BrainPy can also implement Firing rate models. Let's first look at the implementation of a simplified version of the decision model. The model was simplified by the researcher (Wong & Wang, 2006) through a series of means such as mean field approach.

In the end, there are only two variables, S_1 and S_2 , which respectively represent the state of two neuron groups and correspond to two options.

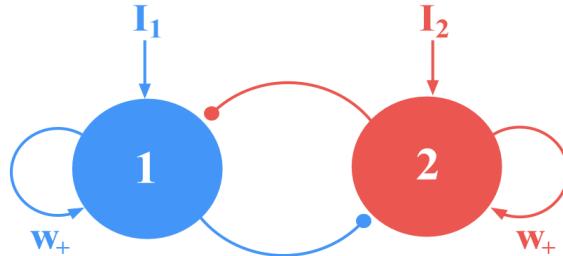


Fig. 3-5 Reduced decision model. (From Wong & Wang, 2006³)

The model is given by,

$$\frac{dS_1}{dt} = -\frac{S_1}{\tau} + (1 - S_1)\gamma r_1$$

$$\frac{dS_2}{dt} = -\frac{S_2}{\tau} + (1 - S_2)\gamma r_2$$

where r_1 and r_2 is the firing rate of two neuron groups, which is given by the input-output function,

$$r_i = f(I_{syn,i})$$

$$f(I) = \frac{aI - b}{1 - \exp[-d(aI - b)]}$$

where $I_{syn,i}$ is given by the model structure (Fig. 3-5),

$$I_{syn,1} = J_{11}S_1 - J_{12}S_2 + I_0 + I_1$$

$$I_{syn,2} = J_{22}S_2 - J_{21}S_1 + I_0 + I_2$$

where I_0 is the background current, and the external inputs I_1, I_2 are determined by the total input strength μ_0 and a coherence c' . The higher the coherence, the more definite S_1 is the correct answer, while the lower the coherence, the more random it is. The formula is as follows:

$$I_1 = J_{A, ext}\mu_0\left(1 + \frac{c'}{100\%}\right)$$

1.1 Biological background

$$I_2 = J_{A,\text{ext}} \mu_0 \left(1 - \frac{c'}{100\%}\right)$$

The code implementation is as follows: we can create a neuron group class, and use S_1 and S_2 to store the two states of the neuron group. The dynamics of the model can be implemented by a `derivative` function for dynamics analysis.

Then we can define a function to perform phase plane analysis.

1.1 Biological background

```

62 def phase_analyze(I, coh):
63     decision = Decision(1, coh=coh)
64
65     phase = bp.analysis.PhasePlane(decision.integral,           Functions defining the differential equations
66                                     target_vars=OrderedDict(s2=[0., 1.],      (from decision model)
67                                     s1=[0., 1.]),                                Variables to be showed
68                                     fixed_vars=None,                            in phase plane.
69                                     pars_update=dict(I=I, coh=coh,
70                                         J_ext=.00117, J_rec=.3725,
71                                         J_inh=.1137, I_0=.3297,
72                                         a=270., b=108., d=0.154,
73                                         tau_s=.06, gamma=0.641),
74                                     numerical_resolution=.001,
75                                     options={'escape_sympy_solver': True})
76
77     phase.plot_nullcline()
78     phase.plot_fixed_point()
79     phase.plot_vector_field(show=True)

```

Specify values of the parameters.

Since the differential equations in the model are very complicated, here we use numerical solutions, skip the analytical solutions provided by sympy, and set the numerical resolution.

Let's first look at the case when there is no external input. At this time,

$$\mu_0 = 0.$$

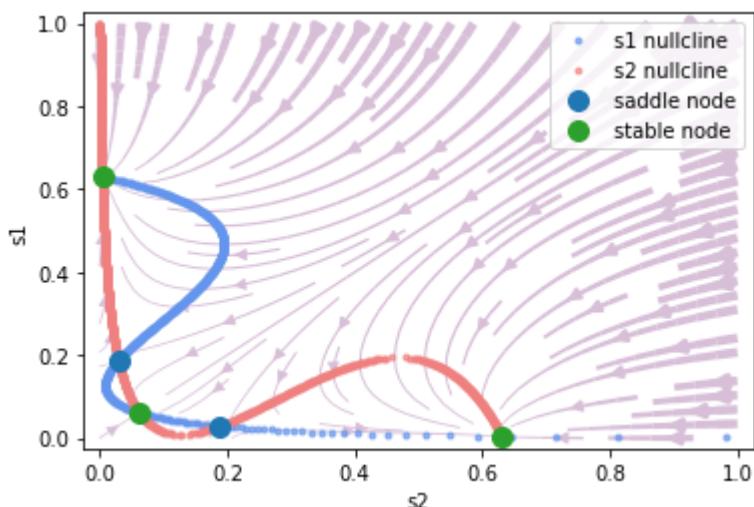
```
phase_analyze(I=0., coh=0.)
```

Output:

```

plot nullcline ...
plot fixed point ...
Fixed point #1 at s2=0.06176109215560733, s1=0.061761097896
Fixed point #2 at s2=0.029354239100062428, s1=0.18815448592
Fixed point #3 at s2=0.0042468423702408655, s1=0.6303045696
Fixed point #4 at s2=0.6303045696241589, s1=0.0042468423702
Fixed point #5 at s2=0.18815439944520335, s1=0.029354240536
plot vector field ...

```



It can be seen that it is very convenient to use BrainPy for dynamics analysis. The vector field and fixed point indicate which option will fall in the end under different initial values.

1.1 Biological background

Here, the x-axis is S_2 which represents choice 2, and the y-axis is S_1 , which represents choice 1. As you can see, the upper-left fixed point represents choice 1, the lower-right fixed point represents choice 2, and the lower-left fixed point represents no choice.

Now let's see which option will eventually fall under different initial values with different coherence, and we fix the external input strength to 30.

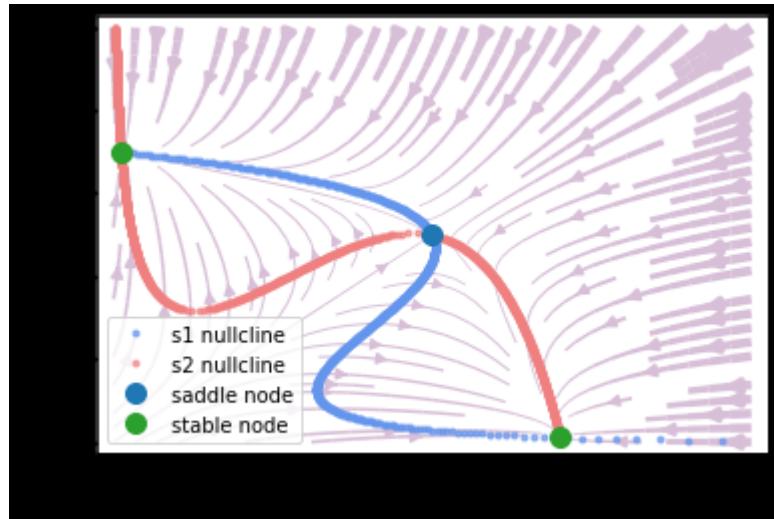
Now let's look at the phase plane under different coherences when we fix the external input strength to 30.

```
# coherence = 0%
print("coherence = 0%")
phase_analyze(I=30., coh=0.)

# coherence = 51.2%
print("coherence = 51.2%")
phase_analyze(I=30., coh=0.512)

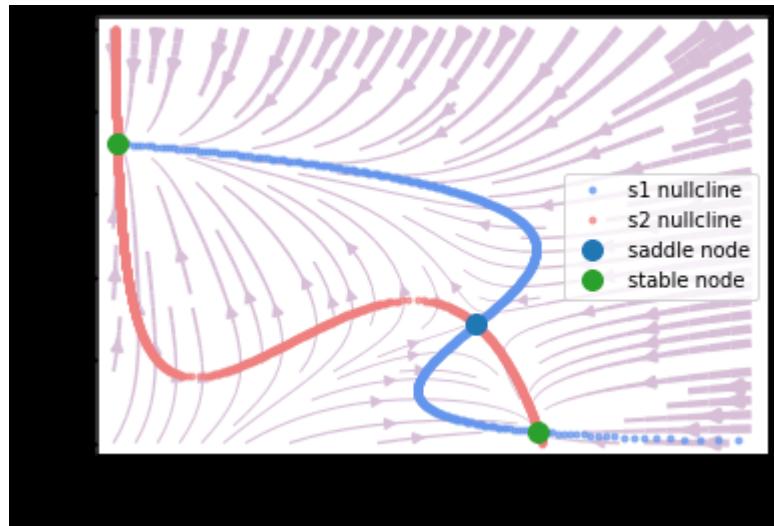
# coherence = 100%
print("coherence = 100%")
phase_analyze(I=30., coh=1.)
```

```
coherence = 0%
plot nullcline ...
plot fixed point ...
Fixed point #1 at s2=0.6993504413889349, s1=0.0116220495267
Fixed point #2 at s2=0.49867489858358865, s1=0.49867489858358865
Fixed point #3 at s2=0.011622051540013889, s1=0.699350435555
plot vector field ...
```

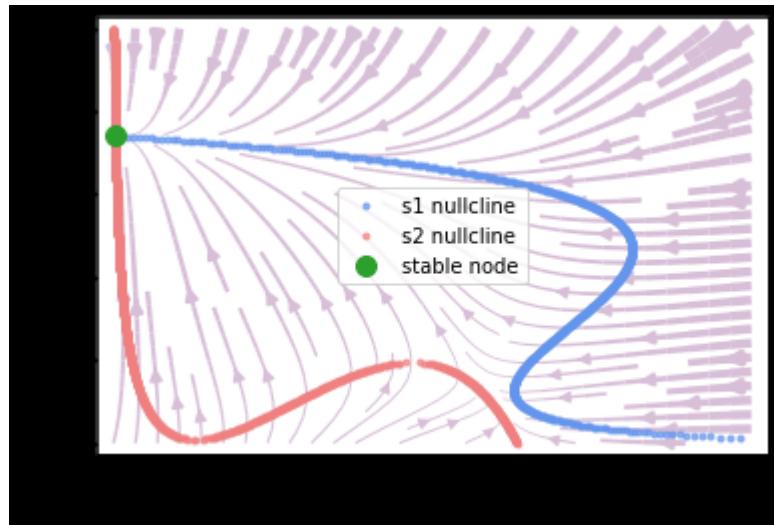


1.1 Biological background

```
coherence = 51.2%
plot nullcline ...
plot fixed point ...
Fixed point #1 at s2=0.5673124813731691, s1=0.2864701069327
Fixed point #2 at s2=0.6655747347157656, s1=0.0278352795659
Fixed point #3 at s2=0.005397687847426814, s1=0.72314535203
plot vector field ...
```



```
coherence = 100%
plot nullcline ...
plot fixed point ...
Fixed point #1 at s2=0.0026865954387078755, s1=0.7410985604
plot vector field ...
```



References

1.1 Biological background

- [1] Gerstner, Wulfram, et al. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.
- [2] Wang, Xiao-Jing. "Probabilistic decision making by slow reverberation in cortical circuits." *Neuron* 36.5 (2002): 955-968.
- [3] Wong, Kong-Fatt, and Xiao-Jing Wang. "A recurrent network mechanism of time integration in perceptual decisions." *Journal of Neuroscience* 26.4 (2006): 1314-1328.

3.3 CANN

Let's see another example of firing rate model, a continuous attractor neural network (CANN)². Fig. 3-6 demonstrates the structure of one-dimensional CANN.

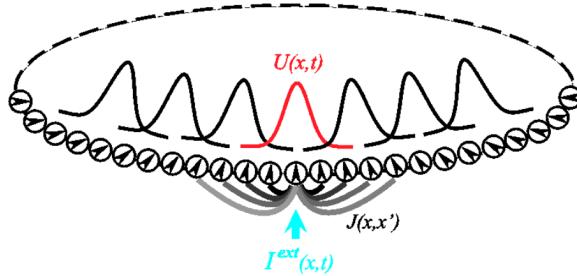


Fig. 3-6 Structure of CANN. (From Wu et al., 2008¹)

We denote (x) as the parameter space site of the neuron group, and the dynamics of the total synaptic input of neuron group (x) $u(x)$ is given by:

$$\tau \frac{du(x,t)}{dt} = -u(x,t) + \rho \int dx' J(x,x') r(x',t) + I_{ext}$$

Where $r(x',t)$ is the firing rate of the neuron group (x') , which is given by:

$$r(x,t) = \frac{u(x,t)^2}{1 + k\rho \int dx' u(x',t)^2}$$

The intensity of excitatory connection between (x) and (x') $J(x,x')$ is given by a Gaussian function:

$$J(x,x') = \frac{1}{\sqrt{2\pi}a} \exp\left(-\frac{|x-x'|^2}{2a^2}\right)$$

The external input I_{ext} is related to position $z(t)$:

$$I_{ext} = A \exp\left[-\frac{|x-z(t)|^2}{4a^2}\right]$$

While implementing with BrainPy, we create a class of `CANN1D` by inheriting `bp.NeuGroup`.

1.1 Biological background

```

1 import brainpy as bp
2 import numpy as np
3 bp.backend.set(backend='numpy', dt=0.1)
4
5 class CANN1D(bp.NeuGroup):
6     target_backend = ['numpy', 'numba']
7
8     def __init__(self, num, tau=1., k=8.1, a=0.5, A=10., J0=4.,
9                  z_min=-np.pi, z_max=np.pi, **kwargs):
10        # parameters
11        self.tau = tau # The synaptic time constant
12        self.k = k # Degree of the rescaled inhibition
13        self.a = a # Half-width of the range of excitatory connections
14        self.A = A # Magnitude of the external input
15        self.J0 = J0 # maximum connection value
16
17        # feature space
18        self.z_min = z_min
19        self.z_max = z_max
20        self.z_range = z_max - z_min
21        self.x = np.linspace(z_min, z_max, num) # The encoded feature values
22
23        # variables
24        self.u = np.zeros(num)
25        self.input = np.zeros(num)
26
27        # The connection matrix
28        self.conn_mat = self.make_conn(self.x)
29
30        super(CANN1D, self).__init__(size=num, **kwargs)
31
32        self.rho = num / self.z_range # The neural density
33        self.dx = self.z_range / num # The stimulus density
34

```

z_{range} denotes the range of x .

$\rho \int dx'$

Then we define the functions.

```

35     @staticmethod
36     @bp.odeint(method='rk4', dt=0.05)
37     def int_u(u, t, conn, k, tau, Iext):
38         r1 = np.square(u)
39         r2 = 1.0 + k * np.sum(r1)
40         r = r1 / r2
41         Irec = np.dot(conn, r)
42         du = (-u + Irec + Iext) / tau
43         return du
44
45     def dist(self, d):
46         d = np.remainder(d, self.z_range)
47         d = np.where(d > 0.5 * self.z_range, d - self.z_range, d)
48         return d
49
50     def make_conn(self, x):
51         assert np.ndim(x) == 1
52         x_left = np.reshape(x, (-1, 1))
53         x_right = np.repeat(x.reshape((1, -1)), len(x), axis=0)
54         d = self.dist(x_left - x_right)
55         Jxx = self.J0 * np.exp(-0.5 * np.square(d / self.a)) / (
56             np.sqrt(2 * np.pi) * self.a)
57         return Jxx
58
59     def get_stimulus_by_pos(self, pos):
60         return self.A * np.exp(-0.25 * np.square(self.dist(self.x -
61                                         pos) / self.a))
62
63     def update(self, _t):
64         self.u = self.int_u(self.u, _t, self.conn_mat, self.k, self.tau,
65                             self.input)
66         self.input[:] = 0.

```

Distances on the ring:
The farthest distance is half of z_{range} , so d should not exceed $0.5 * z_{\text{range}}$

Compute the distance matrix ($d=|x-x'|$) for all positions

$J(x, x') = \frac{\exp(-0.5(\frac{d}{a})^2)}{\sqrt{2\pi}a}$

$I_{\text{ext}} = A \exp\left(\frac{|x-z(t)|^2}{4a^2}\right)$

Where the functions `dist` and `make_conn` are designed to get the connection strength J between each of the two neuron groups. In the `make_conn` function, we first calculate the distance matrix between each of the two x . Because neurons are arranged in rings, the value of x is between $-\pi$ and π , so the range of $|x - x'|$ is 2π , and $-\pi$ and π are the

1.1 Biological background

same points (the actual maximum value is π , that is, half of `z_range` , the distance exceeded needs to be subtracted from a `z_range`). We use the `dist` function to handle the distance on the ring.

The `get_stimulus_by_pos` function processes external inputs based on position `pos` , which allows users to get input current by setting target positions. For example, in a simple population coding, we give an external input of `pos=0` , and we run in this way:

```
cann = CANN1D(num=512, k=0.1, monitors=['u'])

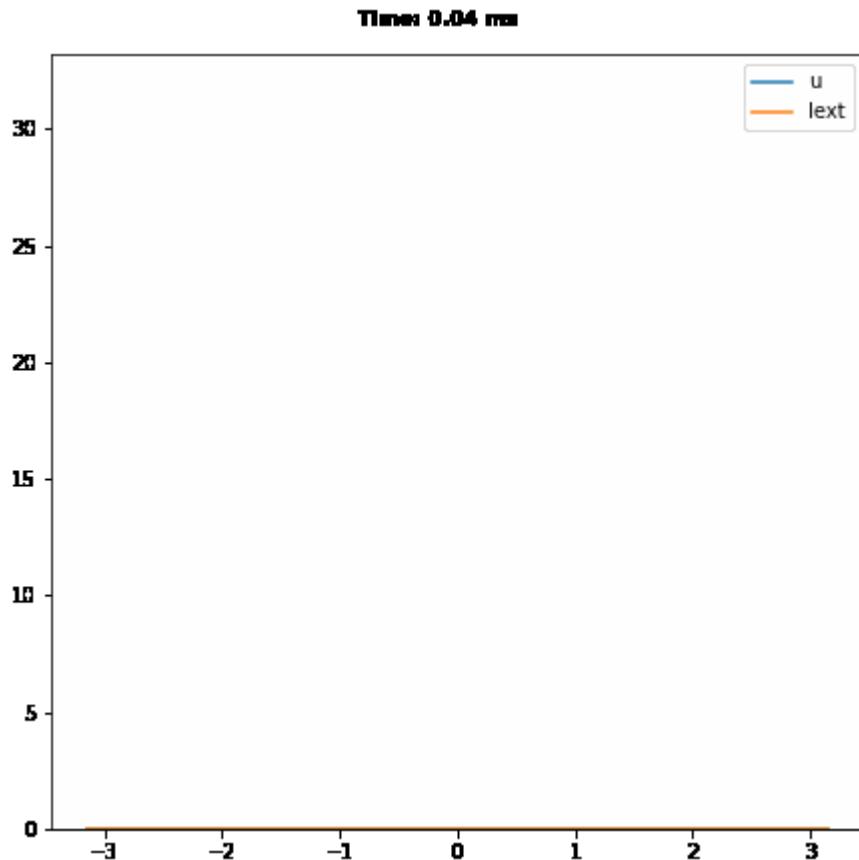
I1 = cann.get_stimulus_by_pos(0.)
Iext, duration = bp.inputs.constant_current([(0., 1.), (I1,
cann.run(duration=duration, inputs=('input', Iext))
```

Then lets plot an animation by calling the `bp.visualize.animate_1D` function.

```
# define function
def plot_animate(frame_step=5, frame_delay=50):
    bp.visualize.animate_1D(dynamical_vars=[{'ys': cann.mor
                                              'legend': 'u'}
                                              'xs': cann.x,
                                              frame_step=frame_step, frame_de
                                              show=True)

# call the function
plot_animate(frame_step=1, frame_delay=100)
```

1.1 Biological background



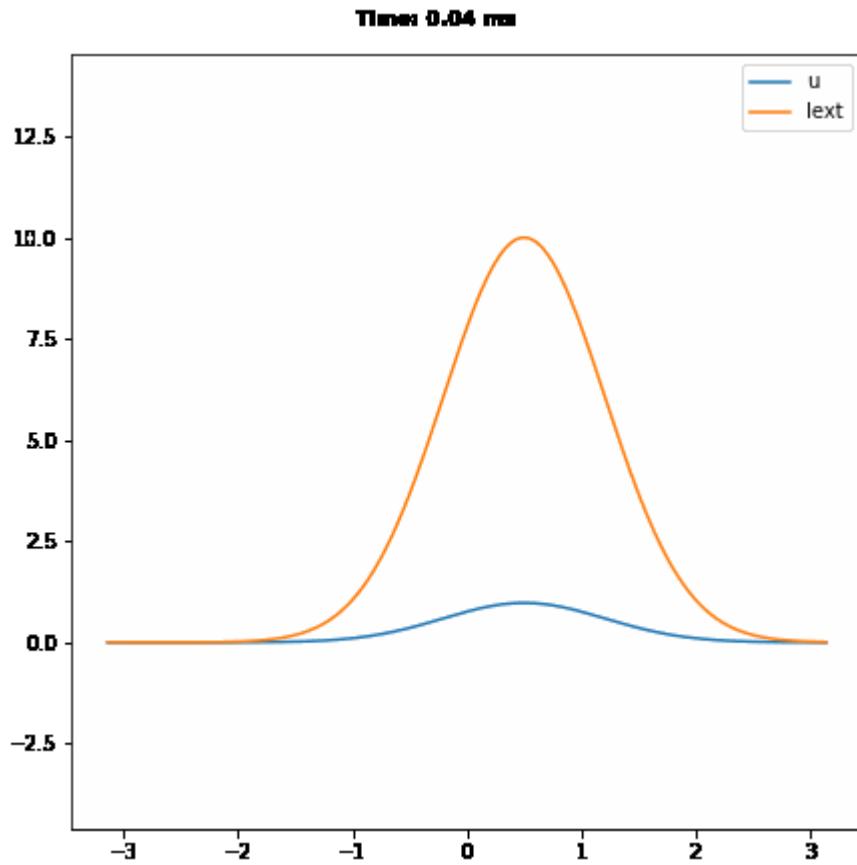
We can see that the shape of u encodes/zh/ the shape of external input.

Now we add random noise to the external input to see how the shape of u changes.

```
cann = CANN1D(num=512, k=8.1, monitors=['u'])

dur1, dur2, dur3 = 10., 30., 0.
num1 = int(dur1 / bp.backend.get_dt())
num2 = int(dur2 / bp.backend.get_dt())
num3 = int(dur3 / bp.backend.get_dt())
Iext = np.zeros((num1 + num2 + num3,) + cann.size)
Iext[:num1] = cann.get_stimulus_by_pos(0.5)
Iext[num1:num1 + num2] = cann.get_stimulus_by_pos(0.)
Iext[num1:num1 + num2] += 0.1 * cann.A * np.random.randn(n
cann.run(duration=dur1 + dur2 + dur3, inputs='input', Iext

plot_animate()
```



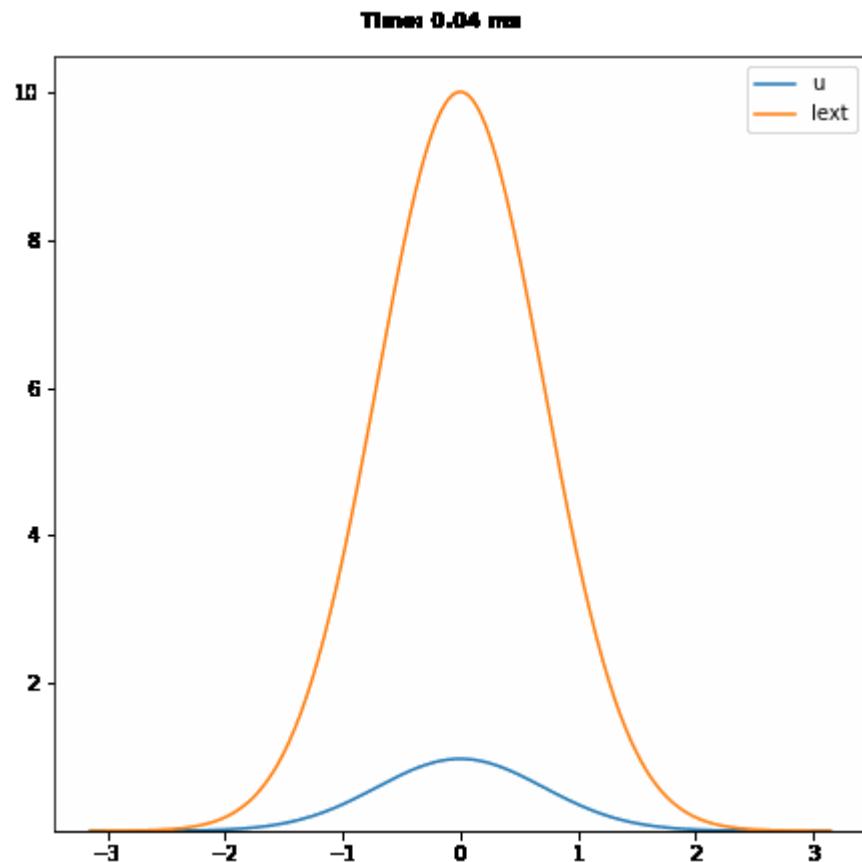
We can see that the shape of u remains like a bell shape, which indicates that it can perform template matching based on the input.

Now let's give a moving input, we vary the position of the input with `np.linspace`, we will see that the u will follow the input, i.e., smooth tracking.

```
cann = CANN1D(num=512, k=8.1, monitors=['u'])

dur1, dur2, dur3 = 20., 20., 20.
num1 = int(dur1 / bp.backend.get_dt())
num2 = int(dur2 / bp.backend.get_dt())
num3 = int(dur3 / bp.backend.get_dt())
position = np.zeros(num1 + num2 + num3)
position[num1: num1 + num2] = np.linspace(0., 12., num2)
position[num1 + num2:] = 12.
position = position.reshape((-1, 1))
Iext = cann.get_stimulus_by_pos(position)
cann.run(duration=dur1 + dur2 + dur3, inputs='input', Iext

plot_animate()
```



Reference

- [1] Wu, Si, Kosuke Hamaguchi, and Shun-ichi Amari. "Dynamics and computation of continuous attractors." *Neural computation* 20.4 (2008): 994-1025.

Appendix

Codes in appendix is the complete version of the codes attached in our main text. We also provide APIs for users to directly use some canonical models' BrainPy implementation. Please see github repository BrainModels (<https://github.com/PKU-NIP-Lab/BrainModels>) and its documentation (<https://brainmodels.readthedocs.io/en/latest/>) for more details.

- Appendix1. [neuron models](#)
- Appendix2. [synapse models](#)
- Appendix3. [network models](#)

Biophysical models

Hodgkin-Huxley model

```

import brainpy as bp
from numba import prange

class HH(bp.NeuGroup):
    target_backend = 'general'

    @staticmethod
    @bp.odeint(method='exponential_euler')
    def integral(V, m, h, n, t, C, gNa, ENa, gK, EK, gL, EL,
                 alpha_m = 0.1*(V+40)/(1-bp.ops.exp(-(V+40)/10))
                 beta_m = 4.0*bp.ops.exp(-(V+65)/18)
                 dmdt = alpha_m * (1 - m) - beta_m * m

                 alpha_h = 0.07*bp.ops.exp(-(V+65)/20)
                 beta_h = 1/(1+bp.ops.exp(-(V+35)/10))
                 dhdt = alpha_h * (1 - h) - beta_h * h

                 alpha_n = 0.01*(V+55)/(1-bp.ops.exp(-(V+55)/10))
                 beta_n = 0.125*bp.ops.exp(-(V+65)/80)
                 dnndt = alpha_n * (1 - n) - beta_n * n

                 I_Na = (gNa * m ** 3.0 * h) * (V - ENa)
                 I_K = (gK * n ** 4.0) * (V - EK)
                 I_leak = gL * (V - EL)
                 dVdt = (- I_Na - I_K - I_leak + Iext) / C

    return dVdt, dmdt, dhdt, dnndt

    def __init__(self, size, ENa=50., gNa=120., EK=-77., gK=3.
                 EL=-54.387, gL=0.03, V_th=20., C=1.0, **kwargs):
        # parameters
        self.ENa = ENa
        self.EK = EK
        self.EL = EL
        self.gNa = gNa
        self.gK = gK
        self.gL = gL
        self.C = C
        self.V_th = V_th

        # variables
        num = bp.size2len(size)
        self.V = -65. * bp.ops.ones(num)
        self.m = 0.5 * bp.ops.ones(num)
        self.h = 0.6 * bp.ops.ones(num)
        self.n = 0.32 * bp.ops.ones(num)
        self.spike = bp.ops.zeros(num, dtype=bool)
        self.input = bp.ops.zeros(num)

```

```

super(HH, self).__init__(size=size, **kwargs)

def update(self, _t):
    V, m, h, n = self.integral(self.V, self.m, self.h, self
                                C, self.gNa, self.ENa,
                                self.EK, self.gL, self.EL, s
    self.spike = (self.V < self.V_th) * (V >= self.V_th)
    self.V = V
    self.m = m
    self.h = h
    self.n = n
    self.input[:] = 0

import brainpy as bp

dt = 0.1
bp.backend.set('numpy', dt=dt)
neu = HH(100, monitors=['V', 'spike'])
neu.t_refractory = 5.
net = bp.Network(neu)
net.run(duration=200., inputs=(neu, 'input', 21.), report=1)
fig, gs = bp.visualize.get_figure(1, 1, 4, 10)
fig.add_subplot(gs[0, 0])
bp.visualize.line_plot(neu.mon.ts, neu.mon.V,
                      xlabel="t", ylabel="V", show=True)

```

Reduced models

LIF model

```

import brainpy as bp
from numba import prange

class LIF(bp.NeuGroup):
    target_backend = ['numpy', 'numba', 'numba-parallel', 'native']

    @staticmethod
    def derivative(V, t, Iext, V_rest, R, tau):
        dvdt = (-V + V_rest + R * Iext) / tau
        return dvdt

    def __init__(self, size, t_refractory=1., V_rest=0.,
                 V_reset=-5., V_th=20., R=1., tau=10., **kwargs):
        # parameters
        self.V_rest = V_rest
        self.V_reset = V_reset
        self.V_th = V_th
        self.R = R
        self.tau = tau
        self.t_refractory = t_refractory

        # variables
        num = bp.size2len(size)
        self.t_last_spike = bp.ops.ones(num) * -1e7
        self.input = bp.ops.zeros(num)
        self.refractory = bp.ops.zeros(num, dtype=bool)
        self.spike = bp.ops.zeros(num, dtype=bool)
        self.V = bp.ops.ones(num) * V_rest

        self.integral = bp.odeint(self.derivative)
        super(LIF, self).__init__(size=size, **kwargs)

    def update(self, _t):
        for i in prange(self.size[0]):
            spike = 0.
            refractory = (_t - self.t_last_spike[i] <= self.t_refractory)
            if not refractory:
                V = self.integral(self.V[i], _t, self.input[i],
                                  self.V_rest, self.R, self.tau)
                spike = (V >= self.V_th)
            if spike:
                V = self.V_reset
                self.t_last_spike[i] = _t
                self.V[i] = V
                self.spike[i] = spike
                self.refractory[i] = refractory or spike
                self.input[i] = 0.

```

1.1 Biological background

```
import brainpy as bp

dt = 0.1
bp.backend.set('numpy', dt=dt)
neu = LIF(100, monitors=['V', 'refractory', 'spike'])
neu.t_refractory = 5.
net = bp.Network(neu)
net.run(duration=200., inputs=(neu, 'input', 21.), report=1)
fig, gs = bp.visualize.get_figure(1, 1, 4, 10)
fig.add_subplot(gs[0, 0])
bp.visualize.line_plot(neu.mon.ts, neu.mon.V,
                      xlabel="t", ylabel="V", show=True)
```

QualF model

```

import brainpy as bp
from numba import prange

class QuaIF(bp.NeuGroup):
    target_backend = 'general'

    @staticmethod
    def derivative(V, t, I_ext, V_rest, V_c, R, tau, a_0):
        dVdt = (a_0 * (V - V_rest) * (V - V_c) + R * I_ext) / t
        return dVdt

    def __init__(self, size, V_rest=-65., V_reset=-68.,
                 V_th=-30., V_c=-50.0, a_0=.07,
                 R=1., tau=10., t_refractory=0., **kwargs):
        # parameters
        self.V_rest = V_rest
        self.V_reset = V_reset
        self.V_th = V_th
        self.V_c = V_c
        self.a_0 = a_0
        self.R = R
        self.tau = tau
        self.t_refractory = t_refractory

        # variables
        num = bp.size2len(size)
        self.V = bp.ops.ones(num) * V_reset
        self.input = bp.ops.zeros(num)
        self.spike = bp.ops.zeros(num, dtype=bool)
        self.refractory = bp.ops.zeros(num, dtype=bool)
        self.t_last_spike = bp.ops.ones(num) * -1e7

        self.integral = bp.odeint(f=self.derivative, method='rk45')
        super(QuaIF, self).__init__(size=size, **kwargs)

    def update(self, _t):
        for i in prange(self.size[0]):
            spike = 0.
            refractory = (_t - self.t_last_spike[i] <= self.t_refractory)
            if not refractory:
                V = self.integral(self.V[i], _t, self.input[i],
                                  self.V_rest, self.V_c, self.R,
                                  self.tau, self.a_0)
                spike = (V >= self.V_th)
            if spike:
                V = self.V_rest
                self.t_last_spike[i] = _t
                self.V[i] = V

```

```
self.spike[i] = spike
self.refractory[i] = refractory or spike
self.input[i] = 0.

dt = 0.1
bp.backend.set('numpy', dt=dt)
neu = QuaIF(100, monitors=['V', 'refractory', 'spike'])
neu.t_refractory = 5.
net = bp.Network(neu)
net.run(duration=200., inputs=(neu, 'input', 21.), report=1)
fig, gs = bp.visualize.get_figure(1, 1, 4, 10)
fig.add_subplot(gs[0, 0])
bp.visualize.line_plot(neu.mon.ts, neu.mon.V,
                      xlabel="t", ylabel="V", show=True)
```

ExplF model

```

import brainpy as bp
from numba import prange

class ExpIF(bp.NeuGroup):
    target_backend = 'general'

    @staticmethod
    def derivative(V, t, I_ext, V_rest, delta_T, V_T, R, tau):
        exp_term = bp.ops.exp((V - V_T) / delta_T)
        dvdt = (-(V-V_rest) + delta_T*exp_term + R*I_ext) / tau
        return dvdt

    def __init__(self, size, V_rest=-65., V_reset=-68.,
                 V_th=-30., V_T=-59.9, delta_T=3.48,
                 R=10., C=1., tau=10., t_refractory=1.7,
                 **kwargs):
        # parameters
        self.V_rest = V_rest
        self.V_reset = V_reset
        self.V_th = V_th
        self.V_T = V_T
        self.delta_T = delta_T
        self.R = R
        self.C = C
        self.tau = tau
        self.t_refractory = t_refractory

        # variables
        self.V = bp.ops.ones(size) * V_rest
        self.input = bp.ops.zeros(size)
        self.spike = bp.ops.zeros(size, dtype=bool)
        self.refractory = bp.ops.zeros(size, dtype=bool)
        self.t_last_spike = bp.ops.ones(size) * -1e7

        self.integral = bp.odeint(self.derivative)
        super(ExpIF, self).__init__(size=size, **kwargs)

    def update(self, _t):
        for i in prange(self.num):
            spike = 0.
            refractory = (_t - self.t_last_spike[i] <= self.t_refractory)
            if not refractory:
                V = self.integral(
                    self.V[i], _t, self.input[i], self.V_rest,
                    self.delta_T, self.V_T, self.R, self.tau
                )
                spike = (V >= self.V_th)
            if spike:

```

```
V = self.V_reset
    self.t_last_spike[i] = _t
    self.V[i] = V
    self.spike[i] = spike
    self.refractory[i] = refractory or spike
    self.input[:] = 0.

dt = 0.1
bp.backend.set('numpy', dt=dt)
neu = ExpIF(100, monitors=['V', 'refractory', 'spike'])
neu.t_refractory = 5.
net = bp.Network(neu)
net.run(duration=200., inputs=(neu, 'input', 21.), report=1)
fig, gs = bp.visualize.get_figure(1, 1, 4, 10)
fig.add_subplot(gs[0, 0])
bp.visualize.line_plot(neu.mon.ts, neu.mon.V,
                      xlabel="t", ylabel="V", show=True)
```

AdExIF model

```

import brainpy as bp
from numba import prange

class AdExIF(bp.NeuGroup):
    target_backend = 'general'

    @staticmethod
    def derivative(V, w, t, I_ext, V_rest, delta_T, V_T, R, t
                  exp_term = bp.ops.exp((V-V_T)/delta_T)
                  dVdt = (-(V-V_rest)+delta_T*exp_term-R*w+R*I_ext)/tau

    dwdt = (a*(V-V_rest)-w)/tau_w

    return dVdt, dwdt

    def __init__(self, size, V_rest=-65., V_reset=-68.,
                 V_th=-30., V_T=-59.9, delta_T=3.48,
                 a=1., b=1., R=10., tau=10., tau_w=30.,
                 t_refractory=0., **kwargs):
        # parameters
        self.V_rest = V_rest
        self.V_reset = V_reset
        self.V_th = V_th
        self.V_T = V_T
        self.delta_T = delta_T
        self.a = a
        self.b = b
        self.R = R
        self.tau = tau
        self.tau_w = tau_w
        self.t_refractory = t_refractory

        # variables
        num = bp.size2len(size)
        self.V = bp.ops.ones(num) * V_reset
        self.w = bp.ops.zeros(size)
        self.input = bp.ops.zeros(num)
        self.spike = bp.ops.zeros(num, dtype=bool)
        self.refractory = bp.ops.zeros(num, dtype=bool)
        self.t_last_spike = bp.ops.ones(num) * -1e7

        self.integral = bp.odeint(f=self.derivative, method='euler')

    super(AdExIF, self).__init__(size=size, **kwargs)

    def update(self, _t):
        for i in prange(self.size[0]):
            spike = 0.

```

```

refractory = (_t - self.t_last_spike[i] <= self.t_ref
if not refractory:
    V, w = self.integral(self.V[i], self.w[i], _t, self
                           self.V_rest, self.delta_T,
                           self.V_T, self.R, self.tau, se
                           spike = (V >= self.V_th)
if spike:
    V = self.V_rest
    w += self.b
    self.t_last_spike[i] = _t
    self.V[i] = V
    self.w[i] = w
    self.spike[i] = spike
    self.refractory[i] = refractory or spike
    self.input[i] = 0.

dt = 0.1
bp.backend.set('numpy', dt=dt)
neu = AdExIF(100, monitors=['V', 'refractory', 'spike'])
neu.t_refractory = 5.
net = bp.Network(neu)
net.run(duration=200., inputs=(neu, 'input', 21.), report=1)
fig, gs = bp.visualize.get_figure(1, 1, 4, 10)
fig.add_subplot(gs[0, 0])
bp.visualize.line_plot(neu.mon.ts, neu.mon.V,
                      xlabel="t", ylabel="V", show=True)

```

Hindmarsh-Rose model

```

import brainpy as bp
from numba import prange

class HindmarshRose(bp.NeuGroup):
    target_backend = 'general'

    @staticmethod
    def derivative(V, y, z, t, a, b, I_ext, c, d, r, s, V_r
                  dVdt = y - a * V * V * V + b * V * V - z + I_ext
                  dydt = c - d * V * V - y
                  dzdt = r * (s * (V - V_rest) - z)
                  return dVdt, dydt, dzdt

    def __init__(self, size, a=1., b=3.,
                 c=1., d=5., r=0.01, s=4.,
                 V_rest=-1.6, **kwargs):
        # parameters
        self.a = a
        self.b = b
        self.c = c
        self.d = d
        self.r = r
        self.s = s
        self.V_rest = V_rest

        # variables
        num = bp.size2len(size)
        self.z = bp.ops.zeros(num)
        self.input = bp.ops.zeros(num)
        self.V = bp.ops.ones(num) * -1.6
        self.y = bp.ops.ones(num) * -10.
        self.spike = bp.ops.zeros(num, dtype=bool)

        self.integral = bp.odeint(f=self.derivative)
        super(HindmarshRose, self).__init__(size=size, **kwargs)

    def update(self, _t):
        for i in prange(self.num):
            V, self.y[i], self.z[i] = self.integral(
                self.V[i], self.y[i], self.z[i], _t,
                self.a, self.b, self.input[i],
                self.c, self.d, self.r, self.s,
                self.V_rest)
            self.V[i] = V
            self.input[i] = 0.

```

```

bp.backend.set('numba', dt=0.02)
mode = 'irregular_bursting'
param = {'quiescence': [1.0, 2.0], # a
          'spiking': [3.5, 5.0], # c
          'bursting': [2.5, 3.0], # d
          'irregular_spiking': [2.95, 3.3], # h
          'irregular_bursting': [2.8, 3.7], # g
          }
# set params of b and I_ext corresponding to different firing modes
print(f"parameters is set to firing mode <{mode}>")

group = HindmarshRose(size=10, b=param[mode][0],
                      monitors=['V', 'y', 'z'])

group.run(350., inputs=('input', param[mode][1]), report=True)
bp.visualize.line_plot(group.mon.ts, group.mon.V, show=True)

# Phase plane analysis
phase_plane_analyzer = bp.analysis.PhasePlane(
    group.integral,
    target_vars={'V': [-3., 3.], 'y': [-20., 5.]},
    fixed_vars={'z': 0.},
    pars_update={'I_ext': param[mode][1], 'a': 1., 'b': 3.,
                 'c': 1., 'd': 5., 'r': 0.01, 's': 4.,
                 'V_rest': -1.6}
)
phase_plane_analyzer.plot_nullcline()
phase_plane_analyzer.plot_fixed_point()
phase_plane_analyzer.plot_vector_field()
phase_plane_analyzer.plot_trajectory(
    [{'V': 1., 'y': 0., 'z': -0.}],
    duration=100.,
    show=True
)

```

GeneralizedIF model

```

import brainpy as bp
from numba import prange

class GeneralizedIF(bp.NeuGroup):
    target_backend = 'general'

    @staticmethod
    def derivative(I1, I2, V_th, V, t,
                  k1, k2, a, V_rest, b, V_th_inf,
                  R, I_ext, tau):
        dI1dt = - k1 * I1
        dI2dt = - k2 * I2
        dVthdt = a * (V - V_rest) - b * (V_th - V_th_inf)
        dVdt = (- (V - V_rest) + R * I_ext + R * I1 + R * I2) /
        return dI1dt, dI2dt, dVthdt, dVdt

    def __init__(self, size, V_rest=-70., V_reset=-70.,
                 V_th_inf=-50., V_th_reset=-60., R=20., tau=2,
                 a=0., b=0.01, k1=0.2, k2=0.02,
                 R1=0., R2=1., A1=0., A2=0.,
                 **kwargs):
        # params
        self.V_rest = V_rest
        self.V_reset = V_reset
        self.V_th_inf = V_th_inf
        self.V_th_reset = V_th_reset
        self.R = R
        self.tau = tau
        self.a = a
        self.b = b
        self.k1 = k1
        self.k2 = k2
        self.R1 = R1
        self.R2 = R2
        self.A1 = A1
        self.A2 = A2

        # vars
        self.input = bp.ops.zeros(size)
        self.spike = bp.ops.zeros(size, dtype=bool)
        self.I1 = bp.ops.zeros(size)
        self.I2 = bp.ops.zeros(size)
        self.V = bp.ops.ones(size) * -70.
        self.V_th = bp.ops.ones(size) * -50.

        self.integral = bp.odeint(self.derivative)
        super(GeneralizedIF, self).__init__(size=size, **kwargs)

```

1.1 Biological background

```
def update(self, _t):
    for i in prange(self.size[0]):
        I1, I2, V_th, V = self.integral(
            self.I1[i], self.I2[i], self.V_th[i], self.V[i], _t
            self.k1, self.k2, self.a, self.V_rest,
            self.b, self.V_th_inf,
            self.R, self.input[i], self.tau
        )
        self.spike[i] = self.V_th[i] < V
        if self.spike[i]:
            V = self.V_reset
            I1 = self.R1 * I1 + self.A1
            I2 = self.R2 * I2 + self.A2
            V_th = max(V_th, self.V_th_reset)
            self.I1[i] = I1
            self.I2[i] = I2
            self.V_th[i] = V_th
            self.V[i] = V
            self.f = 0.
            self.input[:] = self.f

import matplotlib.pyplot as plt
import brainpy as bp
import brainmodels

# set parameters
num2mode = ["tonic_spiking", "class_1", "spike_frequency_adaptation",
            "phasic_spiking", "accommodation", "threshold_value",
            "rebound_spike", "class_2", "integrator",
            "input_bistability", "hyperpolarization_induced_bursting",
            "tonic_bursting", "phasic_bursting", "rebound_kinetics",
            "mixed_mode", "afterpotentials", "basal_bistability",
            "preferred_frequency", "spike_latency"]

mode2param = {
    "tonic_spiking": {
        "input": [(1.5, 200.)]
    },
    "class_1": {
        "input": [(1. + 1e-6, 500.)]
    },
    "spike_frequency_adaptation": {
        "a": 0.005, "input": [(2., 200.)]
    },
    "phasic_spiking": {
        "a": 0.005, "input": [(1.5, 500.)]
    },
    "accommodation": {
```

1.1 Biological background

```
"a": 0.005,
  "input": [(1.5, 100.), (0, 500.), (0.5, 100.),
            (1., 100.), (1.5, 100.), (0., 100.)]
},
  "threshold_variability": {
    "a": 0.005,
    "input": [(1.5, 20.), (0., 180.), (-1.5, 20.),
              (0., 20.), (1.5, 20.), (0., 140.)]
},
  "rebound_spike": {
    "a": 0.005,
    "input": [(0, 50.), (-3.5, 750.), (0., 200.)]
},
  "class_2": {
    "a": 0.005,
    "input": [(2 * (1. + 1e-6), 200.)],
    "v_th": -30.
},
  "integrator": {
    "a": 0.005,
    "input": [(1.5, 20.), (0., 10.), (1.5, 20.), (0., 20.),
              (1.5, 20.), (0., 30.), (1.5, 20.), (0., 30.)]
},
  "input_bistability": {
    "a": 0.005,
    "input": [(1.5, 100.), (1.7, 400.),
              (1.5, 100.), (1.7, 400.)]
},
  "hyperpolarization_induced_spiking": {
    "v_th_reset": -60.,
    "v_th_inf": -120.,
    "input": [(-1., 400.)],
    "v_th": -50.
},
  "hyperpolarization_induced_bursting": {
    "v_th_reset": -60.,
    "v_th_inf": -120.,
    "A1": 10.,
    "A2": -0.6,
    "input": [(-1., 400.)],
    "v_th": -50.
},
  "tonic_bursting": {
    "a": 0.005,
    "A1": 10.,
    "A2": -0.6,
    "input": [(2., 500.)]
},
```

1.1 Biological background

```
"phasic_bursting": {
    "a": 0.005,
    "A1": 10.,
    "A2": -0.6,
    "input": [(1.5, 500.)]
},
"rebound_burst": {
    "a": 0.005,
    "A1": 10.,
    "A2": -0.6,
    "input": [(0, 100.), (-3.5, 500.), (0., 400.)]
},
"mixed_mode": {
    "a": 0.005,
    "A1": 5.,
    "A2": -0.3,
    "input": [(2., 500.)]
},
"afterpotentials": {
    "a": 0.005,
    "A1": 5.,
    "A2": -0.3,
    "input": [(2., 15.), (0, 185.)]
},
"basal_bistability": {
    "A1": 8.,
    "A2": -0.1,
    "input": [(5., 10.), (0., 90.), (5., 10.), (0., 90.)]
},
"preferred_frequency": {
    "a": 0.005,
    "A1": -3.,
    "A2": 0.5,
    "input": [(5., 10.), (0., 10.), (4., 10.), (0., 370),
              (5., 10.), (0., 90.), (4., 10.), (0., 290)]
},
"spike_latency": {
    "a": -0.08,
    "input": [(8., 2.), (0, 48.)]
}
}

def run_GIF_with_mode(mode='tonic_spiking', size=10.,
                      row_p=0, col_p=0, fig=None, gs=None):
    print(f"Running GIF neuron neu with mode '{mode}'")
    neu = brainmodels.neurons.GeneralizedIF(size, monitors=
param = mode2param[mode].items()
```

```

member_type = 0
for (k, v) in param:
    if k == 'input':
        I_ext, dur = bp.inputs.constant_current(v)
        member_type += 1
    else:
        if member_type == 0:
            exec("neu.%s = %f" % (k, v))
        else:
            exec("neu.%s = bp.ops.ones(size) * %f" % (k
neu.run(dur, inputs='input', I_ext), report=False)

ts = neu.mon.ts
ax1 = fig.add_subplot(gs[row_p, col_p])
ax1.title.set_text(f'{mode}')

ax1.plot(ts, neu.mon.V[:, 0], label='V')
ax1.plot(ts, neu.mon.V_th[:, 0], label='V_th')
ax1.set_xlabel('Time (ms)')
ax1.set_ylabel('Membrane potential')
ax1.set_xlim(-0.1, ts[-1] + 0.1)
plt.legend()

ax2 = ax1.twinx()
ax2.plot(ts, I_ext, color='turquoise', label='input')
ax2.set_xlabel('Time (ms)')
ax2.set_ylabel('External input')
ax2.set_xlim(-0.1, ts[-1] + 0.1)
ax2.set_ylim(-5., 20.)
plt.legend(loc='lower left')

size = 10
pattern_num = 20
row_b = 2
col_b = 2
size_b = row_b * col_b
for i in range(pattern_num):
    if i % size_b == 0:
        fig, gs = bp.visualize.get_figure(row_b, col_b, 4,
mode = num2mode[i]
run_GIF_with_mode(mode=mode, size=size,
                  row_p=i % size_b // col_b,
                  col_p=i % size_b % col_b,
                  fig=fig, gs=gs)
    if (i + 1) % size_b == 0:
        plt.show()

```



Firing rate models

Firing Rate Unit model

```

import brainpy as bp
from numba import prange

class FiringRateUnit(bp.NeuGroup):
    target_backend = 'general'

    @staticmethod
    def derivative(a_e, a_i, t,
                  k_e, r_e, c1, c2, I_ext_e, slope_e, theta_e,
                  k_i, r_i, c3, c4, I_ext_i, slope_i, theta_i):
        x_ae = c1 * a_e - c2 * a_i + I_ext_e
        sigmoid_ae_l = 1 / (1 + bp.ops.exp(- slope_e * (x_ae - theta_e)))
        sigmoid_ae_r = 1 / (1 + bp.ops.exp(slope_e * theta_e - x_ae))
        sigmoid_ae = sigmoid_ae_l - sigmoid_ae_r
        daedt = (- a_e + (k_e - r_e * a_e) * sigmoid_ae) / r_e

        x_ai = c3 * a_e - c4 * a_i + I_ext_i
        sigmoid_ai_l = 1 / (1 + bp.ops.exp(- slope_i * (x_ai - theta_i)))
        sigmoid_ai_r = 1 / (1 + bp.ops.exp(slope_i * theta_i - x_ai))
        sigmoid_ai = sigmoid_ai_l - sigmoid_ai_r
        daidt = (- a_i + (k_i - r_i * a_i) * sigmoid_ai) / r_i

        return daedt, daidt

    def __init__(self, size, c1=12., c2=4., c3=13., c4=11.,
                 k_e=1., k_i=1., tau_e=1., tau_i=1., r_e=1.,
                 slope_e=1.2, slope_i=1., theta_e=2.8, theta_i=1.5,
                 **kwargs):
        # params
        self.c1 = c1
        self.c2 = c2
        self.c3 = c3
        self.c4 = c4
        self.k_e = k_e
        self.k_i = k_i
        self.tau_e = tau_e
        self.tau_i = tau_i
        self.r_e = r_e
        self.r_i = r_i
        self.slope_e = slope_e
        self.slope_i = slope_i
        self.theta_e = theta_e
        self.theta_i = theta_i

        # vars
        self.input_e = bp.backend.zeros(size)
        self.input_i = bp.backend.zeros(size)
        self.a_e = bp.backend.ones(size) * 0.1
        self.a_i = bp.backend.ones(size) * 0.05

```

```
self.integral = bp.odeint(self.derivative)
super(FiringRateUnit, self).__init__(size=size, **k

def update(self, _t):
    self.a_e, self.a_i = self.integral(
        self.a_e, self.a_i, _t,
        self.k_e, self.r_e, self.c1, self.c2,
        self.input_e, self.slope_e,
        self.theta_e, self.tau_e,
        self.k_i, self.r_i, self.c3, self.c4,
        self.input_i, self.slope_i,
        self.theta_i, self.tau_i)
    self.input_e[:] = 0.
    self.input_i[:] = 0.
```



Appendix: Synapses

Synapse models

AMPA

```

import brainpy as bp

class AMPA(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

    @staticmethod
    def derivative(s, t, TT, alpha, beta):
        ds = alpha * TT * (1 - s) - beta * s
        return ds

    def __init__(self, pre, post, conn, alpha=0.98, beta=0.18,
                 T_duration=0.5, **kwargs):
        # parameters
        self.alpha = alpha
        self.beta = beta
        self.T = T
        self.T_duration = T_duration

        # connections
        self.conn = conn(pre.size, post.size)
        self.pre_ids, self.post_ids = conn.requires('pre_ids',
                                                    self.size = len(self.pre_ids))

        # variables
        self.s = bp.ops.zeros(self.size)
        self.t_last_pre_spike = -1e7 * bp.ops.ones(self.size)

        self.int_s = bp.odeint(f=self.derivative, method='exp'
super(AMPA, self).__init__(pre=pre, post=post, **kwargs)

    def update(self, _t):
        for i in range(self.size):
            pre_id = self.pre_ids[i]
            post_id = self.post_ids[i]

            if self.pre.spike[pre_id]:
                self.t_last_pre_spike[pre_id] = _t
                TT = ((_t - self.t_last_pre_spike[pre_id])
                      < self.T_duration) * self.T
                self.s[i] = self.int_s(self.s[i], _t, TT, self.alpha,

```

```
import brainmodels as bm

bp.backend.set(backend='numba', dt=0.1)
bm.set_backend(backend='numba')

def run_syn(syn_model, **kwargs):
    neu1 = bm.neurons.LIF(2, monitors=['V'])
    neu2 = bm.neurons.LIF(3, monitors=['V'])

    syn = syn_model(pre=neu1, post=neu2, conn=bp.connect.All2All,
                    monitors=['s'], **kwargs)

    net = bp.Network(neu1, syn, neu2)
    net.run(30., inputs=(neu1, 'input', 35.))
    bp.visualize.line_plot(net.ts, syn.mon.s, ylabel='s', shc

    ◀ ▶

run_syn(AMPA, T_duration=3.)
```

NMDA

```

class NMDA(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

    @staticmethod
    def derivative(s, x, t, tau_rise, tau_decay, a):
        dsdt = -s / tau_decay + a * x * (1 - s)
        dxdt = -x / tau_rise
        return dsdt, dxdt

    def __init__(self, pre, post, conn, delay=0., g_max=0.15,
                 alpha=0.062, beta=3.57, tau=100, a=0.5, tau_
                 # parameters
                 self.g_max = g_max
                 self.E = E
                 self.alpha = alpha
                 self.beta = beta
                 self.cc_Mg = cc_Mg
                 self.tau = tau
                 self.tau_rise = tau_rise
                 self.a = a
                 self.delay = delay

                 # connections
                 self.conn = conn(pre.size, post.size)
                 self.pre_ids, self.post_ids = conn.requires('pre_ids',
                     self.size = len(self.pre_ids)

                 # variables
                 self.s = bp.ops.zeros(self.size)
                 self.x = bp.ops.zeros(self.size)
                 self.g = self.register_constant_delay('g', size=self.size,
                     delay_time=delay)

                 self.integral = bp.odeint(f=self.derivative, method='rk4'
                     super(NMDA, self).__init__(pre=pre, post=post, **kwargs

    def update(self, _t):
        for i in range(self.size):
            pre_id = self.pre_ids[i]
            post_id = self.post_ids[i]

            self.x[i] += self.pre.spike[pre_id]
            self.s[i], self.x[i] = self.integral(self.s[i], self.
                self.tau_rise, self.
                self.a)

            # output

```

1.1 Biological background

```
g_inf_exp = bp.ops.exp(-self.alpha * self.post.V[post
g_inf = 1 + g_inf_exp * self.cc_Mg / self.beta
```

```
self.g.push(i, self.g_max * self.s[i] / g_inf)
```

```
I_syn = self.g.pull(i) * (self.post.V[post_id] - self
self.post.input[post_id] -= I_syn
```



```
run_syn(NMDA)
```

GABA_b

1.1 Biological background

```
self.R[i], G = self.integral(self.R[i], self.G[i], _t
                           TT, self.k4, self.k1, se
                           self.s[i] = G ** 4 / (G ** 4 + self.kd)
                           self.G[i] = G

                           self.g.push(i, self.g_max * self.s[i])
                           I_syn = self.g.pull(i) * (self.post.V[post_id] - self
                           self.post.input[post_id] -= I_syn
```

```
< ━━━━━━ >
```



```
neu1 = bm.neurons.LIF(2, monitors=['V'])
neu2 = bm.neurons.LIF(3, monitors=['V'])
syn = GABAa(pre=neu1, post=neu2, conn=bp.connect.All2All(),
net = bp.Network(neu1, syn, neu2)

# input
I, dur = bp.inputs.constant_current([(25, 20), (0, 1000)])
net.run(dur, inputs=(neu1, 'input', I))

bp.visualize.line_plot(net.ts, syn.mon.s, ylabel='s', show=
```

```
< ━━━━━━ >
```

Differences of two exponentials

```

class Two_exponentials(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

    @staticmethod
    def derivative(s, x, t, tau1, tau2):
        dxdt = (-(tau1 + tau2) * x - s) / (tau1 * tau2)
        dsdt = x
        return dsdt, dxdt

    def __init__(self, pre, post, conn, tau1=1.0, tau2=3.0, *args,
                 **kwargs):
        # parameters
        self.tau1 = tau1
        self.tau2 = tau2

        # connections
        self.conn = conn(pre.size, post.size)
        self.pre_ids, self.post_ids = conn.requires('pre_ids',
                                                     'post_ids')
        self.size = len(self.pre_ids)

        # variables
        self.s = bp.ops.zeros(self.size)
        self.x = bp.ops.zeros(self.size)

        self.integral = bp.odeint(f=self.derivative, method='rk4')

        super(Two_exponentials, self).__init__(pre=pre, post=post, *args,
                                               **kwargs)

    def update(self, _t):
        for i in range(self.size):
            pre_id = self.pre_ids[i]

            self.s[i], self.x[i] = self.integral(self.s[i], self.x[i],
                                                 self.tau1, self.tau2)
            self.x[i] += self.pre.spike[pre_id]

```

```
run_syn(Two_exponentials, tau1=2.)
```

Alpha

```

class Alpha(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

    @staticmethod
    def derivative(s, x, t, tau):
        dxdt = (-2 * tau * x - s) / (tau ** 2)
        dsdt = x
        return dsdt, dxdt

    def __init__(self, pre, post, conn, tau=3.0, **kwargs):
        # parameters
        self.tau = tau

        # connections
        self.conn = conn(pre.size, post.size)
        self.pre_ids, self.post_ids = conn.requires('pre_ids',
                                                    self.size = len(self.pre_ids))

        # variables
        self.s = bp.ops.zeros(self.size)
        self.x = bp.ops.zeros(self.size)

        self.integral = bp.odeint(f=self.derivative, method='rk4')

```

```
super(Alpha, self).__init__(pre=pre, post=post, **kwargs)
```

```

def update(self, _t):
    for i in range(self.size):
        pre_id = self.pre_ids[i]

        self.s[i], self.x[i] = self.integral(self.s[i], self.x[i],
                                              self.tau)
        self.x[i] += self.pre.spike[pre_id]

```

run_syn(Alpha)

Single exponential decay

```

class Exponential(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

    @staticmethod
    def derivative(s, t, tau):
        ds = -s / tau
        return ds

    def __init__(self, pre, post, conn, tau=8.0, **kwargs):
        # parameters
        self.tau = tau

        # connections
        self.conn = conn(pre.size, post.size)
        self.pre_ids, self.post_ids = conn.requires('pre_ids',
                                                     'post_ids')
        self.size = len(self.pre_ids)

        # variables
        self.s = bp.ops.zeros(self.size)

        self.integral = bp.odeint(f=self.derivative, method='ex')

    super(Exponential, self).__init__(pre=pre, post=post, **kwargs)

    def update(self, _t):
        for i in range(self.size):
            pre_id = self.pre_ids[i]

            self.s[i] = self.integral(self.s[i], _t, self.tau)
            self.s[i] += self.pre.spike[pre_id]

```

◀ ▶

```

run_syn(Exponential)

```

Voltage jump

```
class Voltage_jump(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

    def __init__(self, pre, post, conn, **kwargs):
        # connections
        self.conn = conn(pre.size, post.size)
        self.pre_ids, self.post_ids = conn.requires('pre_ids',
                                                    self.size = len(self.pre_ids)

        # variables
        self.s = bp.ops.zeros(self.size)

        super(Voltage_jump, self).__init__(pre=pre, post=post,

    def update(self, _t):
        for i in range(self.size):
            pre_id = self.pre_ids[i]
            self.s[i] = self.pre.spike[pre_id]

    run_syn(Voltage_jump)
```

Gap junction

```

class Gap_junction(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

    def __init__(self, pre, post, conn, delay=0., k_spikelet=
                  post_refractory=False, **kwargs):
        self.delay = delay
        self.k_spikelet = k_spikelet
        self.post_has_refractory = post_refractory

        # connections
        self.conn = conn(pre.size, post.size)
        self.pre_ids, self.post_ids = conn.requires('pre_ids',
                                                     'post_ids')
        self.size = len(self.pre_ids)

        # variables
        self.w = bp.ops.ones(self.size)
        self.spikelet = self.register_constant_delay('spikelet'
                                                      delay_time=delay)

    super(Gap_junction, self).__init__(pre=pre, post=post,
                                      **kwargs)

def update(self, _t):
    for i in range(self.size):
        pre_id = self.pre_ids[i]
        post_id = self.post_ids[i]

        self.post.input[post_id] += self.w[i] * (self.pre.V[pre_id] -
                                                 self.post.V[post_id])

        self.spikelet.push(i, self.w[i] * self.k_spikelet *
                           self.pre.spike[pre_id])

        out = self.spikelet.pull(i)
        if self.post_has_refractory:
            self.post.V[post_id] += out * (1. - self.post.refractory)
        else:
            self.post.V[post_id] += out

```

```

import matplotlib.pyplot as plt

neu0 = bm.neurons.LIF(1, monitors=['V'], t_refractory=0)
neu0.V = bp.ops.ones(neu0.V.shape) * -10.
neu1 = bm.neurons.LIF(1, monitors=['V'], t_refractory=0)
neu1.V = bp.ops.ones(neu1.V.shape) * -10.
syn = Gap_junction(pre=neu0, post=neu1, conn=bp.connect.All
                     k_spikelet=5.)
syn.w = bp.ops.ones(syn.w.shape) * .5

net = bp.Network(neu0, neu1, syn)
net.run(100., inputs=(neu0, 'input', 30.))

fig, gs = bp.visualize.get_figure(row_num=2, col_num=1, )

fig.add_subplot(gs[1, 0])
plt.plot(net.ts, neu0.mon.V[:, 0], label='V0')
plt.legend()

fig.add_subplot(gs[0, 0])
plt.plot(net.ts, neu1.mon.V[:, 0], label='V1')
plt.legend()
plt.show()

```

Synaptic plasticity

STP

```

class STP(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

    @staticmethod
    def derivative(s, u, x, t, tau, tau_d, tau_f):
        dsdt = -s / tau
        dudt = - u / tau_f
        dxdt = (1 - x) / tau_d
        return dsdt, dudt, dxdt

    def __init__(self, pre, post, conn, delay=0., U=0.15, tau=10.,
                 tau_d=200., tau_f=8., **kwargs):
        # parameters
        self.tau_d = tau_d
        self.tau_f = tau_f
        self.tau = tau
        self.U = U
        self.delay = delay

        # connections
        self.conn = conn(pre.size, post.size)
        self.pre_ids, self.post_ids = conn.requires('pre_ids',
                                                     'post_ids')
        self.size = len(self.pre_ids)

        # variables
        self.s = bp.ops.zeros(self.size)
        self.x = bp.ops.ones(self.size)
        self.u = bp.ops.zeros(self.size)
        self.w = bp.ops.ones(self.size)
        self.I_syn = self.register_constant_delay('I_syn', size=self.size,
                                                delay_time=delay)

        self.integral = bp.odeint(f=self.derivative, method='ex')

    super(STP, self).__init__(pre=pre, post=post, **kwargs)

    def update(self, _t):
        for i in range(self.size):
            pre_id = self.pre_ids[i]

            self.s[i], u, x = self.integral(self.s[i], self.u[i],
                                             self.tau, self.tau_d,
                                             self.tau_f)

            if self.pre.spike[pre_id] > 0:
                u += self.U * (1 - self.u[i])
                self.s[i] += self.w[i] * u * self.x[i]
                x -= u * self.x[i]
            self.u[i] = u

```

1.1 Biological background

```
self.x[i] = x

# output
post_id = self.post_ids[i]
self.I_syn.push(i, self.s[i])
self.post.input[post_id] += self.I_syn.pull(i)
```

```
def run_stp(**kwargs):
    neu1 = bm.neurons.LIF(1, monitors=['V'])
    neu2 = bm.neurons.LIF(1, monitors=['V'])

    syn = STP(pre=neu1, post=neu2, conn=bp.connect.All2All(
        monitors=['s', 'u', 'x'], **kwargs)
    net = bp.Network(neu1, syn, neu2)
    net.run(100., inputs=(neu1, 'input', 28.))
```

```
# plot
fig, gs = bp.visualize.get_figure(2, 1, 3, 7)

fig.add_subplot(gs[0, 0])
plt.plot(net.ts, syn.mon.u[:, 0], label='u')
plt.plot(net.ts, syn.mon.x[:, 0], label='x')
plt.legend()

fig.add_subplot(gs[1, 0])
plt.plot(net.ts, syn.mon.s[:, 0], label='s')
plt.legend()

plt.xlabel('Time (ms)')
plt.show()
```

```
run_stp(U=0.2, tau_d=150., tau_f=2.)
```

```
run_stp(U=0.1, tau_d=10, tau_f=100.)
```

STDP

```

class STDP(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

    @staticmethod
    def derivative(s, A_s, A_t, t, tau, tau_s, tau_t):
        dsdt = -s / tau
        dAsdt = - A_s / tau_s
        dAtdt = - A_t / tau_t
        return dsdt, dAsdt, dAtdt

    def __init__(self, pre, post, conn, delay=0., delta_A_s=0.,
                 delta_A_t=0.5, w_min=0., w_max=20., tau_s=10.,
                 tau=10., **kwargs):
        # parameters
        self.tau_s = tau_s
        self.tau_t = tau_t
        self.tau = tau
        self.delta_A_s = delta_A_s
        self.delta_A_t = delta_A_t
        self.w_min = w_min
        self.w_max = w_max
        self.delay = delay

        # connections
        self.conn = conn(pre.size, post.size)
        self.pre_ids, self.post_ids = self.conn.requires('pre_id')
        self.size = len(self.pre_ids)

        # variables
        self.s = bp.ops.zeros(self.size)
        self.A_s = bp.ops.zeros(self.size)
        self.A_t = bp.ops.zeros(self.size)
        self.w = bp.ops.ones(self.size) * 1.
        self.I_syn = self.register_constant_delay('I_syn', size=self.size,
                                                delay_time=delay)
        self.integral = bp.odeint(f=self.derivative, method='ex')

        super(STDP, self).__init__(pre=pre, post=post, **kwargs)

    def update(self, _t):
        for i in range(self.size):
            pre_id = self.pre_ids[i]
            post_id = self.post_ids[i]

            self.s[i], A_s, A_t = self.integral(self.s[i], self.A_s[i],
                                                 self.A_t[i], _t,
                                                 self.tau_s, self.tau_t,
                                                 self.w_min, self.w_max,
                                                 self.w[i], self.I_syn,
                                                 self.delay)

```

1.1 Biological background

```
w = self.w[i]
if self.pre.spike[pre_id] > 0:
    self.s[i] += w
    A_s += self.delta_A_s
    w -= A_t

if self.post.spike[post_id] > 0:
    A_t += self.delta_A_t
    w += A_s

self.A_s[i] = A_s
self.A_t[i] = A_t

self.w[i] = bp.ops.clip(w, self.w_min, self.w_max)

# output
self.I_syn.push(i, self.s[i])
self.post.input[post_id] += self.I_syn.pull(i)
```

```
duration = 300.
(I_pre, _) = bp.inputs.constant_current([(0, 5), (30, 15),
                                         (0, 15), (30, 15),
                                         (0, 15), (30, 15),
                                         (0, 98), (30, 15), # switch order: t_inte
                                         (0, 15), (30, 15),
                                         (0, 15), (30, 15),
                                         (0, duration-155-98)])

(I_post, _) = bp.inputs.constant_current([(0, 10), (30, 15)
                                         (0, 15), (30, 15),
                                         (0, 15), (30, 15),
                                         (0, 90), (30, 15), # switch order: t_inte
                                         (0, 15), (30, 15),
                                         (0, 15), (30, 15),
                                         (0, duration-160-90)])
```

1.1 Biological background

```
pre = bm.neurons.LIF(1, monitors=['spike'])
post = bm.neurons.LIF(1, monitors=['spike'])

syn = STDP(pre=pre, post=post, conn=bp.connect.All2All(),
           monitors=['s', 'w'])
net = bp.Network(pre, syn, post)
net.run(duration, inputs=[(pre, 'input', I_pre), (post, 'ir

# plot
fig, gs = bp.visualize.get_figure(4, 1, 2, 7)

def hide_spines(my_ax):
    plt.legend()
    plt.xticks([])
    plt.yticks([])
    my_ax.spines['left'].set_visible(False)
    my_ax.spines['right'].set_visible(False)
    my_ax.spines['bottom'].set_visible(False)
    my_ax.spines['top'].set_visible(False)

ax=fig.add_subplot(gs[0, 0])
plt.plot(net.ts, syn.mon.s[:, 0], label="s")
hide_spines(ax)

ax1=fig.add_subplot(gs[1, 0])
plt.plot(net.ts, pre.mon.spike[:, 0], label="pre spike")
plt.ylim(0, 2)
hide_spines(ax1)
plt.legend(loc = 'center right')

ax2=fig.add_subplot(gs[2, 0])
plt.plot(net.ts, post.mon.spike[:, 0], label="post spike")
plt.ylim(-1, 1)
hide_spines(ax2)

ax3=fig.add_subplot(gs[3, 0])
plt.plot(net.ts, syn.mon.w[:, 0], label="w")
plt.legend()
# hide spines
plt.yticks([])
ax3.spines['left'].set_visible(False)
ax3.spines['right'].set_visible(False)
ax3.spines['top'].set_visible(False)

plt.xlabel('Time (ms)')
plt.show()
```

Oja's rule

```

import numpy as np

bp.backend.set(backend='numpy', dt=0.1)

class Oja(bp.TwoEndConn):
    target_backend = 'numpy'

    @staticmethod
    def derivative(w, t, gamma, r_pre, r_post):
        dwdt = gamma * (r_post * r_pre - r_post * r_post * w)
        return dwdt

    def __init__(self, pre, post, conn, gamma=.005, w_max=1.,
                 # params
                 self.gamma = gamma
                 self.w_max = w_max
                 self.w_min = w_min
                 # no delay in firing rate models

                 # conns
                 self.conn = conn(pre.size, post.size)
                 self.conn_mat = conn.requires('conn_mat')
                 self.size = bp.ops.shape(self.conn_mat)

                 # data
                 self.w = bp.ops.ones(self.size) * 0.05

                 self.integral = bp.odeint(f=self.derivative)
                 super(Oja, self).__init__(pre=pre, post=post, **kwargs)

    def update(self, _t):
        w = self.conn_mat * self.w
        self.post.r = np.sum(w.T * self.pre.r, axis=1)

        # resize to matrix
        dim = self.size
        r_post = np.vstack((self.post.r,) * dim[0])
        r_pre = np.vstack((self.pre.r,) * dim[1]).T

        self.w = self.integral(w, _t, self.gamma, r_pre, r_post)

```



1.1 Biological background

```
class neu(bp.NeuGroup):
    target_backend = 'numpy'

    def __init__(self, size, **kwargs):
        self.r = bp.ops.zeros(size)
        super(neu, self).__init__(size=size, **kwargs)

    def update(self, _t):
        self.r = self.r
```

```

# create input
current1, _ = bp.inputs.constant_current([(2., 20.), (0., 2
                                              [(0., 20.), (0., 2
current2, _ = bp.inputs.constant_current([(2., 20.), (0., 2
current3, _ = bp.inputs.constant_current([(2., 20.), (0., 2
current_pre = np.vstack((current1, current2))
current_post = np.vstack((current3, current3))

# simulate
neu_pre = neu(2, monitors=['r'])
neu_post = neu(2, monitors=['r'])
syn = Oja(pre=neu_pre, post=neu_post, conn=bp.connect.All2All)
net = bp.Network(neu_pre, syn, neu_post)
net.run(duration=200., inputs=[(neu_pre, 'r', current_pre[0], 1
                                (neu_post, 'r', current_post[0], 1

# plot
fig, gs = bp.visualize.get_figure(4, 1, 2, 6)

fig.add_subplot(gs[0, 0])
plt.plot(net.ts, neu_pre.mon.r[:, 0], 'b', label='pre r1')
plt.legend()

fig.add_subplot(gs[1, 0])
plt.plot(net.ts, neu_pre.mon.r[:, 1], 'r', label='pre r2')
plt.legend()

fig.add_subplot(gs[2, 0])
plt.plot(net.ts, neu_post.mon.r[:, 0], color='purple', label='post r1')
plt.ylim([0, 4])
plt.legend()

fig.add_subplot(gs[3, 0])
plt.plot(net.ts, syn.mon.w[:, 0, 0], 'b', label='syn.w1')
plt.plot(net.ts, syn.mon.w[:, 1, 0], 'r', label='syn.w2')
plt.legend()
plt.show()

```

BCM rule

```

class BCM(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

    @staticmethod
    def derivative(w, t, lr, r_pre, r_post, r_th):
        dwdt = lr * r_post * (r_post - r_th) * r_pre
        return dwdt

    def __init__(self, pre, post, conn, lr=0.005, w_max=1., w_min=-1.):
        # parameters
        self.lr = lr
        self.w_max = w_max
        self.w_min = w_min
        self.dt = bp.backend.get_dt()

        # connections
        self.conn = conn(pre.size, post.size)
        self.conn_mat = conn.requires('conn_mat')
        self.size = bp.ops.shape(self.conn_mat)

        # variables
        self.w = bp.ops.ones(self.size) * .5
        self.sum_post_r = bp.ops.zeros(post.size[0])
        self.r_th = bp.ops.zeros(post.size[0])

        self.int_w = bp.odeint(f=self.derivative, method='rk4')

    super(BCM, self).__init__(pre=pre, post=post, **kwargs)

    def update(self, _t):
        # update threshold
        self.sum_post_r += self.post.r
        r_th = self.sum_post_r / (_t / self.dt + 1)
        self.r_th = r_th

        # resize to matrix
        w = self.w * self.conn_mat
        dim = self.size
        r_th = np.vstack((r_th,) * dim[0])
        r_post = np.vstack((self.post.r,) * dim[0])
        r_pre = np.vstack((self.pre.r,) * dim[1]).T

        # update w
        w = self.int_w(w, _t, self.lr, r_pre, r_post, r_th)
        self.w = np.clip(w, self.w_min, self.w_max)

        # output
        self.post.r = np.sum(w.T * self.pre.r, axis=1)

```

1.1 Biological background

```
# create input
group1, _ = bp.inputs.constant_current(([1.5, 1],
                                         [0, 1]) * 10)
group2, duration = bp.inputs.constant_current(([0, 1],
                                                [1., 1]) * 1
group1 = np.vstack((group1,) * 10))
group2 = np.vstack((group2,) * 10))
input_r = np.vstack((group1, group2))

# simulate
pre = neu(20, monitors=['r'])
post = neu(1, monitors=['r'])
bcm = BCM(pre=pre, post=post, conn=bp.connect.All2All(),
           monitors=['w'])
net = bp.Network(pre, bcm, post)
net.run(duration, inputs=(pre, 'r', input_r.T, "="))

# plot
fig, gs = bp.visualize.get_figure(2, 1)
fig.add_subplot(gs[1, 0], xlim=(0, duration), ylim=(0, bcm.
plt.plot(net.ts, bcm.mon.w[:, 0], 'b', label='w1')
plt.plot(net.ts, bcm.mon.w[:, 11], 'r', label='w2')
plt.title("weights")
plt.ylabel("weights")
plt.xlabel("t")
plt.legend()

fig.add_subplot(gs[0, 0], xlim=(0, duration))
plt.plot(net.ts, pre.mon.r[:, 0], 'b', label='r1')
plt.plot(net.ts, pre.mon.r[:, 11], 'r', label='r2')
plt.title("inputs")
plt.ylabel("firing rate")
plt.xlabel("t")
plt.legend()

plt.show()
```

Spiking neural networks

E/I balanced network

1.1 Biological background

```
# -*- coding: utf-8 -*-
import brainpy as bp
import brainmodels
import matplotlib.pyplot as plt
import numpy as np

bp.backend.set('numba')

N_E = 500
N_I = 500
prob = 0.1

tau = 10.
V_rest = -52.
V_reset = -60.
V_th = -50.

tau_decay = 2.

neu_E = brainmodels.neurons.LIF(N_E, monitors=['spike'])
neu_I = brainmodels.neurons.LIF(N_I, monitors=['spike'])
neu_E.V = V_rest + np.random.random(N_E) * (V_th - V_rest)
neu_I.V = V_rest + np.random.random(N_I) * (V_th - V_rest)

syn_E2E = brainmodels.synapses.Exponential(pre=neu_E, post=
                                             conn=bp.connect.
syn_E2I = brainmodels.synapses.Exponential(pre=neu_E, post=
                                             conn=bp.connect.
syn_I2E = brainmodels.synapses.Exponential(pre=neu_I, post=
                                             conn=bp.connect.
syn_I2I = brainmodels.synapses.Exponential(pre=neu_I, post=
                                             conn=bp.connect.

JE = 1 / np.sqrt(prob * N_E)
JI = 1 / np.sqrt(prob * N_I)
syn_E2E.w = JE
syn_E2I.w = JE
syn_I2E.w = -JI
syn_I2I.w = -JI

net = bp.Network(neu_E, neu_I,
                 syn_E2E, syn_E2I,
                 syn_I2E, syn_I2I)
net.run(500., inputs=[(neu_E, 'input', 3.), (neu_I, 'input'

fig, gs = bp.visualize.get_figure(4, 1, 2, 10)
fig.add_subplot(gs[:3, 0])
bp.visualization.raster_plot(net.ts, neu_E.mon.spike)
```

```
fig.add_subplot(gs[3, 0])
rate = bp.measure.firing_rate(neu_E.mon.spike, 5.)
plt.plot(net.ts, rate)
plt.show()
```

Decision making network

1.1 Biological background

```
# -*- coding: utf-8 -*-
"""
Implementation of the paper:

Wang, Xiao-Jing. "Probabilistic decision making by slow reverberation in cortical circuits." Neuron 36.5 (2002): 95
"""

import brainpy as bp
import numpy as np
import matplotlib.pyplot as plt

# set params
# set global params
dt = 0.05 # ms
method = 'exponential'
bp.backend.set('numpy', dt=dt)

# set network params
base_N_E = 1600
base_N_I = 400
net_scale = 5.
N_E = int(base_N_E // net_scale)
N_I = int(base_N_I // net_scale)

f = 0.15 # Note: proportion of neurons activated by one of
N_A = int(f * N_E)
N_B = int(f * N_E)
N_non = N_E - N_A - N_B # Note: N_E = N_A + N_B + N_non
print(f"N_E = {N_E} = {N_A} + {N_B} + {N_non}, N_I = {N_I}")
# Note: N_E[0:N_A]: A_group
#       N_E[N_A : N_A+N_B]: B_group
#       N_E[N_A + N_B: N_E]: non of A or B

time_scale = 1.
pre_period = 100. / time_scale
stim_period = 1000.
delay_period = 500. / time_scale
total_period = pre_period + stim_period + delay_period

# set LIF neu params
V_rest_E = -70. # mV
V_reset_E = -55. # mV
V_th_E = -50. # mV
g_E = 25. * 1e-3 # uS
R_E = 1 / g_E # MOhm
C_E = 0.5 # nF
tau_E = 20. # ms
t_refractory_E = 2. # ms
```

1.1 Biological background

```
print(f"R_E * C_E = {R_E * C_E} should be equal to tau_E =\n\nV_rest_I = -70. # mV\nV_reset_I = -55. # mV\nV_th_I = -50. # mV\ng_I = 20. * 1e-3 # uS\nR_I = 1 / g_I # Mohm\nC_I = 0.2 # nF\ntau_I = 10. # ms\nt_refractory_I = 1. # ms\nprint(f"R_I * C_I = {R_I * C_I} should be equal to tau_I =\n\n\nclass LIF(bp.NeuGroup):\n    target_backend = 'general'\n\n    @staticmethod\n    def derivative(V, t, I_ext, V_rest, R, tau):\n        dvdt = (- (V - V_rest) + R * I_ext) / tau\n        return dvdt\n\n    def __init__(self, size, V_rest=0., V_reset=0.,\n                 V_th=0., R=0., tau=0., t_refractory=0.,\n                 **kwargs):\n        self.V_rest = V_rest\n        self.V_reset = V_reset\n        self.V_th = V_th\n        self.R = R\n        self.tau = tau\n        self.t_refractory = t_refractory\n\n        self.V = bp.ops.zeros(size)\n        self.input = bp.ops.zeros(size)\n        self.spike = bp.ops.zeros(size, dtype=bool)\n        self.refractory = bp.ops.zeros(size, dtype=bool)\n        self.t_last_spike = bp.ops.ones(size) * -1e7\n\n        self.integral = bp.odeint(self.derivative)\n        super(LIF, self).__init__(size=size, **kwargs)\n\n    def update(self, _t):\n        # update variables\n        not_ref = (_t - self.t_last_spike > self.t_refractory)\n        self.V[not_ref] = self.integral(\n            self.V[not_ref], _t, self.input[not_ref],\n            self.V_rest, self.R, self.tau)\n        sp = (self.V > self.V_th)\n        self.V[sp] = self.V_reset
```

1.1 Biological background

```
self.t_last_spike[sp] = _t
self.spike = sp
self.refractory = ~not_ref
self.input[:] = 0.

# set syn params
E_AMPA = 0. # mV
tau_decay_AMPA = 2 # ms

E_NMDA = 0. # mV
alpha_NMDA = 0.062 #
beta_NMDA = 3.57 #
cc_Mg_NMDA = 1. # mM
a_NMDA = 0.5 # kHz/ms^-1
tau_rise_NMDA = 2. # ms
tau_decay_NMDA = 100. # ms

E_GABAa = -70. # mV
tau_decay_GABAa = 5. # ms

delay_syn = 0.5 # ms

class NMDA(bp.TwoEndConn):
    target_backend = 'general'

    @staticmethod
    def derivative(s, x, t, tau_rise, tau_decay, a):
        dxdt = -x / tau_rise
        dsdt = -s / tau_decay + a * x * (1 - s)
        return dsdt, dxdt

    def __init__(self, pre, post, conn, delay=0.,
                 g_max=0.15, E=0., cc_Mg=1.2,
                 alpha=0.062, beta=3.57, tau=100,
                 a=0.5, tau_rise=2., **kwargs):
        # parameters
        self.g_max = g_max
        self.E = E
        self.alpha = alpha
        self.beta = beta
        self.cc_Mg = cc_Mg
        self.tau = tau
        self.tau_rise = tau_rise
        self.a = a
        self.delay = delay
```

1.1 Biological background

```
# connections
self.conn = conn(pre.size, post.size)
self.conn_mat = conn.requires('conn_mat')
self.size = bp.ops.shape(self.conn_mat)

# variables
self.s = bp.ops.zeros(self.size)
self.x = bp.ops.zeros(self.size)
self.g = self.register_constant_delay('g', size=self.size,
                                       delay_time=delay)

self.integral = bp.odeint(self.derivative)
super(NMDA, self).__init__(pre=pre, post=post, **kwargs)

def update(self, _t):
    self.x += bp.ops.unsqueeze(self.pre.spike, 1) * self.cc_Mg
    self.s, self.x = self.integral(self.s, self.x, _t,
                                   self.tau_rise, self.tau,
                                   self.g.push(self.g_max * self.s))

    g_inf = 1 + self.cc_Mg / self.beta * \
            bp.ops.exp(-self.alpha * self.post.V)
    g_inf = 1 / g_inf
    self.post.input -= bp.ops.sum(self.g.pull(), axis=0) * \
        (self.post.V - self.E) * g_inf

class AMPA(bp.TwoEndConn):
    target_backend = 'general'

    @staticmethod
    def derivative(s, t, tau):
        ds = - s / tau
        return ds

    def __init__(self, pre, post, conn, delay=0.,
                 g_max=0.1, E=0., tau=2.0, **kwargs):
        # parameters
        self.g_max = g_max
        self.E = E
        self.tau = tau
        self.delay = delay

        # connections
        self.conn = conn(pre.size, post.size)
        self.conn_mat = conn.requires('conn_mat')
        self.size = bp.ops.shape(self.conn_mat)
```

1.1 Biological background

```
# data
self.s = bp.ops.zeros(self.size)
self.g = self.register_constant_delay('g', size=self.size,
                                         delay_time=delay)

self.int_s = bp.odeint(f=self.derivative, method='euler')
super(AMPA, self).__init__(pre=pre, post=post, **kwargs)

def update(self, _t):
    self.s = self.int_s(self.s, _t, self.tau)
    self.s += bp.ops.unsqueeze(self.pre.spike, 1) * self.cc
    self.g.push(self.g_max * self.s)
    self.post.input -= bp.ops.sum(self.g.pull(), 0) * (self

class GABAa(bp.TwoEndConn):
    target_backend = 'general'

    @staticmethod
    def derivative(s, t, tau_decay):
        dsdt = - s / tau_decay
        return dsdt

    def __init__(self, pre, post, conn, delay=0.,
                 g_max=0.4, E=-80., tau_decay=6.,
                 **kwargs):
        # parameters
        self.g_max = g_max
        self.E = E
        self.tau_decay = tau_decay
        self.delay = delay

        # connections
        self.conn = conn(pre.size, post.size)
        self.conn_mat = conn.requires('conn_mat')
        self.size = bp.ops.shape(self.conn_mat)

        # data
        self.s = bp.ops.zeros(self.size)
        self.g = self.register_constant_delay('g', size=self.size,
                                             delay_time=delay)

        self.integral = bp.odeint(self.derivative)
        super(GABAa, self).__init__(pre=pre, post=post, **kwargs)

    def update(self, _t):
        self.s = self.integral(self.s, _t, self.tau_decay)
        for i in range(self.pre.size[0]):
```

1.1 Biological background

```

        if self.pre.spike[i] > 0:
            self.s[i] += self.conn_mat[i]
            self.g.push(self.g_max * self.s)
            g = self.g.pull()
            self.post.input -= bp.ops.sum(g, axis=0) * (self.post.V

# set syn weights (only used in recurrent E connections)
w_pos = 1.7
w_neg = 1. - f * (w_pos - 1.) / (1. - f)
print(f"the structured weight is: w_pos = {w_pos}, w_neg =
# inside select group: w = w+
# between group / from non-select group to select group: w
# A2A B2B w+, A2B B2A w-, non2A non2B w-
weight = np.ones((N_E, N_E), dtype=np.float)
for i in range(N_A):
    weight[i, 0: N_A] = w_pos
    weight[i, N_A: N_A + N_B] = w_neg
for i in range(N_A, N_A + N_B):
    weight[i, N_A: N_A + N_B] = w_pos
    weight[i, 0: N_A] = w_neg
for i in range(N_A + N_B, N_E):
    weight[i, 0: N_A + N_B] = w_neg
print(f"Check constraints: Weight sum {weight.sum(axis=0)[0]} should be equal to N_E = {N_E}")

# set background params
poisson_freq = 2400. # Hz
g_max_ext2E_AMPA = 2.1 * 1e-3 # uS
g_max_ext2I_AMPA = 1.62 * 1e-3 # uS

g_max_E2E_AMPA = 0.05 * 1e-3 * net_scale
g_max_E2E_NMDA = 0.165 * 1e-3 * net_scale
g_max_E2I_AMPA = 0.04 * 1e-3 * net_scale
g_max_E2I_NMDA = 0.13 * 1e-3 * net_scale
g_max_I2E_GABAa = 1.3 * 1e-3 * net_scale
g_max_I2I_GABAa = 1.0 * 1e-3 * net_scale

# def neurons
# def E neurons/pyramidal neurons
neu_A = LIF(N_A, monitors=['spike', 'input', 'V'])
neu_A.V_rest = V_rest_E
neu_A.V_reset = V_reset_E
neu_A.V_th = V_th_E
neu_A.R = R_E
neu_A.tau = tau_E
neu_A.t_refractory = t_refractory_E
neu_A.V = bp.ops.ones(N_A) * V_rest_E

```

```

neu_B = LIF(N_B, monitors=['spike', 'input', 'V'])
neu_B.V_rest = V_rest_E
neu_B.V_reset = V_reset_E
neu_B.V_th = V_th_E
neu_B.R = R_E
neu_B.tau = tau_E
neu_B.t_refractory = t_refractory_E
neu_B.V = bp.ops.ones(N_B) * V_rest_E

neu_non = LIF(N_non, monitors=['spike', 'input', 'V'])
neu_non.V_rest = V_rest_E
neu_non.V_reset = V_reset_E
neu_non.V_th = V_th_E
neu_non.R = R_E
neu_non.tau = tau_E
neu_non.t_refractory = t_refractory_E
neu_non.V = bp.ops.ones(N_non) * V_rest_E

# def I neurons/interneurons
neu_I = LIF(N_I, monitors=['input', 'V'])
neu_I.V_rest = V_rest_I
neu_I.V_reset = V_reset_I
neu_I.V_th = V_th_I
neu_I.R = R_I
neu_I.tau = tau_I
neu_I.t_refractory = t_refractory_I
neu_I.V = bp.ops.ones(N_I) * V_rest_I

# def synapse connections
## define E2E conn
syn_A2A_AMPA = AMPA(pre=neu_A, post=neu_A,
                      conn=bp.connect.All2All(),
                      delay=delay_syn)
syn_A2A_NMDA = NMDA(pre=neu_A, post=neu_A,
                      conn=bp.connect.All2All(),
                      delay=delay_syn)

syn_A2B_AMPA = AMPA(pre=neu_A, post=neu_B,
                      conn=bp.connect.All2All(),
                      delay=delay_syn)
syn_A2B_NMDA = NMDA(pre=neu_A, post=neu_B,
                      conn=bp.connect.All2All(),
                      delay=delay_syn)

syn_A2non_AMPA = AMPA(pre=neu_A, post=neu_non,
                      conn=bp.connect.All2All(),
                      delay=delay_syn)

```

1.1 Biological background

```
syn_A2non_NMDA = NMDA(pre=neu_A, post=neu_non,
                       conn=bp.connect.All2All(),
                       delay=delay_syn)

syn_B2A_AMPA = AMPA(pre=neu_B, post=neu_A,
                      conn=bp.connect.All2All(),
                      delay=delay_syn)
syn_B2A_NMDA = NMDA(pre=neu_B, post=neu_A,
                      conn=bp.connect.All2All(),
                      delay=delay_syn)

syn_B2B_AMPA = AMPA(pre=neu_B, post=neu_B,
                      conn=bp.connect.All2All(),
                      delay=delay_syn)
syn_B2B_NMDA = NMDA(pre=neu_B, post=neu_B,
                      conn=bp.connect.All2All(),
                      delay=delay_syn)

syn_B2non_AMPA = AMPA(pre=neu_B, post=neu_non,
                       conn=bp.connect.All2All(),
                       delay=delay_syn)
syn_B2non_NMDA = NMDA(pre=neu_B, post=neu_non,
                       conn=bp.connect.All2All(),
                       delay=delay_syn)

syn_non2A_AMPA = AMPA(pre=neu_non, post=neu_A,
                       conn=bp.connect.All2All(),
                       delay=delay_syn)
syn_non2A_NMDA = NMDA(pre=neu_non, post=neu_A,
                       conn=bp.connect.All2All(),
                       delay=delay_syn)

syn_non2B_AMPA = AMPA(pre=neu_non, post=neu_B,
                       conn=bp.connect.All2All(),
                       delay=delay_syn)
syn_non2B_NMDA = NMDA(pre=neu_non, post=neu_B,
                       conn=bp.connect.All2All(),
                       delay=delay_syn)

syn_non2non_AMPA = AMPA(pre=neu_non, post=neu_non,
                        conn=bp.connect.All2All(),
                        delay=delay_syn)
syn_non2non_NMDA = NMDA(pre=neu_non, post=neu_non,
                        conn=bp.connect.All2All(),
                        delay=delay_syn)

syn_A2A_AMPA.g_max = g_max_E2E_AMPA * w_pos
syn_A2A_NMDA.g_max = g_max_E2E_NMDA * w_pos
```

1.1 Biological background

```
syn_A2B_AMPA.g_max = g_max_E2E_AMPA * w_neg
syn_A2B_NMDA.g_max = g_max_E2E_NMDA * w_neg

syn_A2non_AMPA.g_max = g_max_E2E_AMPA
syn_A2non_NMDA.g_max = g_max_E2E_NMDA

syn_B2A_AMPA.g_max = g_max_E2E_AMPA * w_neg
syn_B2A_NMDA.g_max = g_max_E2E_NMDA * w_neg

syn_B2B_AMPA.g_max = g_max_E2E_AMPA * w_pos
syn_B2B_NMDA.g_max = g_max_E2E_NMDA * w_pos

syn_B2non_AMPA.g_max = g_max_E2E_AMPA
syn_B2non_NMDA.g_max = g_max_E2E_NMDA

syn_non2A_AMPA.g_max = g_max_E2E_AMPA * w_neg
syn_non2A_NMDA.g_max = g_max_E2E_NMDA * w_neg

syn_non2B_AMPA.g_max = g_max_E2E_AMPA * w_neg
syn_non2B_NMDA.g_max = g_max_E2E_NMDA * w_neg

syn_non2non_AMPA.g_max = g_max_E2E_AMPA
syn_non2non_NMDA.g_max = g_max_E2E_NMDA

for i in [syn_A2A_AMPA, syn_A2B_AMPA, syn_A2non_AMPA,
           syn_B2A_AMPA, syn_B2B_AMPA, syn_B2non_AMPA,
           syn_non2A_AMPA, syn_non2B_AMPA, syn_non2non_AMPA]
    i.E = E_AMPA
    i.tau_decay = tau_decay_AMPA
    i.E = E_NMDA

for i in [syn_A2A_NMDA, syn_A2B_NMDA, syn_A2non_NMDA,
           syn_B2A_NMDA, syn_B2B_NMDA, syn_B2non_NMDA,
           syn_non2A_NMDA, syn_non2B_NMDA, syn_non2non_NMDA]
    i.alpha = alpha_NMDA
    i.beta = beta_NMDA
    i.cc_Mg = cc_Mg_NMDA
    i.a = a_NMDA
    i.tau_decay = tau_decay_NMDA
    i.tau_rise = tau_rise_NMDA

## define E2I conn
syn_A2I_AMPA = AMPA(pre=neu_A, post=neu_I,
                      conn=bp.connect.All2All(),
                      delay=delay_syn)
syn_A2I_NMDA = NMDA(pre=neu_A, post=neu_I,
                      conn=bp.connect.All2All(),
```

1.1 Biological background

```
        delay=delay_syn)

syn_B2I_AMPA = AMPA(pre=neu_B, post=neu_I,
                     conn=bp.connect.All2All(),
                     delay=delay_syn)
syn_B2I_NMDA = NMDA(pre=neu_B, post=neu_I,
                     conn=bp.connect.All2All(),
                     delay=delay_syn)

syn_non2I_AMPA = AMPA(pre=neu_non, post=neu_I,
                     conn=bp.connect.All2All(),
                     delay=delay_syn)
syn_non2I_NMDA = NMDA(pre=neu_non, post=neu_I,
                     conn=bp.connect.All2All(),
                     delay=delay_syn)

for i in [syn_A2I_AMPA, syn_B2I_AMPA, syn_non2I_AMPA]:
    i.g_max = g_max_E2I_AMPA
    i.E = E_AMPA
    i.tau_decay = tau_decay_AMPA

for i in [syn_A2I_NMDA, syn_B2I_NMDA, syn_non2I_NMDA]:
    i.g_max = g_max_E2I_NMDA
    i.E = E_NMDA
    i.alpha = alpha_NMDA
    i.beta = beta_NMDA
    i.cc_Mg = cc_Mg_NMDA
    i.a = a_NMDA
    i.tau_decay = tau_decay_NMDA
    i.tau_rise = tau_rise_NMDA

## define I2E conn
syn_I2A_GABAa = GABAa(pre=neu_I, post=neu_A,
                      conn=bp.connect.All2All(),
                      delay=delay_syn)
syn_I2B_GABAa = GABAa(pre=neu_I, post=neu_B,
                      conn=bp.connect.All2All(),
                      delay=delay_syn)
syn_I2non_GABAa = GABAa(pre=neu_I, post=neu_non,
                      conn=bp.connect.All2All(),
                      delay=delay_syn)
for i in [syn_I2A_GABAa, syn_I2B_GABAa, syn_I2non_GABAa]:
    i.g_max = g_max_I2E_GABAa
    i.E = E_GABAa
    i.tau_decay = tau_decay_GABAa

## define I2I conn
syn_I2I_GABAa = GABAa(pre=neu_I, post=neu_I,
```

1.1 Biological background

```
conn=bp.connect.All2All(),
delay=delay_syn)
syn_I2I_GABAa.g_max = g_max_I2I_GABAa
syn_I2I_GABAa.E = E_GABAa
syn_I2I_GABAa.tau_decay = tau_decay_GABAa

# def background poisson input
class PoissonInput(bp.NeuGroup):
    target_backend = 'general'

    def __init__(self, size, freqs, dt, **kwargs):
        self.freqs = freqs
        self.dt = dt

        self.spike = bp.ops.zeros(size, dtype=bool)

    super(PoissonInput, self).__init__(size=size, **kwargs)

    def update(self, _t):
        self.spike = np.random.random(self.size) < self.freqs *

# poisson_freq = 2400Hz
neu_poisson_A = PoissonInput(N_A, freqs=poisson_freq, dt=dt,
neu_poisson_B = PoissonInput(N_B, freqs=poisson_freq, dt=dt,
neu_poisson_non = PoissonInput(N_non, freqs=poisson_freq, c
neu_poisson_I = PoissonInput(N_I, freqs=poisson_freq, dt=dt

syn_back2A_AMPA = AMPA(pre=neu_poisson_A, post=neu_A,
                       conn=bp.connect.One2One())
syn_back2B_AMPA = AMPA(pre=neu_poisson_B, post=neu_B,
                       conn=bp.connect.One2One())
syn_back2non_AMPA = AMPA(pre=neu_poisson_non, post=neu_non,
                           conn=bp.connect.One2One())

syn_back2I_AMPA = AMPA(pre=neu_poisson_I, post=neu_I,
                       conn=bp.connect.One2One())

for i in [syn_back2A_AMPA, syn_back2B_AMPA, syn_back2non_AMPA]:
    i.g_max = g_max_ext2E_AMPA
    i.E = E_AMPA
    i.tau_decay = tau_decay_AMPA

syn_back2I_AMPA.g_max = g_max_ext2I_AMPA
syn_back2I_AMPA.E = E_AMPA
syn_back2I_AMPA.tau_decay = tau_decay_AMPA
# Note: all neurons receive 2400Hz background poisson input
```

1.1 Biological background

```
## def stimulus input
# Note: inputs only given to A and B group
mu_0 = 40.
coherence = 25.6
rou_A = mu_0 / 100.
rou_B = mu_0 / 100.
mu_A = mu_0 + rou_A * coherence
mu_B = mu_0 - rou_B * coherence
print(f"coherence = {coherence}, mu_A = {mu_A}, mu_B = {mu_B}")

class PoissonStim(bp.NeuGroup):
    """
    from time <t_start> to <t_end> during the simulation, this
    generates a poisson spike with frequency <self.freq>. how
    the value of <self.freq> changes every <t_interval> ms ar
    a Gaussian distribution defined by <mean_freq> and <var_f
    """
    target_backend = 'general'

    def __init__(self, size, dt=0., t_start=0., t_end=0., t_i
                  mean_freq=0., var_freq=20., **kwargs):
        self.dt = dt
        self.stim_start_t = t_start
        self.stim_end_t = t_end
        self.stim_change_freq_interval = t_interval
        self.mean_freq = mean_freq
        self.var_freq = var_freq

        self.freq = 0.
        self.t_last_change_freq = -1e7
        self.spike = bp.ops.zeros(size, dtype=bool)

    super(PoissonStim, self).__init__(size=size, **kwargs)

    def update(self, _t):
        if self.stim_start_t < _t < self.stim_end_t:
            if self.stim_change_freq_interval <= _t - self.t_last_change_freq:
                self.freq = np.random.normal(self.mean_freq, self.var_freq)
                self.freq = max(self.freq, 0)
                self.t_last_change_freq = _t
            self.spike = np.random.random(self.size) < (self.freq * self.dt)
        else:
            self.freq = 0.
            self.spike[:] = False

neu_input2A = PoissonStim(N_A, dt=dt, t_start=pre_period,
                         t_end=pre_period + stim_period,
```

1.1 Biological background

```
    t_interval=50., mean_freq=mu_A, \
    monitors=['freq'])
neu_input2B = PoissonStim(N_B, dt=dt, t_start=pre_period,
                           t_end=pre_period + stim_period,
                           t_interval=50., mean_freq=mu_B, \
                           monitors=['freq'])

syn_input2A_AMPA = AMPA(pre=neu_input2A, post=neu_A,
                        conn=bp.connect.OneToOne())
syn_input2B_AMPA = AMPA(pre=neu_input2B, post=neu_B,
                        conn=bp.connect.OneToOne())

syn_input2A_AMPA.g_max = g_max_ext2E_AMPA
syn_input2A_AMPA.E = E_AMPA
syn_input2A_AMPA.tau_decay = tau_decay_AMPA

syn_input2B_AMPA.g_max = g_max_ext2E_AMPA
syn_input2B_AMPA.E = E_AMPA
syn_input2B_AMPA.tau_decay = tau_decay_AMPA

# build & simulate network
net = bp.Network(
    neu_poisson_A, neu_poisson_B,
    neu_poisson_non, neu_poisson_I,
    # bg input
    syn_back2A_AMPA, syn_back2B_AMPA,
    syn_back2non_AMPA, syn_back2I_AMPA,
    # bg conn
    neu_input2A, neu_input2B,
    # stim input
    syn_input2A_AMPA, syn_input2B_AMPA,
    # stim conn
    neu_A, neu_B, neu_non, neu_I,
    # E(A B non), I neu
    syn_A2A_AMPA, syn_A2A_NMDA,
    syn_A2B_AMPA, syn_A2B_NMDA,
    syn_A2non_AMPA, syn_A2non_NMDA,
    syn_B2A_AMPA, syn_B2A_NMDA,
    syn_B2B_AMPA, syn_B2B_NMDA,
    syn_B2non_AMPA, syn_B2non_NMDA,
    syn_non2A_AMPA, syn_non2A_NMDA,
    syn_non2B_AMPA, syn_non2B_NMDA,
    syn_non2non_AMPA, syn_non2non_NMDA,
    # E2E conn
    syn_A2I_AMPA, syn_A2I_NMDA,
    syn_B2I_AMPA, syn_B2I_NMDA,
    syn_non2I_AMPA, syn_non2I_NMDA,
```

1.1 Biological background

```
# E2I conn
syn_I2A_GABAa, syn_I2B_GABAa, syn_I2non_GABAa,
# I2E conn
syn_I2I_GABAa
# I2I conn
)
# Note: you may also use .add method of bp.Network to add
#       NeuGroups and SynConns to network

net.run(duration=total_period, inputs=[], report=True)

# visualize
def compute_population_fr(data, time_window, time_step):
    spike_cnt_group = data.sum(axis=1)
    pop_num = data.shape[1]
    time_cnt = int(time_step // dt)
    first_step_sum = spike_cnt_group[0:time_cnt].sum(axis=0)
    pop_fr_group = []
    for t in range(data.shape[0]):
        if t < time_cnt:
            pop_fr_group.append((first_step_sum / time_step) / pc
        else:
            pop_fr_group.append(spike_cnt_group[t - time_cnt:t].s
    return pop_fr_group

fig, gs = bp.visualize.get_figure(4, 1, 4, 8)

fig.add_subplot(gs[0, 0])
bp.visualize.raster_plot(net.ts, neu_A.mon.spike,
                        markersize=1)
plt.xlabel("time")
plt.ylabel("spike of group A")
fig.add_subplot(gs[1, 0])
bp.visualize.raster_plot(net.ts, neu_B.mon.spike,
                        markersize=1)
plt.xlabel("time")
plt.ylabel("spike of group B")

fig.add_subplot(gs[2, 0])
print("computing fr...")
pop_fr_A = compute_population_fr(neu_A.mon.spike, time_winc
pop_fr_B = compute_population_fr(neu_B.mon.spike, time_winc
print("get fr")
plt.bar(net.ts, pop_fr_A, label="group A")
plt.bar(net.ts, pop_fr_B, label="group B")
plt.xlabel("time")
```

```
plt.ylabel("population activity")
plt.legend()

fig.add_subplot(gs[3, 0])
plt.plot(net.ts, neu_input2A.mon.freq, label="group A")
plt.plot(net.ts, neu_input2B.mon.freq, label="group B")
plt.xlabel("time")
plt.ylabel("input firing rate")
plt.legend()

plt.show()
```

Firing rate networks

Decision model

```

from collections import OrderedDict
import brainpy as bp

bp.backend.set(backend='numba', dt=0.1)

class Decision(bp.NeuGroup):
    target_backend = ['numpy', 'numba']

    @staticmethod
    def derivative(s1, s2, t, I, coh,
                  JAext, J_rec, J_inh, I_0,
                  a, b, d, tau_s, gamma):
        I1 = JAext * I * (1. + coh)
        I2 = JAext * I * (1. - coh)

        I_syn1 = J_rec * s1 - J_inh * s2 + I_0 + I1
        r1 = (a * I_syn1 - b) / (1. - bp.ops.exp(-d * (a * I_sy
        ds1dt = - s1 / tau_s + (1. - s1) * gamma * r1

        I_syn2 = J_rec * s2 - J_inh * s1 + I_0 + I2
        r2 = (a * I_syn2 - b) / (1. - bp.ops.exp(-d * (a * I_sy
        ds2dt = - s2 / tau_s + (1. - s2) * gamma * r2

    return ds1dt, ds2dt

def __init__(self, size, coh, JAext=.00117, J_rec=.3725,
            I_0=.3297, a=270., b=108., d=0.154, tau_s=.06
            **kwargs):
    # parameters
    self.coh = coh
    self.JAext = JAext
    self.J_rec = J_rec
    self.J_inh = J_inh
    self.I0 = I_0
    self.a = a
    self.b = b
    self.d = d
    self.tau_s = tau_s
    self.gamma = gamma

    # variables
    self.s1 = bp.ops.ones(size) * .06
    self.s2 = bp.ops.ones(size) * .06
    self.input = bp.ops.zeros(size)

    self.integral = bp.odeint(f=self.derivative, method='rk'

```

1.1 Biological background

```
super(Decision, self).__init__(size=size, **kwargs)

def update(self, _t):
    for i in range(self.size):
        self.s1[i], self.s2[i] = self.integral(self.s1[i], se
                                                self.input[i],
                                                self.JAext, se
                                                self.J_inh, se
                                                self.a, self.k
                                                self.tau_s, se
        self.input[i] = 0.
```

```
def phase_analyze(I, coh):
    decision = Decision(I, coh=coh)

    phase = bp.analysis.PhasePlane(decision.integral,
                                    target_vars=OrderedDict(s2
                                                               s1
                                                               fixed_vars=None,
                                                               pars_update=dict(I=I, coh=
                                                                 JAext=.06
                                                                 J_inh=.11
                                                                 a=270., k
                                                                 tau_s=.06
                                                               numerical_resolution=.001,
                                                               options={'escape_sympy_sol
```

```
phase.plot_nullcline()
phase.plot_fixed_point()
phase.plot_vector_field(show=True)
```

```
# no input
phase_analyze(I=0., coh=0.)
```

1.1 Biological background

```
# coherence = 0%
print("coherence = 0%")
phase_analyze(I=30., coh=0.)

# coherence = 51.2%
print("coherence = 51.2%")
phase_analyze(I=30., coh=0.512)

# coherence = 100%
print("coherence = 100%")
phase_analyze(I=30., coh=1.)
```

CANN

```

import brainpy as bp
import numpy as np
bp.backend.set(backend='numpy', dt=0.1)

class CANN1D(bp.NeuGroup):
    target_backend = ['numpy', 'numba']

    def __init__(self, num, tau=1., k=8.1, a=0.5, A=10., J0=4,
                 z_min=-np.pi, z_max=np.pi, **kwargs):
        # parameters
        self.tau = tau    # The synaptic time constant
        self.k = k        # Degree of the rescaled inhibition
        self.a = a        # Half-width of the range of excitatory
        self.A = A        # Magnitude of the external input
        self.J0 = J0      # maximum connection value

        # feature space
        self.z_min = z_min
        self.z_max = z_max
        self.z_range = z_max - z_min
        self.x = np.linspace(z_min, z_max, num)  # The encoded

        # variables
        self.u = np.zeros(num)
        self.input = np.zeros(num)

        # The connection matrix
        self.conn_mat = self.make_conn(self.x)

    super(CANN1D, self).__init__(size=num, **kwargs)

    self.rho = num / self.z_range    # The neural density
    self.dx = self.z_range / num    # The stimulus density

    @staticmethod
    @bp.odeint(method='rk4', dt=0.05)
    def int_u(u, t, conn, k, tau, Iext):
        r1 = np.square(u)
        r2 = 1.0 + k * np.sum(r1)
        r = r1 / r2
        Irec = np.dot(conn, r)
        du = (-u + Irec + Iext) / tau
        return du

    def dist(self, d):
        d = np.remainder(d, self.z_range)
        d = np.where(d > 0.5 * self.z_range, d - self.z_range,
                     return d

```

```

def make_conn(self, x):
    assert np.ndim(x) == 1
    x_left = np.reshape(x, (-1, 1))
    x_right = np.repeat(x.reshape((1, -1)), len(x), axis=0)
    d = self.dist(x_left - x_right)
    Jxx = self.J0 * np.exp(-0.5 * np.square(d / self.a)) /
          np.sqrt(2 * np.pi) * self.a)
    return Jxx

def get_stimulus_by_pos(self, pos):
    return self.A * np.exp(-0.25 * np.square(self.dist(self,
                                                       pos)))

def update(self, _t):
    self.u = self.int_u(self.u, _t, self.conn_mat, self.k,
                        self.input)
    self.input[:] = 0.

def plot_animate(frame_step=5, frame_delay=50):
    bp.visualize.animate_1D(dynamical_vars=[{'ys': cann.mon.u,
                                              'legend': 'u'},
                                             {'ys': self.u,
                                              'legend': 'u'}],
                            frame_step=frame_step, frame_delay=frame_delay,
                            show=True)

```

```

cann = CANN1D(num=512, k=0.1, monitors=['u'])

I1 = cann.get_stimulus_by_pos(0.)
Iext, duration = bp.inputs.constant_current([(0., 1.), (I1,
cann.run(duration=duration, inputs=('input', Iext)))

```

1.1 Biological background

```
# define function
def plot_animate(frame_step=5, frame_delay=50):
    bp.visualize.animate_1D(dynamical_vars=[{'ys': cann.mor
                                              'legend': 'u'}
                                             'xs': cann.x,
                                             frame_step=frame_step, frame_de
                                             show=True)

# call the function
plot_animate(frame_step=1, frame_delay=100)
```

```
cann = CANN1D(num=512, k=8.1, monitors=['u'])

dur1, dur2, dur3 = 10., 30., 0.
num1 = int(dur1 / bp.backend.get_dt())
num2 = int(dur2 / bp.backend.get_dt())
num3 = int(dur3 / bp.backend.get_dt())
Iext = np.zeros((num1 + num2 + num3,) + cann.size)
Iext[:num1] = cann.get_stimulus_by_pos(0.5)
Iext[num1:num1 + num2] = cann.get_stimulus_by_pos(0.)
Iext[num1:num1 + num2] += 0.1 * cann.A * np.random.randn(nl
cann.run(duration=dur1 + dur2 + dur3, inputs='input', Iext

plot_animate()
```

```
cann = CANN1D(num=512, k=8.1, monitors=['u'])

dur1, dur2, dur3 = 20., 20., 20.
num1 = int(dur1 / bp.backend.get_dt())
num2 = int(dur2 / bp.backend.get_dt())
num3 = int(dur3 / bp.backend.get_dt())
position = np.zeros(num1 + num2 + num3)
position[num1: num1 + num2] = np.linspace(0., 12., num2)
position[num1 + num2:] = 12.
position = position.reshape((-1, 1))
Iext = cann.get_stimulus_by_pos(position)
cann.run(duration=dur1 + dur2 + dur3, inputs='input', Iext

plot_animate()
```