

# 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 Spiking neural networks	1.4.1
3.2 Firing rate networks	1.4.2

## 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 group 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 [neuron models](#), [synapse models](#), and [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**

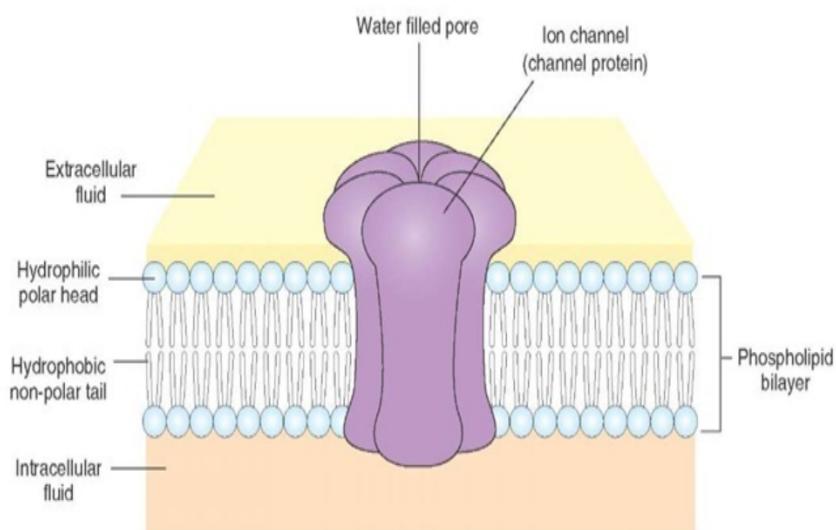
**1.3 Reduced models**

**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.



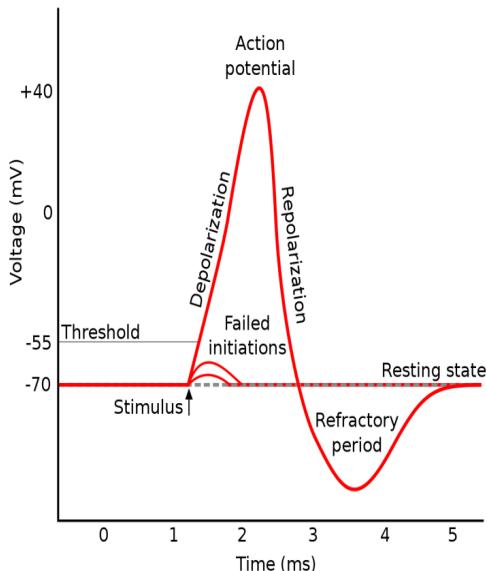
**Fig. 1-1 Neuron membrane diagram | what-when-how.com**

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.

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.



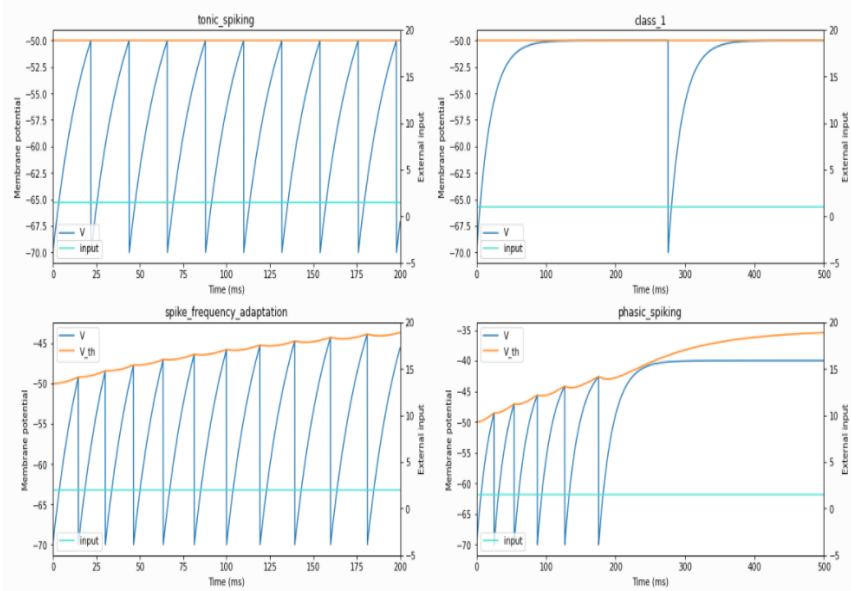
**Fig. 1-2 Action Potential | Wikipedia**

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  $\text{Na}^+$  and  $\text{K}^+$ ) 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,  $\text{Na}^+$  flow into the neuron and  $\text{K}^+$  flow out of the neuron, however the inflow of  $\text{Na}^+$  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  $\text{K}^+$  becomes faster than  $\text{Na}^+$ , 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.

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.

## 1.1 Biological background



**Figure 1-3 Some firing patterns**

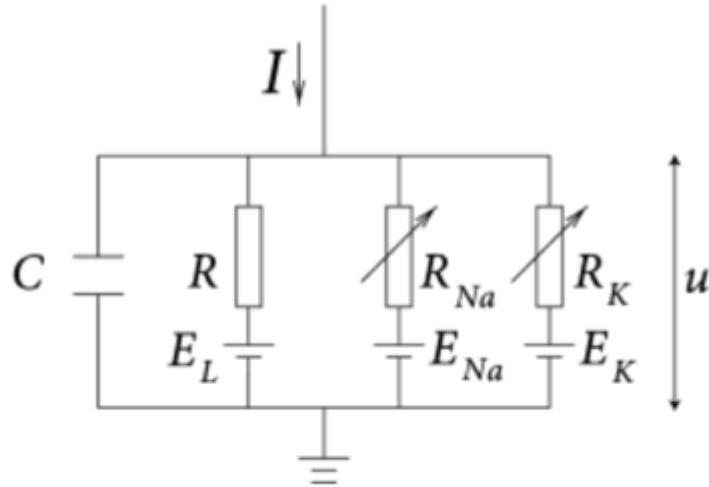
Those firing patterns, together with the shape of action potentials, are what computational neuroscience wants to model at the cellular level.

## 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 | NeuroDynamics**

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  $\text{Na}^+$  ion channels and  $\text{K}^+$  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:

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

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

## 1.1 Biological background

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<sup>+</sup> ion channels, K<sup>+</sup> 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<sup>+</sup> ion channel, variable n controls the state of K<sup>+</sup> 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  $\alpha_x$  refers to the activation rate of gating variable x, and  $\beta_x$  refers to the de-activation rate of x. The expressions of  $\alpha_x$  and  $\beta_x$  (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   class HH(bp.NeuGroup):           bp.NeuGroup class:
2       target_backend = 'general'    Group of neurons
3                                           ↓ Set backend for model.
4       @staticmethod               Call 'bp.odeint' to integrate ODEs.
5       @bp.odeint(method='exponential_euler') → Set parameter 'method' to choose
6       def integral(V, m, h, n, t, C, gNa, ENa, gK, EK, gL, EL, Text): numerical integration methods.
7           alpha_m = 0.1*(V+40)/(1-bp.ops.exp(-(V+40)/10)) α_m(V) = (0.1(V + 40))/(1 - exp(-(V + 40)/10))
8           beta_m = 4.0*bp.ops.exp(-(V+65)/18)             β_m(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)          α_h(V) = 0.07 exp(-(V + 65)/20)
12          beta_h = 1/(1+bp.ops.exp(-(V+35)/10))         β_h(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)) α_n(V) = (0.01(V + 55))/(1 - exp(-(V + 55)/10))
16          beta_n = 0.125*bp.ops.exp(-(V+65)/80)           β_n(V) = 0.125exp(-(V + 65)/80)
17          dn dt = 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)
20          I_K = (gK * n ** 4.0) * (V - EK)
21          I_leak = gL * (V - EL)
22          dVdt = (- I_Na - I_K - I_leak + Text) / C
23
24      return dVdt, dmdt, dhdt, dn dt

```

## 1.1 Biological background

```

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.

```

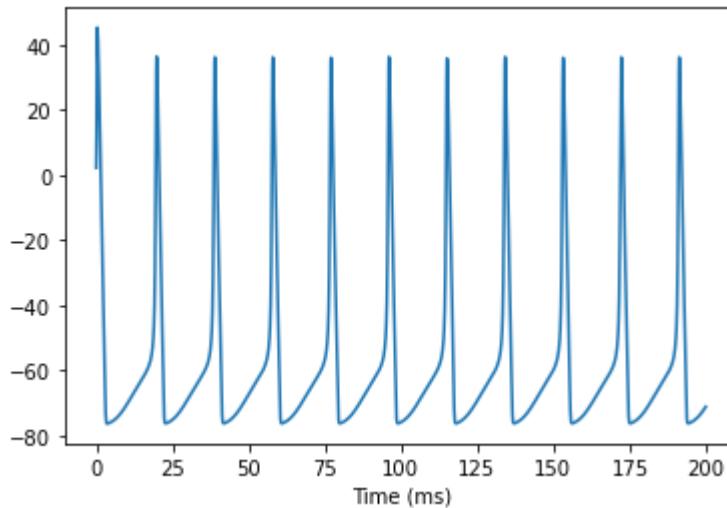
Model parameters saved as floating point numbers.

Model variables saved as vectors of floating point numbers.  
For example, if size = 100,

ENa	EK	...	V <sub>th</sub>
V[0]	V[1]	...	V[100]
m[0]	m[1]	...	m[100]
...	...	...	...

*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.



## 1.3 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

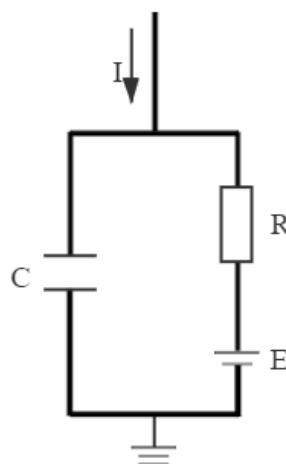
The most typical reduced model is the **Leaky Integrate-and-Fire model (LIF model)** presented by Lapicque (1907). LIF model is a combination of integrate process represented by differential equation and spike process represented by conditional judgment:

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

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

$$V \leftarrow V_{reset}$$

$\tau = RC$  is the time constant of LIF model, the larger  $\tau$  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  $\text{Na}^+$  and  $\text{K}^+$  ion channels any more. Actually, in LIF model, only the resistance  $R$ , capacitance  $C$ , battery  $E$  and external input  $I$  is modeled.



**Fig1-4 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

## 1.1 Biological background

generate it, another conditional judgment must be added:

If

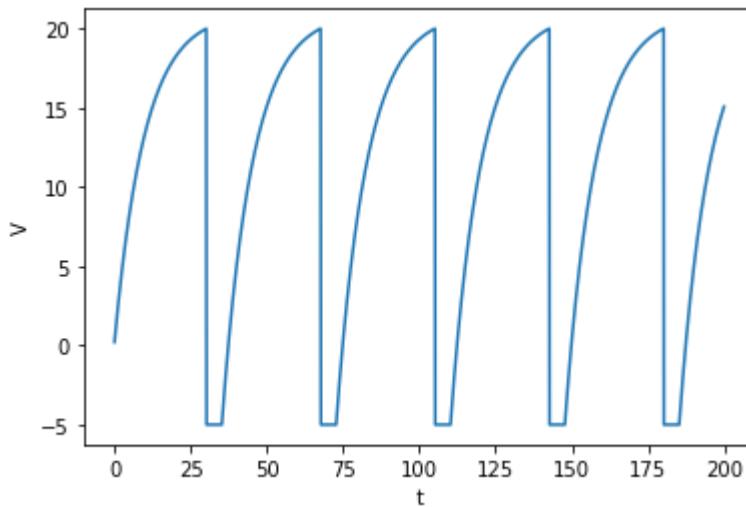
$$t - t_{lastspike} \leq refractoryperiod$$

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

```

1  class LIF(bp.NeuGroup): -----> bp.NeuGroup class:
2      target_backend = ['numpy', 'numba', 'numba-parallel', 'numba-cuda']
3      |-----> Group of neurons
4      @staticmethod
5      def derivative(V, t, Iext, V_rest, R, tau):
6          dvdt = (-V + V_rest + R * Iext) / tau ----->  $\tau \frac{dV}{dt} = -(V - V_{rest}) + RI(t)$ 
7          return dvdt
8
9      def __init__(self, size, t_refractory=1., V_rest=0.,
10                  V_reset=-5., V_th=20., R=1., tau=10., **kwargs):
11          # parameters
12          self.V_rest = V_rest
13          self.V_reset = V_reset
14          self.V_th = V_th
15          self.R = R
16          self.tau = tau
17          self.t_refractory = t_refractory
18
19          # variables
20          num = bp.size2len(size)
21          self.t_last_spike = bp.ops.ones(num) * -1e7
22          self.input = bp.ops.zeros(num)
23          self.refractory = bp.ops.zeros(num, dtype=bool)
24          self.spike = bp.ops.zeros(num, dtype=bool)
25          self.V = bp.ops.ones(num) * V_rest
26
27          self.integral = bp.odeint(self.derivative) -----> Call 'bp.odeint' to integrate ODEs.
28          super(LIF, self).__init__(size=size, **kwargs) -----> Set parameter 'method' to choose
29          |-----> numerical integration methods.
30      def update(self, _t):
31          for i in prange(self.size[0]): -----> Pass 'size' and '**kwargs' to
32              |-----> superclass bp.NeuGroup's constructor.
33              spike = 0.
34              refractory = (_t - self.t_last_spike[i] <= self.t_refractory) -----> For each neuron in neuron group.
35              if not refractory:
36                  V = self.integral(self.V[i], _t, self.input[i],
37                                    self.V_rest, self.R, self.tau) -----> Check if neuron is
38                  spike = (V >= self.V_th) -----> in refractory period.
39                  if spike:
40                      V = self.V_reset
41                      self.t_last_spike[i] = _t
42                      self.V[i] = V
43                      self.spike[i] = spike
44                      self.refractory[i] = refractory or spike
45                      self.input[i] = 0. -----> Update variables
46                      |-----> with numerical integration
47                      |-----> one by one.
48
49      |-----> Reset neuron.
50
51      |-----> Reset external input
52      |-----> for this time step.

```



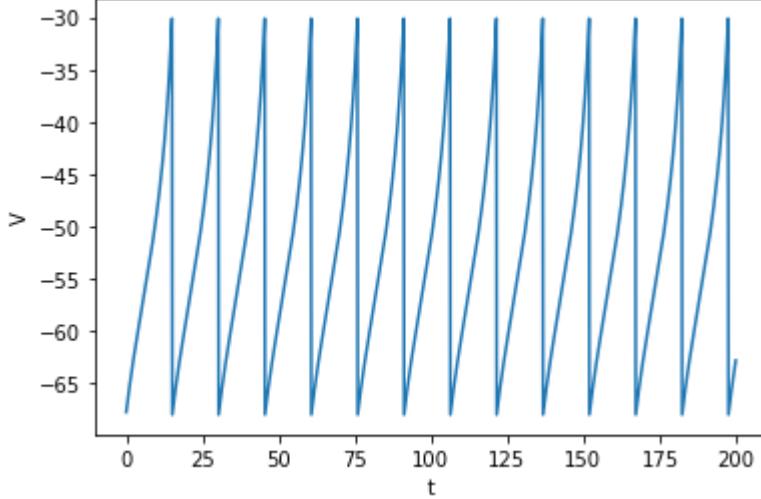
### 1.3.2 Quadratic Integrate-and-Fire model

To pursue higher representation capability, Latham et al. (2000) proposed **Quadratic Integrate-and-Fire model (QualF 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.



### 1.3.3 Exponential Integrate-and-Fire model

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

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

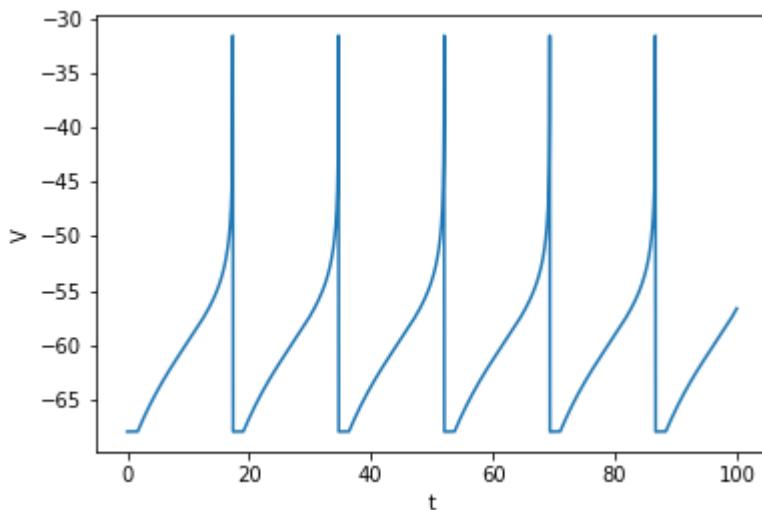
In the exponential term,  $V_T$  is the critical potential of generating action potential, below which  $V$  increases slowly and above which rapidly.  $\Delta_T$  is the slope of action potentials in ExplF model, and when  $\Delta_T \rightarrow 0$ , the shape of spikes in ExplF model will be equivalent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

## 1.1 Biological background

```

1   class ExpIF(bp.NeuGroup): → bp.NeuGroup class:
2       target_backend = 'general'
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
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
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
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
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
39              if not refractory: in refractory period.
40                  V = self.integral(
41                      self.V[i], _t, self.input[i], self.V_rest, → Update variables
42                      self.delta_T, self.V_T, self.R, self.tau → with numerical integration
43                  ) one by one.
44                  spike = (V >= self.V_th) → Check if neuron spikes.
45                  if spike:
46                      V = self.V_reset → Reset neuron.
47                      self.t_last_spike[i] = _t
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

## 1.1 Biological background

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, QualIF and ExpIF models. Here we introduce a typical one: **Adaptive Exponential Integrate-and-Fire model (AdExIF model)** (Gerstner et al., 2014).

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

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

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.

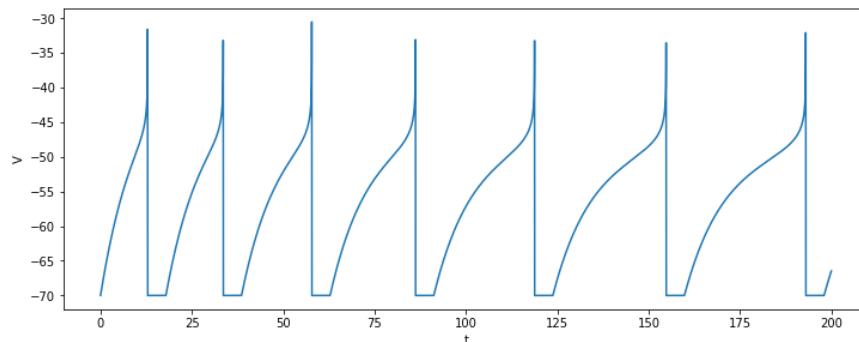
The weight term  $w$  is regulated by the second differential equation.  $a$  describes the sensitivity of the recovery variable  $w$  to the sub-threshold fluctuations of  $V$ , and  $b$  is the increment value of  $w$  generated by a spike, and  $w$  will also decay over time.

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):
2     target_backend = 'general'                                bp.NeuGroup class:
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                $\tau_w \frac{dw}{dt} = a(V - V_{rest}) - w + b \tau_w \sum \delta(t - t')$ 
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     # parameters
18     self.V_rest = V_rest
19     self.V_reset = V_reset
20     self.V_th = V_th
21     self.V_T = V_T
22     self.delta_T = delta_T
23     self.a = a
24     self.b = b
25     self.R = R
26     self.tau = tau
27     self.tau_w = tau_w
28     self.t_refractory = t_refractory
29
30     # variables
31     num = bp.size2len(size)
32     self.V = bp.ops.ones(num) * V_reset
33     self.w = bp.ops.zeros(size)
34     self.input = bp.ops.zeros(num)
35     self.spike = bp.ops.zeros(num, dtype=bool)
36     self.refractory = bp.ops.zeros(num, dtype=bool)
37     self.t_last_spike = bp.ops.ones(num) * -1e7
38
39     self.integral = bp.odeint(f=self.derivative, method='euler') → Call 'bp.odeint' to integrate ODEs.
40
41     super(AdExIF, self).__init__(size=size, **kwargs) → Set parameter 'method' to choose
42
43     def update(self, _t):
44         for i in prange(self.size[0]): → For each neuron in neuron group.
45             spike = 0.
46             refractory = (_t - self.t_last_spike[i] <= self.t_refractory) → Check if neuron is
47             if not refractory:                                         in refractory period.
48                 V, w = self.integral(self.V[i], self.w[i], _t, self.input[i], → Update variables
49                                         self.V_rest, self.delta_T, → with numerical integration
50                                         self.V_T, self.R, self.tau, self.tau_w, self.a) one by one.
51                 spike = (V >= self.V_th) → Check if neuron spikes.
52
53                 if spike:
54                     V = self.V_rest
55                     w += self.b
56                     self.t_last_spike[i] = _t
57                     self.V[i] = V
58                     self.w[i] = w
59                     self.spike[i] = spike
60                     self.refractory[i] = refractory or spike
61                     self.input[i] = 0.
62
63             self.w[i] = a(V - V_rest) - w + bτw ∑ δ(t - t')
64
65             dwdt = a(V - V_rest) - w + bτw ∑ δ(t - t')
66
67             self.w[i] = w
68
69             self.refractory[i] = refractory
70
71             self.input[i] = 0.
72
73             self.t_last_spike[i] = _t
74
75             self.V[i] = V
76
77             self.w[i] = w
78
79             self.refractory[i] = refractory
80
81             self.input[i] = 0.
82
83             self.t_last_spike[i] = _t
84
85             self.V[i] = V
86
87             self.w[i] = w
88
89             self.refractory[i] = refractory
90
91             self.input[i] = 0.
92
93             self.t_last_spike[i] = _t
94
95             self.V[i] = V
96
97             self.w[i] = w
98
99             self.refractory[i] = refractory
100
101             self.input[i] = 0.
102
103             self.t_last_spike[i] = _t
104
105             self.V[i] = V
106
107             self.w[i] = w
108
109             self.refractory[i] = refractory
110
111             self.input[i] = 0.
112
113             self.t_last_spike[i] = _t
114
115             self.V[i] = V
116
117             self.w[i] = w
118
119             self.refractory[i] = refractory
120
121             self.input[i] = 0.
122
123             self.t_last_spike[i] = _t
124
125             self.V[i] = V
126
127             self.w[i] = w
128
129             self.refractory[i] = refractory
130
131             self.input[i] = 0.
132
133             self.t_last_spike[i] = _t
134
135             self.V[i] = V
136
137             self.w[i] = w
138
139             self.refractory[i] = refractory
140
141             self.input[i] = 0.
142
143             self.t_last_spike[i] = _t
144
145             self.V[i] = V
146
147             self.w[i] = w
148
149             self.refractory[i] = refractory
150
151             self.input[i] = 0.
152
153             self.t_last_spike[i] = _t
154
155             self.V[i] = V
156
157             self.w[i] = w
158
159             self.refractory[i] = refractory
160
161             self.input[i] = 0.
162
163             self.t_last_spike[i] = _t
164
165             self.V[i] = V
166
167             self.w[i] = w
168
169             self.refractory[i] = refractory
170
171             self.input[i] = 0.
172
173             self.t_last_spike[i] = _t
174
175             self.V[i] = V
176
177             self.w[i] = w
178
179             self.refractory[i] = refractory
180
181             self.input[i] = 0.
182
183             self.t_last_spike[i] = _t
184
185             self.V[i] = V
186
187             self.w[i] = w
188
189             self.refractory[i] = refractory
190
191             self.input[i] = 0.
192
193             self.t_last_spike[i] = _t
194
195             self.V[i] = V
196
197             self.w[i] = w
198
199             self.refractory[i] = refractory
200
201             self.input[i] = 0.
202
203             self.t_last_spike[i] = _t
204
205             self.V[i] = V
206
207             self.w[i] = w
208
209             self.refractory[i] = refractory
210
211             self.input[i] = 0.
212
213             self.t_last_spike[i] = _t
214
215             self.V[i] = V
216
217             self.w[i] = w
218
219             self.refractory[i] = refractory
220
221             self.input[i] = 0.
222
223             self.t_last_spike[i] = _t
224
225             self.V[i] = V
226
227             self.w[i] = w
228
229             self.refractory[i] = refractory
230
231             self.input[i] = 0.
232
233             self.t_last_spike[i] = _t
234
235             self.V[i] = V
236
237             self.w[i] = w
238
239             self.refractory[i] = refractory
240
241             self.input[i] = 0.
242
243             self.t_last_spike[i] = _t
244
245             self.V[i] = V
246
247             self.w[i] = w
248
249             self.refractory[i] = refractory
250
251             self.input[i] = 0.
252
253             self.t_last_spike[i] = _t
254
255             self.V[i] = V
256
257             self.w[i] = w
258
259             self.refractory[i] = refractory
260
261             self.input[i] = 0.
262
263             self.t_last_spike[i] = _t
264
265             self.V[i] = V
266
267             self.w[i] = w
268
269             self.refractory[i] = refractory
270
271             self.input[i] = 0.
272
273             self.t_last_spike[i] = _t
274
275             self.V[i] = V
276
277             self.w[i] = w
278
279             self.refractory[i] = refractory
280
281             self.input[i] = 0.
282
283             self.t_last_spike[i] = _t
284
285             self.V[i] = V
286
287             self.w[i] = w
288
289             self.refractory[i] = refractory
290
291             self.input[i] = 0.
292
293             self.t_last_spike[i] = _t
294
295             self.V[i] = V
296
297             self.w[i] = w
298
299             self.refractory[i] = refractory
300
301             self.input[i] = 0.
302
303             self.t_last_spike[i] = _t
304
305             self.V[i] = V
306
307             self.w[i] = w
308
309             self.refractory[i] = refractory
310
311             self.input[i] = 0.
312
313             self.t_last_spike[i] = _t
314
315             self.V[i] = V
316
317             self.w[i] = w
318
319             self.refractory[i] = refractory
320
321             self.input[i] = 0.
322
323             self.t_last_spike[i] = _t
324
325             self.V[i] = V
326
327             self.w[i] = w
328
329             self.refractory[i] = refractory
330
331             self.input[i] = 0.
332
333             self.t_last_spike[i] = _t
334
335             self.V[i] = V
336
337             self.w[i] = w
338
339             self.refractory[i] = refractory
340
341             self.input[i] = 0.
342
343             self.t_last_spike[i] = _t
344
345             self.V[i] = V
346
347             self.w[i] = w
348
349             self.refractory[i] = refractory
350
351             self.input[i] = 0.
352
353             self.t_last_spike[i] = _t
354
355             self.V[i] = V
356
357             self.w[i] = w
358
359             self.refractory[i] = refractory
360
361             self.input[i] = 0.
362
363             self.t_last_spike[i] = _t
364
365             self.V[i] = V
366
367             self.w[i] = w
368
369             self.refractory[i] = refractory
370
371             self.input[i] = 0.
372
373             self.t_last_spike[i] = _t
374
375             self.V[i] = V
376
377             self.w[i] = w
378
379             self.refractory[i] = refractory
380
381             self.input[i] = 0.
382
383             self.t_last_spike[i] = _t
384
385             self.V[i] = V
386
387             self.w[i] = w
388
389             self.refractory[i] = refractory
390
391             self.input[i] = 0.
392
393             self.t_last_spike[i] = _t
394
395             self.V[i] = V
396
397             self.w[i] = w
398
399             self.refractory[i] = refractory
400
401             self.input[i] = 0.
402
403             self.t_last_spike[i] = _t
404
405             self.V[i] = V
406
407             self.w[i] = w
408
409             self.refractory[i] = refractory
410
411             self.input[i] = 0.
412
413             self.t_last_spike[i] = _t
414
415             self.V[i] = V
416
417             self.w[i] = w
418
419             self.refractory[i] = refractory
420
421             self.input[i] = 0.
422
423             self.t_last_spike[i] = _t
424
425             self.V[i] = V
426
427             self.w[i] = w
428
429             self.refractory[i] = refractory
430
431             self.input[i] = 0.
432
433             self.t_last_spike[i] = _t
434
435             self.V[i] = V
436
437             self.w[i] = w
438
439             self.refractory[i] = refractory
440
441             self.input[i] = 0.
442
443             self.t_last_spike[i] = _t
444
445             self.V[i] = V
446
447             self.w[i] = w
448
449             self.refractory[i] = refractory
450
451             self.input[i] = 0.
452
453             self.t_last_spike[i] = _t
454
455             self.V[i] = V
456
457             self.w[i] = w
458
459             self.refractory[i] = refractory
460
461             self.input[i] = 0.
462
463             self.t_last_spike[i] = _t
464
465             self.V[i] = V
466
467             self.w[i] = w
468
469             self.refractory[i] = refractory
470
471             self.input[i] = 0.
472
473             self.t_last_spike[i] = _t
474
475             self.V[i] = V
476
477             self.w[i] = w
478
479             self.refractory[i] = refractory
480
481             self.input[i] = 0.
482
483             self.t_last_spike[i] = _t
484
485             self.V[i] = V
486
487             self.w[i] = w
488
489             self.refractory[i] = refractory
490
491             self.input[i] = 0.
492
493             self.t_last_spike[i] = _t
494
495             self.V[i] = V
496
497             self.w[i] = w
498
499             self.refractory[i] = refractory
500
501             self.input[i] = 0.
502
503             self.t_last_spike[i] = _t
504
505             self.V[i] = V
506
507             self.w[i] = w
508
509             self.refractory[i] = refractory
510
511             self.input[i] = 0.
512
513             self.t_last_spike[i] = _t
514
515             self.V[i] = V
516
517             self.w[i] = w
518
519             self.refractory[i] = refractory
520
521             self.input[i] = 0.
522
523             self.t_last_spike[i] = _t
524
525             self.V[i] = V
526
527             self.w[i] = w
528
529             self.refractory[i] = refractory
530
531             self.input[i] = 0.
532
533             self.t_last_spike[i] = _t
534
535             self.V[i] = V
536
537             self.w[i] = w
538
539             self.refractory[i] = refractory
540
541             self.input[i] = 0.
542
543             self.t_last_spike[i] = _t
544
545             self.V[i] = V
546
547             self.w[i] = w
548
549             self.refractory[i] = refractory
550
551             self.input[i] = 0.
552
553             self.t_last_spike[i] = _t
554
555             self.V[i] = V
556
557             self.w[i] = w
558
559             self.refractory[i] = refractory
560
561             self.input[i] = 0.
562
563             self.t_last_spike[i] = _t
564
565             self.V[i] = V
566
567             self.w[i] = w
568
569             self.refractory[i] = refractory
570
571             self.input[i] = 0.
572
573             self.t_last_spike[i] = _t
574
575             self.V[i] = V
576
577             self.w[i] = w
578
579             self.refractory[i] = refractory
580
581             self.input[i] = 0.
582
583             self.t_last_spike[i] = _t
584
585             self.V[i] = V
586
587             self.w[i] = w
588
589             self.refractory[i] = refractory
590
591             self.input[i] = 0.
592
593             self.t_last_spike[i] = _t
594
595             self.V[i] = V
596
597             self.w[i] = w
598
599             self.refractory[i] = refractory
600
601             self.input[i] = 0.
602
603             self.t_last_spike[i] = _t
604
605             self.V[i] = V
606
607             self.w[i] = w
608
609             self.refractory[i] = refractory
610
611             self.input[i] = 0.
612
613             self.t_last_spike[i] = _t
614
615             self.V[i] = V
616
617             self.w[i] = w
618
619             self.refractory[i] = refractory
620
621             self.input[i] = 0.
622
623             self.t_last_spike[i] = _t
624
625             self.V[i] = V
626
627             self.w[i] = w
628
629             self.refractory[i] = refractory
630
631             self.input[i] = 0.
632
633             self.t_last_spike[i] = _t
634
635             self.V[i] = V
636
637             self.w[i] = w
638
639             self.refractory[i] = refractory
640
641             self.input[i] = 0.
642
643             self.t_last_spike[i] = _t
644
645             self.V[i] = V
646
647             self.w[i] = w
648
649             self.refractory[i] = refractory
650
651             self.input[i] = 0.
652
653             self.t_last_spike[i] = _t
654
655             self.V[i] = V
656
657             self.w[i] = w
658
659             self.refractory[i] = refractory
660
661             self.input[i] = 0.
662
663             self.t_last_spike[i] = _t
664
665             self.V[i] = V
666
667             self.w[i] = w
668
669             self.refractory[i] = refractory
670
671             self.input[i] = 0.
672
673             self.t_last_spike[i] = _t
674
675             self.V[i] = V
676
677             self.w[i] = w
678
679             self.refractory[i] = refractory
680
681             self.input[i] = 0.
682
683             self.t_last_spike[i] = _t
684
685             self.V[i] = V
686
687             self.w[i] = w
688
689             self.refractory[i] = refractory
690
691             self.input[i] = 0.
692
693             self.t_last_spike[i] = _t
694
695             self.V[i] = V
696
697             self.w[i] = w
698
699             self.refractory[i] = refractory
700
701             self.input[i] = 0.
702
703             self.t_last_spike[i] = _t
704
705             self.V[i] = V
706
707             self.w[i] = w
708
709             self.refractory[i] = refractory
710
711             self.input[i] = 0.
712
713             self.t_last_spike[i] = _t
714
715             self.V[i] = V
716
717             self.w[i] = w
718
719             self.refractory[i] = refractory
720
721             self.input[i] = 0.
722
723             self.t_last_spike[i] = _t
724
725             self.V[i] = V
726
727             self.w[i] = w
728
729             self.refractory[i] = refractory
730
731             self.input[i] = 0.
732
733             self.t_last_spike[i] = _t
734
735             self.V[i] = V
736
737             self.w[i] = w
738
739             self.refractory[i] = refractory
740
741             self.input[i] = 0.
742
743             self.t_last_spike[i] = _t
744
745             self.V[i] = V
746
747             self.w[i] = w
748
749             self.refractory[i] = refractory
750
751             self.input[i] = 0.
752
753             self.t_last_spike[i] = _t
754
755             self.V[i] = V
756
757             self.w[i] = w
758
759             self.refractory[i] = refractory
760
761             self.input[i] = 0.
762
763             self.t_last_spike[i] = _t
764
765             self.V[i] = V
766
767             self.w[i] = w
768
769             self.refractory[i] = refractory
770
771             self.input[i] = 0.
772
773             self.t_last_spike[i] = _t
774
775             self.V[i] = V
776
777             self.w[i] = w
778
779             self.refractory[i] = refractory
780
781             self.input[i] = 0.
782
783             self.t_last_spike[i] = _t
784
785             self.V[i] = V
786
787             self.w[i] = w
788
789             self.refractory[i] = refractory
790
791             self.input[i] = 0.
792
793             self.t_last_spike[i] = _t
794
795             self.V[i] = V
796
797             self.w[i] = w
798
799             self.refractory[i] = refractory
800
801             self.input[i] = 0.
802
803             self.t_last_spike[i] = _t
804
805             self.V[i] = V
806
807             self.w[i] = w
808
809             self.refractory[i] = refractory
810
811             self.input[i] = 0.
812
813             self.t_last_spike[i] = _t
814
815             self.V[i] = V
816
817             self.w[i] = w
818
819             self.refractory[i] = refractory
820
821             self.input[i] = 0.
822
823             self.t_last_spike[i] = _t
824
825             self.V[i] = V
826
827             self.w[i] = w
828
829             self.refractory[i] = refractory
830
831             self.input[i] = 0.
832
833             self.t_last_spike[i] = _t
834
835             self.V[i] = V
836
837             self.w[i] = w
838
839             self.refractory[i] = refractory
840
841             self.input[i] = 0.
842
843             self.t_last_spike[i] = _t
844
845             self.V[i] = V
846
847             self.w[i] = w
848
849             self.refractory[i] = refractory
850
851             self.input[i] = 0.
852
853             self.t_last_spike[i] = _t
854
855             self.V[i] = V
856
857             self.w[i] = w
858
859             self.refractory[i] = refractory
860
861             self.input[i] = 0.
862
863             self.t_last_spike[i] = _t
864
865             self.V[i] = V
866
867             self.w[i] = w
868
869             self.refractory[i] = refractory
870
871             self.input[i] = 0.
872
873             self.t_last_spike[i] = _t
874
875             self.V[i] = V
876
877             self.w[i] = w
878
879             self.refractory[i] = refractory
880
881             self.input[i] = 0.
882
883             self.t_last_spike[i] = _t
884
885             self.V[i] = V
886
887             self.w[i] = w
888
889             self.refractory[i] = refractory
890
891             self.input[i] = 0.
892
893             self.t_last_spike[i] = _t
894
895             self.V[i] = V
896
897             self.w[i] = w
898
899             self.refractory[i] = refractory
900
901             self.input[i] = 0.
902
903             self.t_last_spike[i] = _t
904
905             self.V[i] = V
906
907             self.w[i] = w
908
909             self.refractory[i] = refractory
910
911             self.input[i] = 0.
912
913             self.t_last_spike[i] = _t
914
915             self.V[i] = V
916
917             self.w[i] = w
918
919             self.refractory[i] = refractory
920
921             self.input[i] = 0.
922
923             self.t_last_spike[i] = _t
924
925             self.V[i] = V
926
927             self.w[i] = w
928
929             self.refractory[i] = refractory
930
931             self.input[i] = 0.
932
933             self.t_last_spike[i] = _t
934
935             self.V[i] = V
936
937             self.w[i] = w
938
939             self.refractory[i] = refractory
940
941             self.input[i] = 0.
942
943             self.t_last_spike[i] = _t
944
945             self.V[i] = V
946
947             self.w[i] = w
948
949             self.refractory[i] = refractory
950
951             self.input[i] = 0.
952
953             self.t_last_spike[i] = _t
954
955             self.V[i] = V
956
957             self.w[i] = w
958
959             self.refractory[i] = refractory
960
961             self.input[i] = 0.
962
963             self.t_last_spike[i] = _t
964
965             self.V[i] = V
966
967             self.w[i] = w
968
969             self.refractory[i] = refractory
970
971             self.input[i] = 0.
972
973             self.t_last_spike[i] = _t
974
975             self.V[i] = V
976
977             self.w[i] = w
978
979             self.refractory[i] = refractory
980
981             self.input[i] = 0.
982
983             self.t_last_spike[i] = _t
984
985             self.V[i] = V
986
987             self.w[i] = w
988
989             self.refractory[i] = refractory
990
991             self.input[i] = 0.
992
993             self.t_last_spike[i] = _t
994
995             self.V[i] = V
996
997             self.w[i] = w
998
999             self.refractory[i] = refractory
1000
1001             self.input[i] = 0.
1002
1003             self.t_last_spike[i] = _t
1004
1005             self.V[i] = V
1006
1007             self.w[i] = w
1008
1009             self.refractory[i] = refractory
1010
1011             self.input[i] = 0.
1012
1013             self.t_last_spike[i] = _t
1014
1015             self.V[i] = V
1016
1017             self.w[i] = w
1018
1019             self.refractory[i] = refractory
1020
1021             self.input[i] = 0.
1022
1023             self.t_last_spike[i] = _t
1024
1025             self.V[i] = V
1026
1027             self.w[i] = w
1028
1029             self.refractory[i] = refractory
1030
1031             self.input[i] = 0.
1032
1033             self.t_last_spike[i] = _t
1034
1035             self.V[i] = V
1036
1037             self.w[i] = w
1038
1039             self.refractory[i] = refractory
1040
1041             self.input[i] = 0.
1042
1043             self.t_last_spike[i] = _t
1044
1045             self.V[i] = V
1046
1047             self.w[i] = w
1048
1049             self.refractory[i] = refractory
1050
1051             self.input[i] = 0.
1052
1053             self.t_last_spike[i] = _t
1054
1055             self.V[i] = V
1056
1057             self.w[i] = w
1058
1059             self.refractory[i] = refractory
1060
1061             self.input[i] = 0.
1062
1063             self.t_last_spike[i] = _t
1064
1065             self.V[i] = V
1066
1067             self.w[i] = w
1068
1069             self.refractory[i] = refractory
1070
1071             self.input[i] = 0.
1072
1073             self.t_last_spike[i] = _t
1074
1075             self.V[i] = V
1076
1077             self.w[i] = w
1078
1079             self.refractory[i] = refractory
1080
1081             self.input[i] = 0.
1082
1083             self.t_last_spike[i] = _t
1084
1085             self.V[i] = V
1086
1087             self.w[i] = w
1088
1089             self.refractory[i] = refractory
1090
1091             self.input[i] = 0.
1092
1093             self.t_last_spike[i] = _t
1094
1095             self.V[i] = V
1096
1097             self.w[i] = w
1098
1099             self.refractory[i] = refractory
1100
1101             self.input[i] = 0.
1102
1103             self.t_last_spike[i] = _t
1104
1105             self.V[i] = V
1106
1107             self.w[i] = w
1108
1109             self.refractory[i] = refractory
1110
1111             self.input[i] = 0.
1112
1113             self.t_last_spike[i] = _t
1114
1115             self.V[i] = V
1116
1117             self.w[i] = w
1118
1119             self.refractory[i] = refractory
1120
1121             self.input[i] = 0.
1122
1123             self.t_last_spike[i] = _t
1124
1125             self.V[i] = V
1126
1127             self.w[i] = w
1128
1129             self.refractory[i] = refractory
1130
1131             self.input[i] = 0.
1132
1133             self.t_last_spike[i] = _t
1134
1135             self.V[i] = V
1136
1137             self.w[i] = w
1138
1139             self.refractory[i] = refractory
1140
1141             self.input[i] = 0.
1142
1143             self.t_last_spike[i] = _t
1144
1145             self.V[i] = V
1146
1147             self.w[i] = w
1148
1149             self.refractory[i] = refractory
1150
1151             self.input[i] = 0.
1152
1153             self.t_last_spike[i] = _t
1154
1155             self.V[i] = V
1156
1157             self.w[i] = w
1158
1159             self.refractory[i] = refractory
1160
1161             self.input[i] = 0.
1162
1163             self.t_last_spike[i] = _t
1164
1165             self.V[i] = V
1166
1167             self.w[i] = w
1168
1169             self.refractory[i] = refractory
1170
1171             self.input[i] = 0.
1172
1173             self.t_last_spike[i] = _t
1174
1175             self.V[i] = V
1176
1177             self.w[i] = w
1178
1179             self.refractory[i] = refractory
1180
1181             self.input[i] = 0.
1182
1183             self.t_last_spike[i] = _t
1184
1185             self.V[i] = V
1186
1187             self.w[i] = w
1188
1189             self.refractory[i] = refractory
1190
1191             self.input[i] = 0.
1192
1193             self.t_last_spike[i] = _t
1194
1195             self.V[i] = V
1196
1197             self.w[i] = w
1198
1199             self.refractory[i] = refractory
1200
1201             self.input[i] = 0.
1202
1203             self.t_last_spike[i] = _t
1204
1205             self.V[i] = V
1206
1207             self.w[i] = w
1208
1209             self.refractory[i] = refractory
1210
1211             self.input[i] = 0.
1212
1213             self.t_last_spike[i] = _t
1214
1215             self.V[i] = V
1216
1217             self.w[i] = w
1218
1219             self.refractory[i] = refractory
1220
1221             self.input[i] = 0.
1222
1223             self.t_last_spike[i] = _t
1224
1225             self.V[i] = V
1226
1227             self.w[i] = w
1228
1229             self.refractory[i] = refractory
1230
1231             self.input[i] = 0.
1232
1233             self.t_last_spike[i] = _t
1234
1235             self.V[i] = V
1236
1237             self.w[i] = w
1238
1239             self.refractory[i] = refractory
1240
1241             self.input[i] = 0.
1242
1243             self.t_last_spike[i] = _t
1244
1245             self.V[i] = V
1246
1247             self.w[i] = w
1248
1249             self.refractory[i] = refractory
1250
1251             self.input[i] = 0.
1252
1253             self.t_last_spike[i] = _t
1254
1255             self.V[i] = V
1256
1257             self.w[i] = w
1258
1259             self.refractory[i] = refractory
1260
1261             self.input[i] = 0.
1262
1263             self.t_last_spike[i] = _t
1264
1265             self.V[i] = V
1266
1267             self.w[i] = w
1268
1269             self.refractory[i] = refractory
1270
1271             self.input[i] = 0.
1272
1273             self.t_last_spike[i] = _t
1274
1275             self.V[i] = V
1276
1277             self.w[i] = w
1278
1279             self.refractory[i] = refractory
1280
1281             self.input[i] = 0.
1282
1283             self.t_last_spike[i] = _t
1284
1285             self.V[i] = V
1286
1287             self.w[i] = w
1288
1289             self.refractory[i] = refractory
1290
1291             self.input[i] = 0.
1292
1293             self.t_last_spike[i] = _t
1294
1295             self.V[i] = V
1296
1297             self.w[i] = w
1298
1299             self.refractory[i] = refractory
1300
1301             self.input[i] = 0.
1302
1303             self.t_last_spike[i] = _t
1304
1305             self.V[i] = V
1306
1307             self.w[i] = w
1308
1309             self.refractory[i] = refractory
1310
1311             self.input[i] = 0.
1312
1313             self.t_last_spike[i] = _t
1314
1315             self.V[i] = V
1316
1317             self.w[i] = w
1318
1319             self.refractory[i] = refractory
1320
1321             self.input[i] = 0.
1322
1323             self.t_last_spike[i] = _t
1324
1325             self.V[i] = V
1326
1327             self.w[i] = w
1328
1329             self.refractory[i] = refractory
1330
1331             self.input[i] = 0.
1332
1333             self.t_last_spike[i] = _t
1334
1335             self.V[i] = V
1336
1337             self.w[i] = w
1338
1339             self.refractory[i] = refractory
1340
1341             self.input[i] = 0.
1342
1343             self.t_last_spike[i] = _t
1344
1345             self.V[i] = V
1346
1347             self.w[i] = w
1348
1349             self.refractory[i] = refractory
1350
1351             self.input[i] = 0.
1352
1353             self.t_last_spike[i] = _t
1354
1355             self.V[i] = V
1356
1357             self.w[i] = w
1358
1359             self.refractory[i] = refractory
1360
1361             self.input[i] = 0.
1362
1363             self.t_last_spike[i] = _t
1364
1365             self.V[i] = V
1366
1367             self.w[i] = w
1368
1369             self.refractory[i] = refractory
1370
1371             self.input[i] = 0.
1372
1373             self.t_last_spike[i] = _t
1374
1375             self.V[i] = V
1376
1377             self.w[i] = w
1378
1379             self.refractory[i] = refractory
1380
1381             self.input[i] = 0.
1382
1383             self.t_last_spike[i] = _t
1384
1385             self.V[i] = V
1386
1387             self.w[i] = w
1388
1389             self.refractory[i] = refractory
1390
1391             self.input[i] = 0.
1392
1393             self.t_last_spike[i] = _t
1394
1395             self.V[i] = V
1396
1397            
```



### 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.

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

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

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

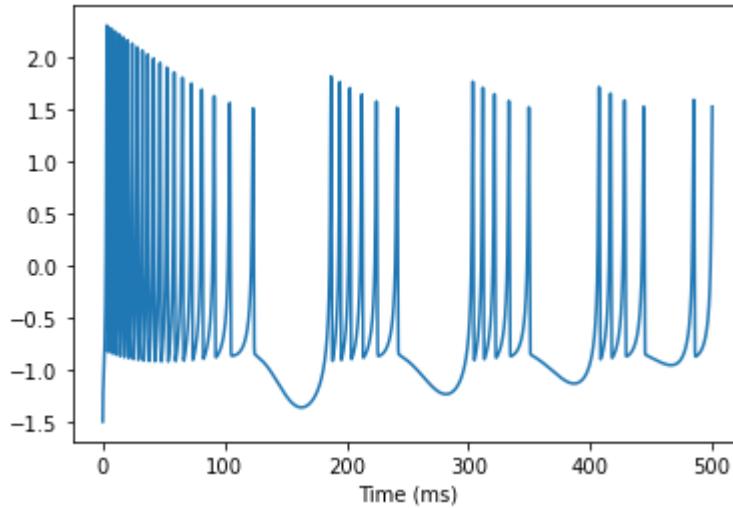
The  $V$  variable refers to membrane potential, and  $y, z$  are two gating variables.

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.

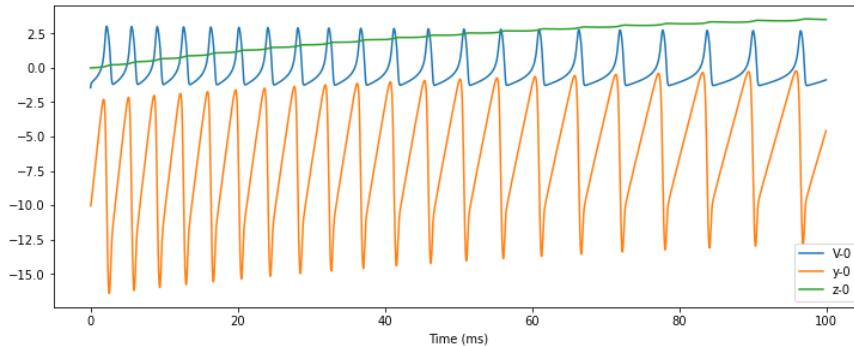
```

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 - Vrest) - 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, → For each neuron in neuron group.
38                 self.a, self.b, self.input[i], → Update variables with
39                 self.c, self.d, self.r, self.s, → numerical integration
40                 self.V_rest) → one by one.
41             self.V[i] = V
42             self.input[i] = 0. → Reset external input
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
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
698
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
826
827
828
828
829
829
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
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
15
```

## 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.

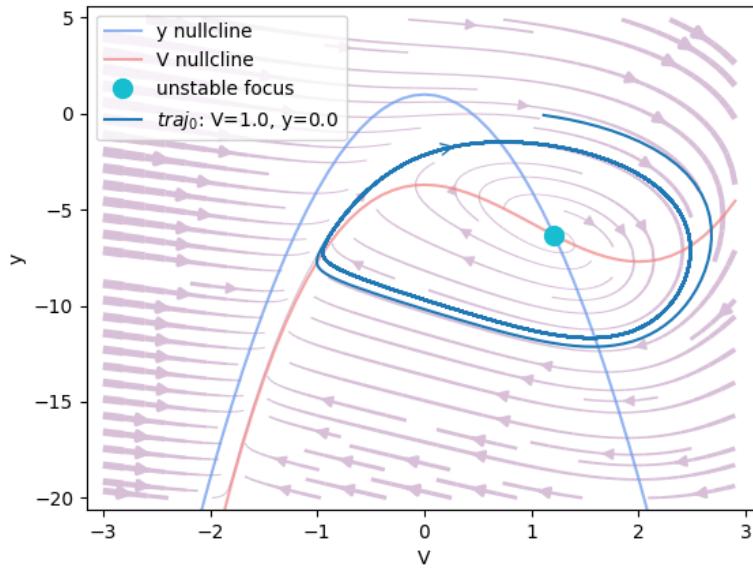


With the theoretical analysis module `analysis` of BrainPy, we may explain the existence of this periodicity through theoretical analysis. In Hindmarsh-Rose model, the trajectory of  $V$  and  $y$  approaches a limit cycle in phase plane, therefore their values change periodically along the limit cycle.

```

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.,
74                  'c': 1., 'd': 5., 'r': 0.01, 's': 4.,
75                  'V_rest': -1.6}                         ──────────> Other parameters to be fixed.
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(
81      [{"V": 1., "y": 0., "z": -0.}],
82      duration=100.,
83      show=True
84  )                                              ──────────> Plot trajectory.
                                                    Define start point of trajectory and
                                                    simulation duration.

```



### 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,  $\tau$  is time constant,  $V$  is membrane potential,  $V_{rest}$  is resting potential,  $R$  is conductance, and  $I$  is external input.

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

## 1.1 Biological background

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 →  $dI_1/dt = -k_1 I_1$ 
9          dI2dt = - k2 * I2 →  $dI_2/dt = -k_2 I_2$ 
10         dVthdt = a * (V - V_rest) - b * (V_th - V_th_inf) →  $dV_{th}/dt = a(V - V_{rest}) - b(V_{th} - V_{th_\infty})$ 
11         dVdt = (- (V - V_rest) + R * I_ext + R * I1 + R * I2) / tau →  $\tau dV/dt = -(V - V_{rest}) + R \sum I_j + RI(t),$ 
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.

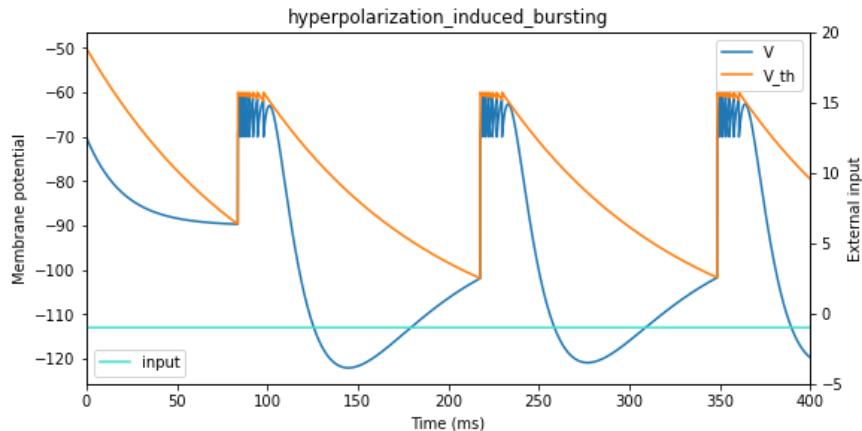
→ Pass 'size' and '\*\*kwargs' to superclass bp.NeuGroup's constructor.

## 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. In these models, each compute unit represents a neuron group, the membrane potential variable  $V$  in single neuron models is replaced by firing rate variable  $a$  (or  $r$  or  $\nu$ ). Here we introduce a canonical firing rate unit.

### 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,  $\tau_x$  refers to the time constant of neuron columns, parameters  $k_x$  and  $r_x$  control the refractory periods,  $a_x$  and  $\theta_x$  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         x_ai = 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

```

$$\begin{aligned}\tau_e \frac{da_e}{dt} &= -a_e + (k_e - r_e * a_e) * S_e(c_1 a_e - c_2 a_i + I_{ext_e}) \\ \tau_i \frac{da_i}{dt} &= -a_i + (k_i - r_i * a_i) * S_i(c_3 a_e - c_4 a_i + I_{ext_i})\end{aligned}$$

## 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         self.input_e[:] = 0.   → Reset external input
61         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

When we model the firing of neurons, we need to connect them. Synapse is very important for 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

In the previous section, we learned how to model neurons and their action potentials. In this section, we will focus on how neurons communicate.

### 2.1.1 Chemical Synapses

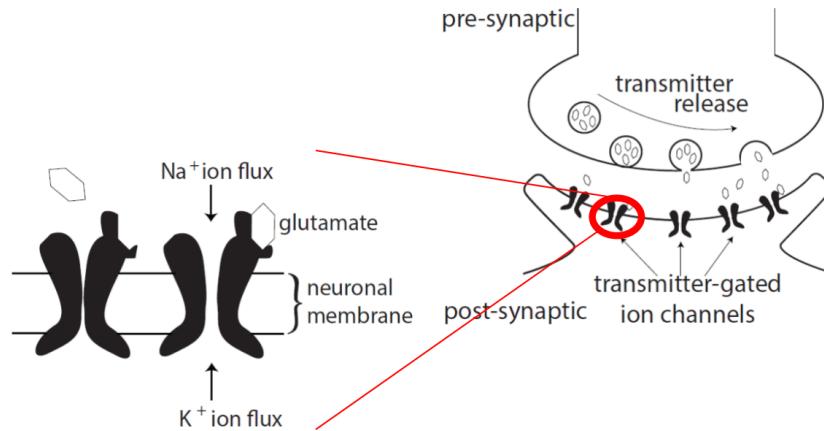
#### Biological Background

Fig. 2-1 shows the biological process of information transmission between neurons. The action potential of the presynaptic neuron makes the axon terminal release **neurotransmitters** (also called transmitters) into the synaptic cleft, and then the membrane potential of the postsynaptic cell changes after a brief delay. These changes are called postsynaptic potentials (PSP), and they can be either excitatory or inhibitory depending on the type of transmitter. **Glutamate** is one of the important excitatory neurotransmitters, and Gamma-aminobutyric acid (**GABA**) is one of the important inhibitory neurotransmitters.

Neurotransmitters affect their targets by interacting with receptors on the postsynaptic membrane. When the transmitter binds to the receptor, it would either open an ion channel (**ionotropic** receptors) or alter chemical reactions within the target cell (**metabotropic** receptors).

In this section, we will introduce how to model some common synapses and their implementations with `BrainPy`:

- **AMPA** and **NMDA** receptors are both ionotropic receptors of Glutamate, but the NMDA receptor are typically blocked by magnesium ions ( $Mg^{2+}$ ) and cannot respond to the glutamate. With repeated activation of AMPA receptors, the change in postsynaptic potential drives  $Mg^{2+}$  out of NMDA channel, then the NMDA receptors are able to respond to glutamate. Therefore, the dynamics of NMDA is much slower than that of AMPA.
- **GABA<sub>A</sub>** and **GABA<sub>B</sub>** are two classes of GABA receptors. GABA<sub>A</sub> receptors are ionotropic, typically producing fast inhibitory postsynaptic potential; while GABA<sub>B</sub> receptors are metabotropic receptors, typically producing a slow-occurring inhibitory postsynaptic potential.



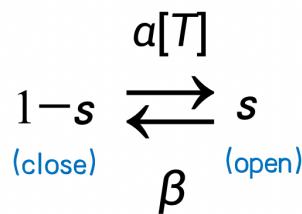
**Fig. 2-1 Biological Synapse** (Adapted from Gerstner et al., 2014 [1](#))

In order to keep things simple, we use gating variable  $s$  to describe how many portion of ion channels will open whenever a presynaptic spike arrives while modeling. We will first introduce AMPA receptor as an example to show how to develop synapse models and implement them 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  $\text{Na}^+$  and  $\text{K}^+$  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  $\alpha$  and  $\beta$  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.

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

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

## 1.1 Biological background

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):
9         ds = alpha * TT * (1 - s) - beta * s    }  $\frac{ds}{dt} = \alpha[T](1-s) - \beta s$ 
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') } Specify how the
23         self.size = len(self.pre_ids)   presynaptic and
24
25         # variables
26         self.s = bp.ops.zeros(self.size)
27         self.t_last_pre_spike = -1e7 * bp.ops.ones(self.size)   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)
31

```

Specify how the presynaptic and postsynaptic neuron groups connect to each other

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

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]:
38                 self.t_last_pre_spike[pre_id] = _t
39                 TT = ((_t - self.t_last_pre_spike[pre_id])      } TT denotes [T], TT=T if pre neuron
40                     < self.T_duration) * self.T } spikes within T_duration,
41             self.s[i] = self.int_s(self.s[i], _t, TT, self.alpha, self.beta)   otherwise TT=0.
42

```

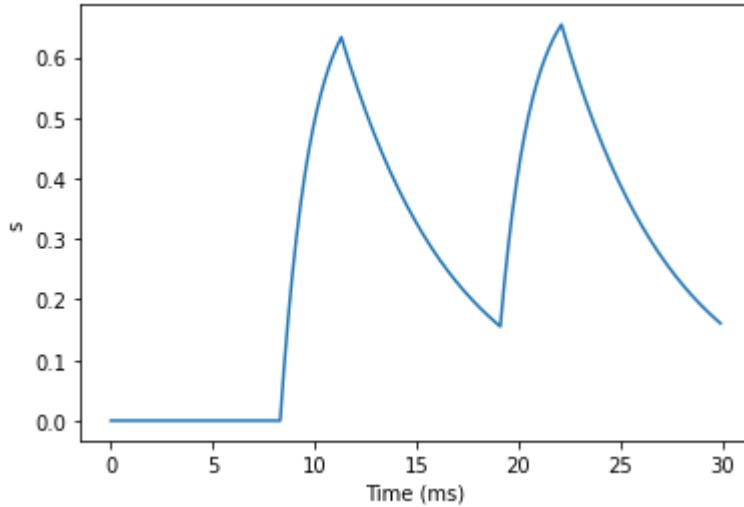
After the implementation, we can plot the graph of  $s$  changing with time. We would first write a `run_syn` function to run and plot the graph. To run a synapse, we need neuron groups, so we use the LIF neuron provided by `brainmodels` package.

```

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']) } Get two LIF neuron groups from
51     neu2 = bm.neurons.LIF(3, monitors=['V']) } brainmodels package
52
53     syn = syn_model(pre=neu1, post=neu2, conn=bp.connect.All2All(),
54                      monitors=['s'], **kwargs) } Specify pre and post neurons and
55
56     net = bp.Network(neu1, syn, neu2)           there connections. Here we use all to
57     net.run(30., inputs=(neu1, 'input', 35.))   all connections provided by BrainPy.
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.

```

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.

## Alpha、Exponential Synapses

Because many synaptic models have the same dynamic characteristics as AMPA synapses, 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. These models can also be found in the `Brain-Models` package.

### (1) Differences of two exponentials

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

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

Where  $t_s$  denotes the spike timing of the presynaptic neuron,  $\tau_1$  and  $\tau_2$  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:

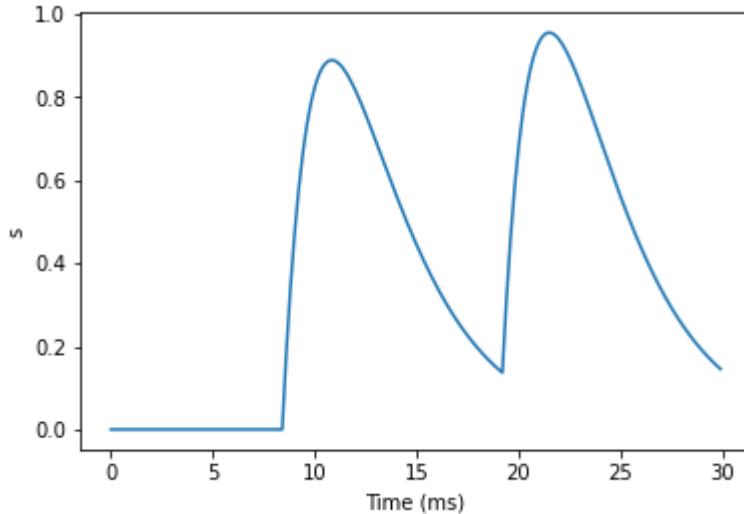
## 1.1 Biological background

```

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)    }    $\frac{dx}{dt} = (-(\tau_1 + \tau_2)x - s)/(\tau_1\tau_2)$ 
7          dsdt = x                                         }    $\frac{ds}{dt} = x$ 
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]
35
36             if (pre spike), then x = x + 1
37
38 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  $\tau$ .

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:

## 1.1 Biological background

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

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

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

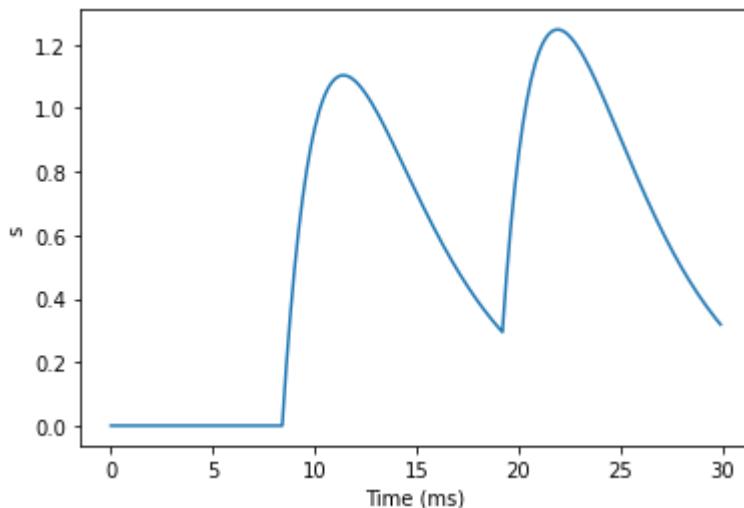
Code implementation is similar:

```

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]
34
35             if (pre spike), then x = x + 1
36
37 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:

## 1.1 Biological background

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

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

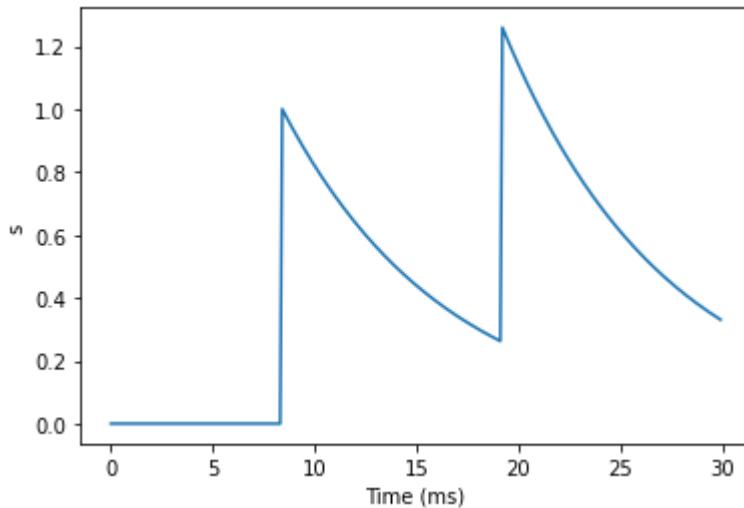
The implementing code is given by:

```

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

Sometimes even the decay process can be ignored, so there is a `voltage_jump` model, which is given by:

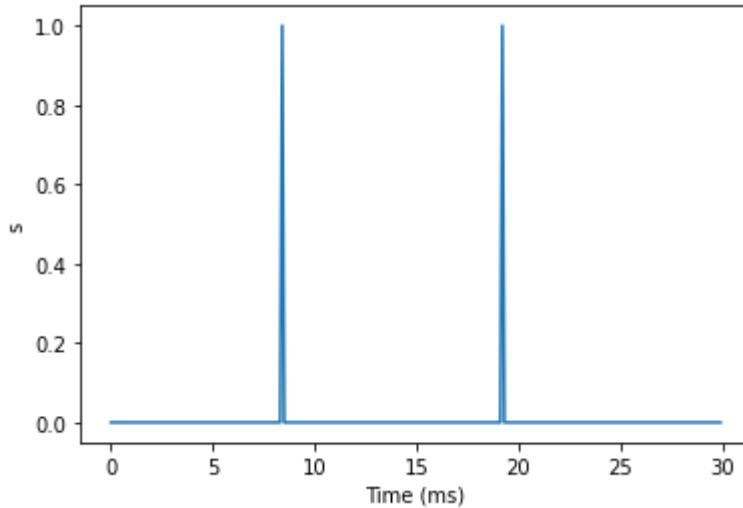
if (fire), then  $s \leftarrow s + 1$

The code is as follows:

## 1.1 Biological background

```
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:



## Current-based and Conductance-based synapses

So far, we have modeled the gating variable  $s$ , now let's see how to model the effect of the gating variables on the synaptic current. The current that passes through a synaptic channel is denoted as  $I$ . 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 `t_syn` variable using the `register_constant_delay` function provided by BrainPy.

## 1.1 Biological background

```

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, delay_time=delay) ——> set delay time before
6                                         changing synaptic current
7                                         by using the
8                                         register_constant_delay
9                                         method
10    def update(self, _t):
11        for i in range(self.size):
12            # ...
13            self.I_syn.push(i, self.w[i] * self.s[i]) ] use push and pull to set delay before applying
14            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  $g$ .

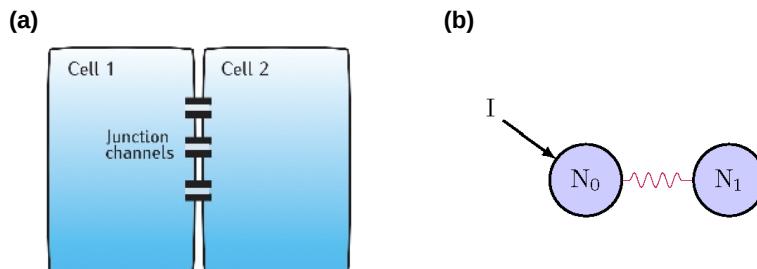
```

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
15

```

### 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 Sterratt et al., 2011<sup>2</sup>)

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.

```

6   class Gap_junction(bp.TwoEndConn):
7       target_backend = ['numpy', 'numba']
8
9       def __init__(self, pre, post, conn, delay=0., k_spikelet=0.1,
10                  post_refractory=False, **kwargs):
11           self.delay = delay
12           self.k_spikelet = k_spikelet
13           self.post_has_refractory = post_refractory
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.w = bp.ops.ones(self.size)
22           self.spikelet = self.register_constant_delay('spikelet', size=self.size,
23                                                       delay_time=self.delay)
23
24
25       super(Gap_junction, 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               post_id = self.post_ids[i]
31
32               self.post.input[post_id] += self.w[i] * (self.pre.V[pre_id] -
33                                                       self.post.V[post_id])  #  $I_{post} = w(V_{pre} - V_{post})$ 
34
35               self.spikelet.push(i, self.w[i] * self.k_spikelet *
36                                 self.pre.spike[pre_id])
37
38               out = self.spikelet.pull(i)
39               if self.post_has_refractory:
40                   self.post.V[post_id] += out * (1. -
41                                                 self.post.refractory[post_id])
42               else:
43                   self.post.V[post_id] += out
44

```

Then we can run a simulation.

## 1.1 Biological background

```
import matplotlib.pyplot as plt
import numpy as np

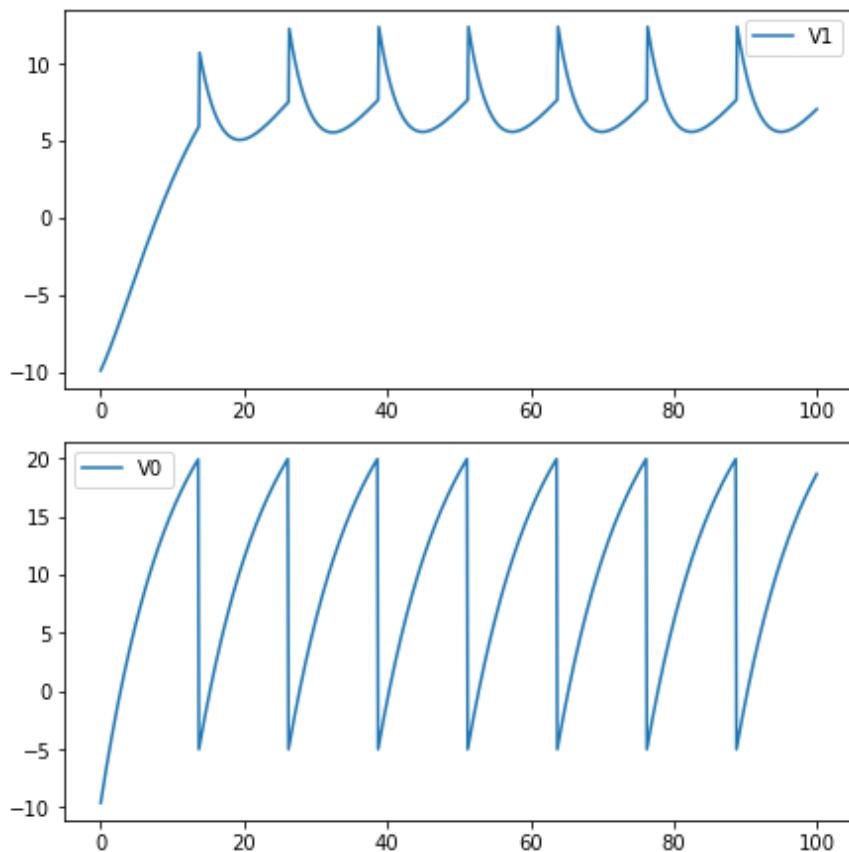
neu0 = bm.neurons.LIF(2, monitors=['V'], t_refractory=0)
neu0.V = np.ones(neu0.V.shape) * -10.
neu1 = bm.neurons.LIF(3, monitors=['V'], t_refractory=0)
neu1.V = np.ones(neu1.V.shape) * -10.
syn = Gap_junction(k_spikelet=5., pre=neu0, post=neu1,
                     conn=bp.connect.All2All())
syn.w = np.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

- <sup>1</sup>. Gerstner, Wulfram, et al. Neuronal dynamics: From single neurons to networks and models of cognition. Cambridge University Press, 2014. ↵

~

## 1.1 Biological background

<sup>2</sup>. Sterratt, David, et al. Principles of computational modeling 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.

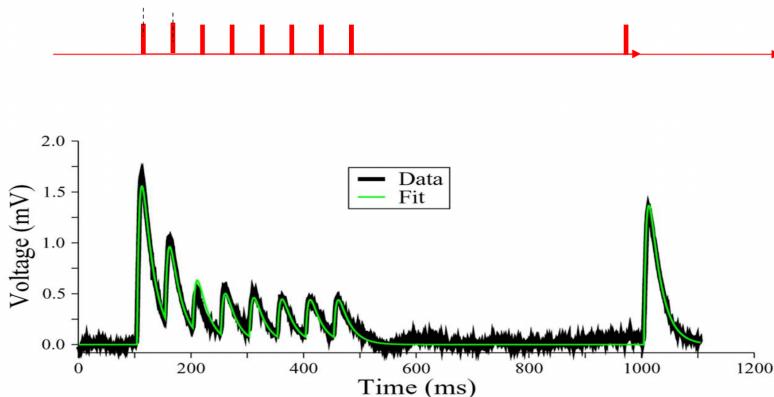
Plasticity mainly distinguishes short-term plasticity from long-term plasticity. We will first introduce short-term plasticity (STP), and then introduce several different models of long-term synaptic plasticity (also known as learning rules).

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. 2-1 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 fire 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. 2-1 Short-term plasticity.** (Adapted from Gerstner et al., 2014 <sup>1</sup>)

Now let's turn to the model. 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,

## 1.1 Biological background

$$\frac{dI}{dt} = -\frac{I}{\tau}$$

$$\frac{du}{dt} = -\frac{u}{\tau_f}$$

$$\frac{dx}{dt} = \frac{1-x}{\tau_d}$$

$$\text{if (pre fire), then } \begin{cases} u^+ = u^- + U(1-u^-) \\ I^+ = I^- + Au^+x^- \\ x^+ = x^- - u^+x^- \end{cases}$$

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.  $\tau_f$  and  $\tau_d$  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  $\tau_f$  and  $\tau_d$  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  class STP(bp.TwoEndConn):
2      target_backend = ['numpy', 'numba']
3
4      @staticmethod
5      def derivative(s, u, x, t, tau, tau_d, tau_f):
6          dsdt = -s / tau
7          dudt = - u / tau_f
8          dxdt = (1 - x) / tau_d
9          return dsdt, dudt, dxdt
10
11     def __init__(self, pre, post, conn, delay=0., U=0.15, tau_f=1500.,
12                  tau_d=200., tau=8., **kwargs):
13         # parameters
14         self.tau_d = tau_d
15         self.tau_f = tau_f
16         self.tau = tau
17         self.U = U
18         self.delay = delay
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)
24
25         # variables
26         self.s = bp.ops.zeros(self.size)
27         self.x = bp.ops.ones(self.size)
28         self.u = bp.ops.zeros(self.size)
29         self.w = bp.ops.ones(self.size)
30         self.I_syn = self.register_constant_delay('I_syn', size=self.size,
31                                               delay_time=delay)
32
33         self.integral = bp.odeint(f=self.derivative, method='exponential_euler')
34
35     super(STP, self).__init__(pre=pre, post=post, **kwargs)
36

```

## 1.1 Biological background

```

37     def update(self, _t):
38         for i in range(self.size):
39             pre_id = self.pre_ids[i]
40
41             self.s[i], u, x = self.integral(self.s[i], self.u[i], self.x[i], _t,
42                                             self.tau, self.tau_d, self.tau_f)
43
44             if self.pre.spike[pre_id] > 0:
45                 u += self.U * (1 - self.u[i])
46                 self.s[i] += self.w[i] * u * self.x[i]
47                 x -= u * self.x[i]
48             self.u[i] = u
49             self.x[i] = x
50
51             # output
52             post_id = self.post_ids[i]
53             self.I_syn.push(i, self.s[i])
54             self.post.input[post_id] += self.I_syn.pull(i)
55

```

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.

```

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()

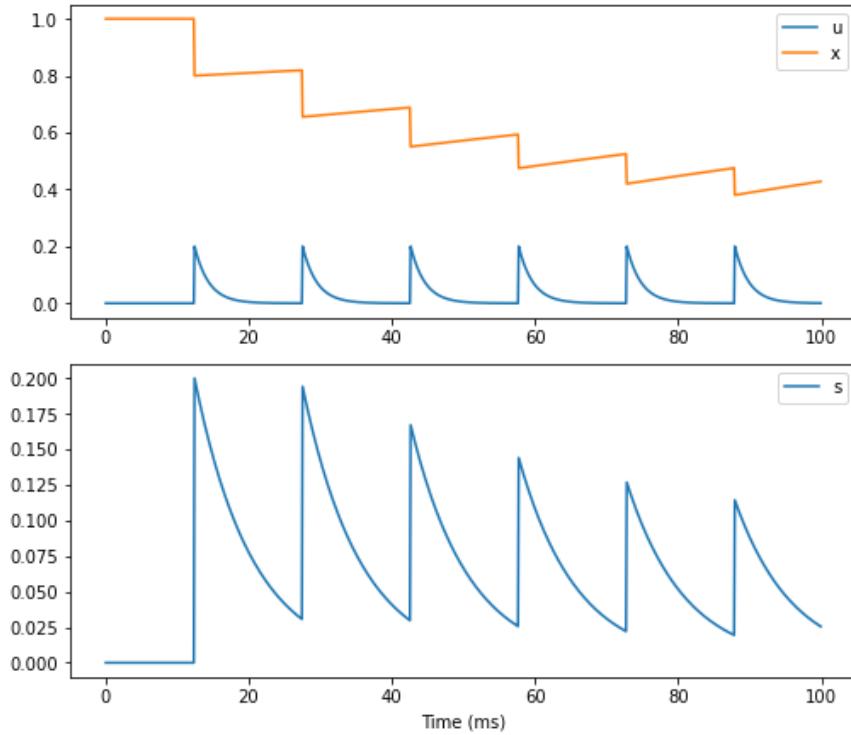
    plt.xlabel('Time (ms)')
    plt.show()

```

Let's first set `tau_d > tau_f`.

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

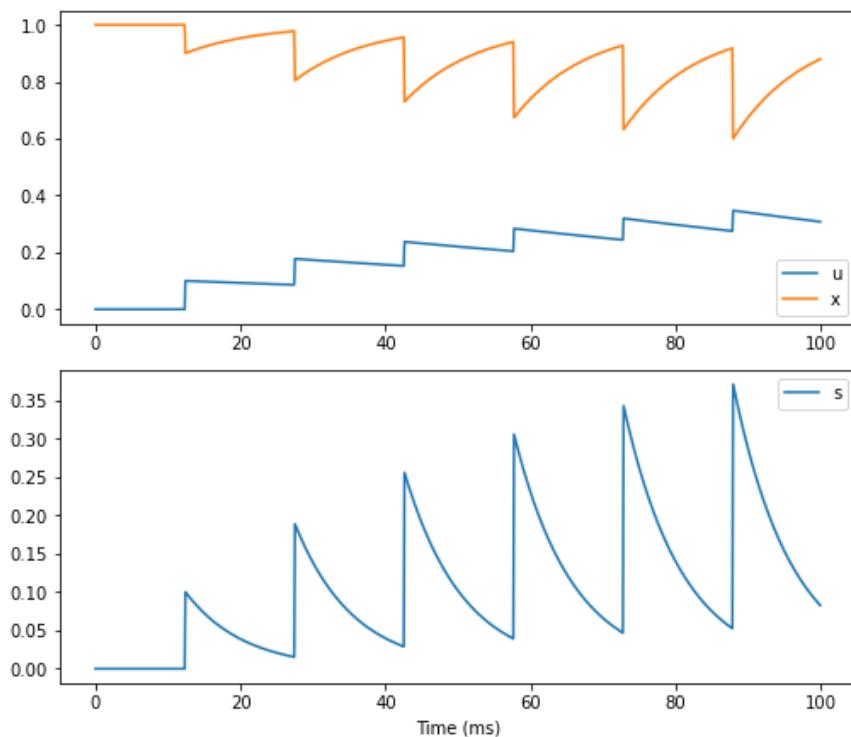
## 1.1 Biological background



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.)
```

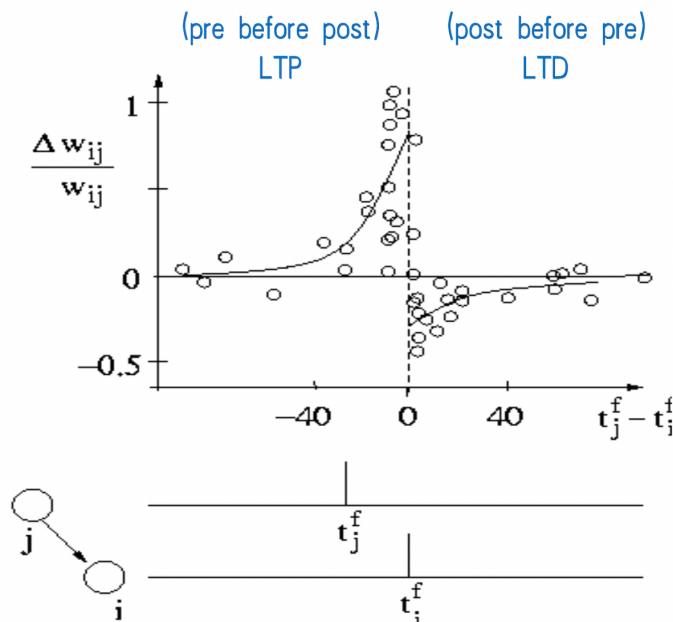


We can see from the figure that when we set  $\tau_f > \tau_d$ , on the contrary, every time  $x$  is used, it will be added back quickly. There are always enough transmitters available. At the same time, the decay of  $u$  is very slow, so the probability of releasing transmitters is getting higher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

## 2.2.2 Long-term Plasticity

### Spike-timing dependent plasticity (STDP)

Fig. 2-2 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. 2-2 Spike timing dependent plasticity.** (Adapted from Bi & Poo, 2001<sup>2</sup>)

The computational model of STDP is given by,

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

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

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

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

## 1.1 Biological background

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.  $\Delta A_s$  and  $\Delta A_t$  are the increments of  $A_s$  and  $A_t$ , respectively.  $\tau_s$  and  $\tau_t$  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.

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  $s$ .

```

1  class STDP(bp.TwoEndConn):
2      target_backend = ['numpy', 'numba']
3
4      @staticmethod
5      def derivative(s, A_s, A_t, t, tau, tau_s, tau_t):
6          dsdt = -s / tau
7          dAsdt = -A_s / tau_s
8          dAtdt = -A_t / tau_t
9          return dsdt, dAsdt, dAtdt
10
11     def __init__(self, pre, post, conn, delay=0., delta_A_s=0.5,
12                  delta_A_t=0.5, w_min=0., w_max=20., tau_s=10., tau_t=10.,
13                  tau=10., **kwargs):
14         # parameters
15         self.tau_s = tau_s
16         self.tau_t = tau_t
17         self.tau = tau
18         self.delta_A_s = delta_A_s
19         self.delta_A_t = delta_A_t
20         self.w_min = w_min
21         self.w_max = w_max
22         self.delay = delay
23
24         # connections
25         self.conn = conn(pre.size, post.size)
26         self.pre_ids, self.post_ids = self.conn.requires('pre_ids', 'post_ids')
27         self.size = len(self.pre_ids)
28
29         # variables
30         self.s = bp.ops.zeros(self.size)
31         self.A_s = bp.ops.zeros(self.size)
32         self.A_t = bp.ops.zeros(self.size)
33         self.w = bp.ops.ones(self.size) * 1.
34         self.I_syn = self.register_constant_delay('I_syn', size=self.size,
35                                               delay_time=delay)
36         self.integral = bp.odeint(f=self.derivative, method='exponential_euler')
37
38     super(STDP, self).__init__(pre=pre, post=post, **kwargs)
39

```

## 1.1 Biological background

```

40     def update(self, _t):
41         for i in range(self.size):
42             pre_id = self.pre_ids[i]
43             post_id = self.post_ids[i]
44
45             self.s[i], A_s, A_t = self.integral(self.s[i], self.A_s[i],
46                                               self.A_t[i], _t, self.tau,
47                                               self.tau_s, self.tau_t)
48
49             w = self.w[i]
50             if self.pre.spike[pre_id] > 0:   }   if (pre fire), then
51                 self.s[i] += w               }    $\begin{cases} s \leftarrow s + w \\ A_s \leftarrow A_s + \Delta A_s \\ w \leftarrow w - A_t \end{cases}$ 
52                 A_s += self.delta_A_s
53                 w -= A_t
54
55             if self.post.spike[post_id] > 0:  }   if (post fire), then
56                 A_t += self.delta_A_t      }    $\begin{cases} A_t \leftarrow A_t + \Delta A_t \\ w \leftarrow w + A_s \end{cases}$ 
57                 w += A_s
58
59             self.A_s[i] = A_s
60             self.A_t[i] = A_t
61
62             self.w[i] = bp.ops.clip(w, self.w_min, self.w_max) ----- limit w
63
64             # output
65             self.I_syn.push(i, self.s[i])
66             self.post.input[post_id] += self.I_syn.pull(i)
67

```

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 each two inputs are 15ms. We keep those

$t_{\text{post}} = t_{\text{pre}} + 5$  during the first 3 spike-pairs. Then we set a long interval before

switching the stimulating order to be  $t_{\text{post}} = t_{\text{pre}} - 3$  since the 4th spike.

```

duration = 300.
(I_pre, _) = bp.inputs.constant_current([(0, 5), (30, 15),    # pre at 5ms
                                           (0, 15), (30, 15),
                                           (0, 15), (30, 15),
                                           (0, 98), (30, 15),  # switch order: t_interval=98ms
                                           (0, 15), (30, 15),
                                           (0, 15), (30, 15),
                                           (0, duration-155-98)])
(I_post, _) = bp.inputs.constant_current([(0, 10), (30, 15), # post at 10
                                           (0, 15), (30, 15),
                                           (0, 15), (30, 15),
                                           (0, 90), (30, 15), # switch order: t_interval=98-8=90(ms)
                                           (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, 'input', I_post)])

# 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)

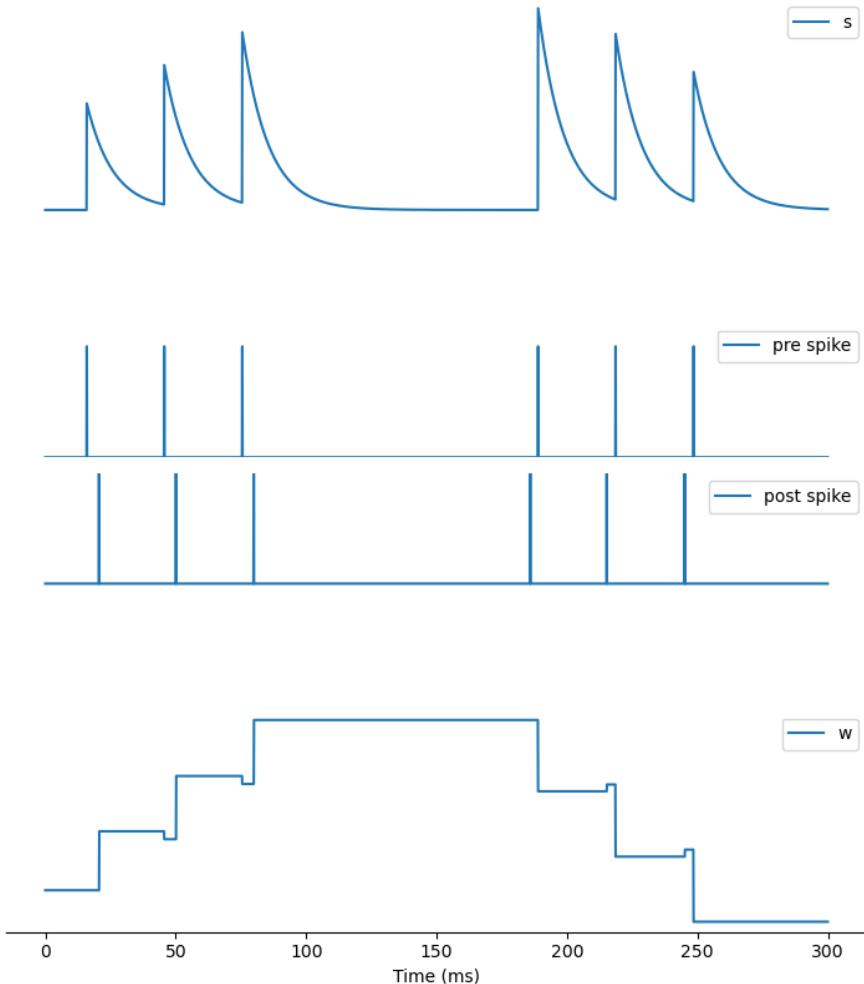
ax=fig.add_subplot(gs[0, 0])
plt.plot(net.ts, syn.mon.s[:, 0], label="s")
hide_spines(ax)

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.

$$\frac{d}{dt}w_{ij} = c_{00}w_{ij} + c_{10}w_{ij}r_j + c_{01}w_{ij}r_i + c_{20}w_{ij}r_j^2 + c_{02}w_{ij}r_i^2 + c_{11}w_{ij}r_ir_j + O(r^3)$$

## 1.1 Biological background

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.

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

$\gamma$  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', 'numba']   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)   }  $\frac{dw}{dt} = \gamma(r_i r_j - w r_i^2)$ 
7          return dwdt
8
9      def __init__(self, pre, post, conn, delay=0,
10                  gamma=0.005, w_max=1., w_min=0.,
11                  **kwargs):
12          # params
13          self.gamma = gamma
14          self.w_max = w_max
15          self.w_min = w_min
16          # no delay in firing rate models
17
18          # connns
19          self.conn = conn(pre.size, post.size)
20          self.pre_ids, self.post_ids = self.conn.requires('pre_ids', 'post_ids')
21          self.size = len(self.pre_ids)
22
23          # data
24          self.w = bp.ops.ones(self.size) * 0.05
25
26          self.integral = bp.odeint(f=self.derivative)
27          super(Oja, self).__init__(pre=pre, post=post, **kwargs)
28
29
30      def update(self, _t):
31          post_r = bp.ops.zeros(self.post.size[0])
32          for i in range(self.size):
33              pre_id = self.pre_ids[i]                                rate model:  $r_i = \sum_j w_{ij} r_j$ 
34              post_id = self.post_ids[i]
35              add = self.w[i] * self.pre.r[pre_id]    ----> let  $add_j = w_{ij} r_j$ 
36              post_r[post_id] += add                ----> then  $r_i = \sum_j add_j$ 
37              self.w[i] = self.integral(
38                  self.w[i], _t, self.gamma,
39                  self.pre.r[pre_id], self.post.r[post_id])    ----> use self.post.r, not post_r, see next line
40              self.post.r = post_r    ---->  $r_i = \sum_j add_j$ 
41
42          Note that the effects of pre groups
43          are simultaneously, so we update
44          post_r after the for loop

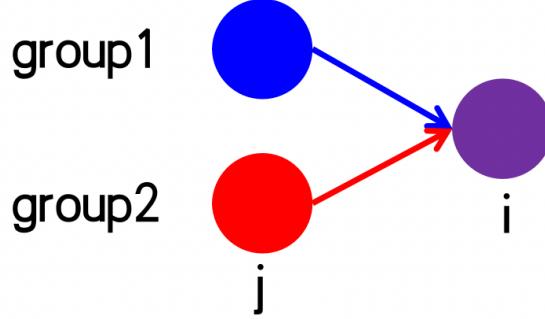
```

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.

```

41  class neu(bp.NeuGroup):
42      target_backend = ['numpy', 'numba']
43
44      def __init__(self, size, **kwargs):
45          self.r = bp.ops.zeros(size)           ----> we need a firing rate neuron with
46          super(neu, self).__init__(size=size, **kwargs) parameter r
47
48      def update(self, _t):
49          self.r = self.r    ----> we only need a simple unit to test oja's

```

**Fig. 2-3 Connection of neuron groups.**

We aim to implement the connection as shown in Fig. 2-3. 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.

```

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

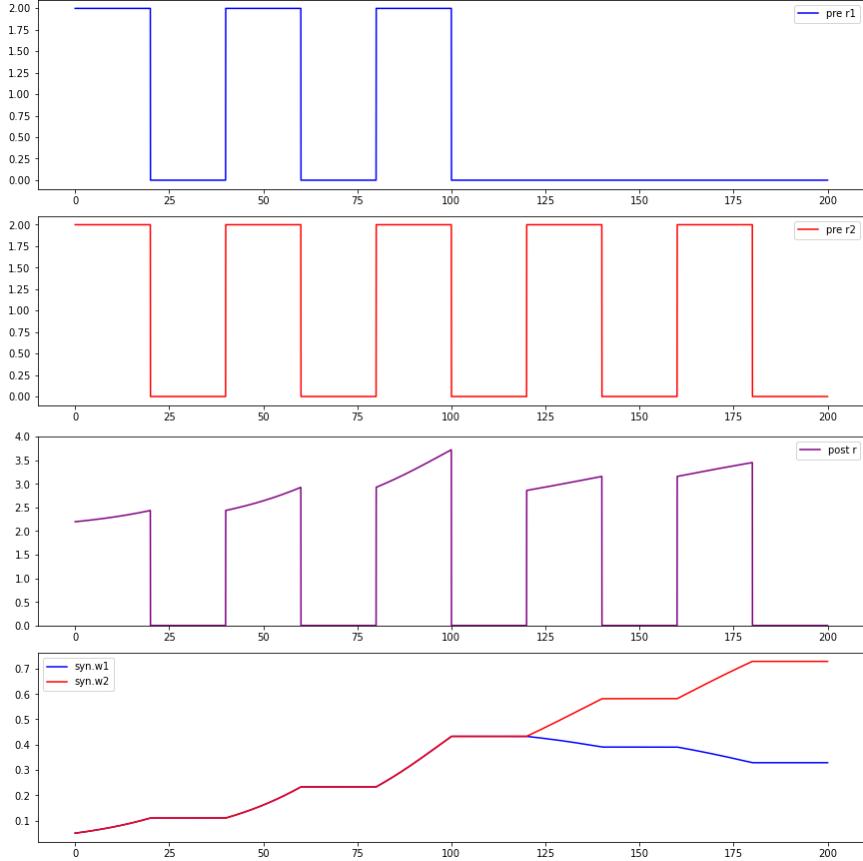
58 # simulate
59 neu_pre = neu(2, monitors=['r'])
60 neu_post = neu(2, monitors=['r'])
61 syn = Oja(pre=neu_pre, post=neu_post, conn=bp.connect.All2All(), monitors=['w'])
62 net = bp.Network(neu_pre, syn, neu_post)
63 net.run(duration=200., inputs=[(neu_pre, 'r', current_pre.T, '='),
64             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

```

Annotations on the right side of the code:

- 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



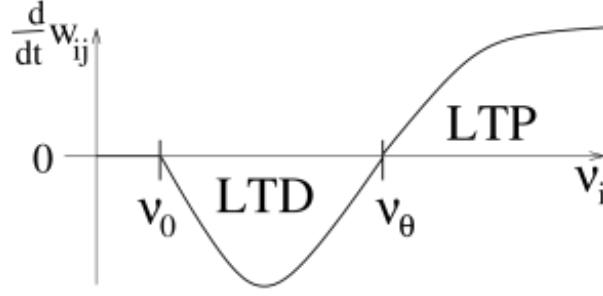
It can be seen from the results that at the beginning, when the two groups of neurons were given input at the same time, their weights increased simultaneously, and the response of post became stronger and stronger, showing LTP. After 100ms, the blue group is no longer fire together, only the red group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

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

where  $\eta$  represents the learning rate, and  $r_\theta$  represents the threshold of learning (see Fig. 2-4). Fig. 2-4 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  $r_\theta$ .



**Fig. 2-4 BCM rule** (From Gerstner et al., 2014 [1](#))

We will implement the same connections as the previous Oja's rule (Fig. 2-3), 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=2., 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._dt
15
16         # connections
17         self.conn = conn(pre.size, post.size)
18         self.pre_ids, self.post_ids = self.conn.requires('pre_ids', 'post_ids')
19         self.size = len(self.pre_ids)
20
21         # variables
22         self.w = bp.ops.ones(self.size)
23         self.sum_post_r = bp.ops.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)           }    $r_\theta = \frac{\sum_t dr_i}{T}$ 
33                                         self.dt+1 here since
34                                         self.dt begin with 0
35
36         # update w and post_r
37         post_r = bp.ops.zeros(self.post.size[0])
38
39         for i in range(self.size):
40             pre_id = self.pre_ids[i]
41             post_id = self.post_ids[i]                         rate model:  $r_i = \sum_j w_{ij} r_j$ 
42             post_r[post_id] += self.w[i] * self.pre.r[pre_id]    let  $add_j = w_{ij} r_j$ 
43             w = self.int_w(self.w[i], _t, self.lr, self.pre.r[pre_id],           then  $r_i = \sum_j add_j$ 
44             self.post.r[post_id], r_th[post_id])                self.post.r = post_r
45
46             self.w[i] = bp.ops.clip(w, self.w_min, self.w_max)    limit w
47
48             self.post.r = post_r                                 $r_i = \sum_j add_j$ 
```

Then we can run the simulation with the following code.

## 1.1 Biological background

```

n_post = 1
n_pre = 20

# group selection
group1, duration = bp.inputs.constant_current(([1.5, 1], [0, 1]) * 20)
group2, duration = bp.inputs.constant_current(([0, 1], [1, 1]) * 20)
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)

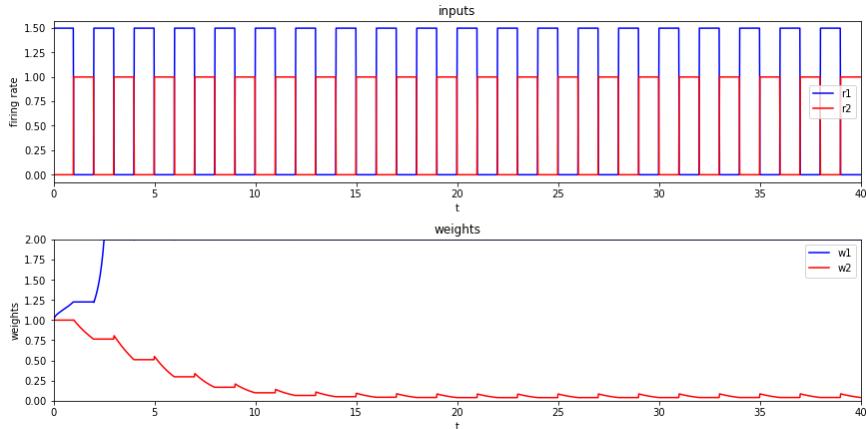
r1 = bp.ops.mean(pre.mon.r[:, :10], 1)
r2 = bp.ops.mean(pre.mon.r[:, 10:], 1)

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()

```



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

- <sup>1</sup>. Gerstner, Wulfram, et al. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014. ↪
- <sup>2</sup>. 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.

### 3.1 Spiking Neural Networks

### 3.2 Firing Rate Networks

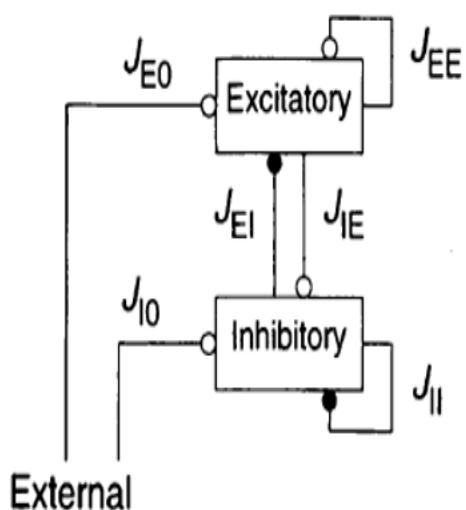
## 3.1 Spiking Neural Network

### 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.

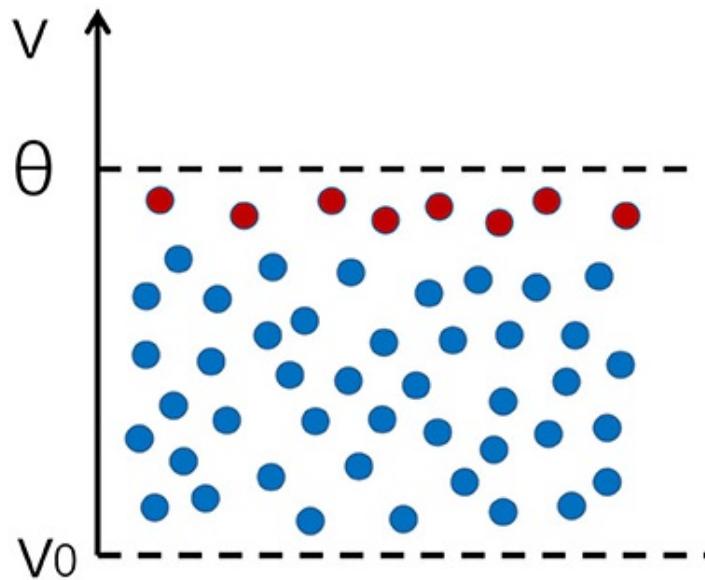
The randomness and noise in E/I balanced network give each neuron in the network an internal input which varies with time and space at the order of threshold potential. Therefore, the firing of neurons also has randomness, ensures that E/I balanced network can generate temporal irregular firing pattern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



**Fig.3-1 Structure of E/I balanced network | Vreeswijk and Sompolinsky, 1996**

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.

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  $\theta$ .



**Fig.3-2 Distribution of neuron membrane potentials in E/I balanced network |  
Tian et al., 2020**

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.

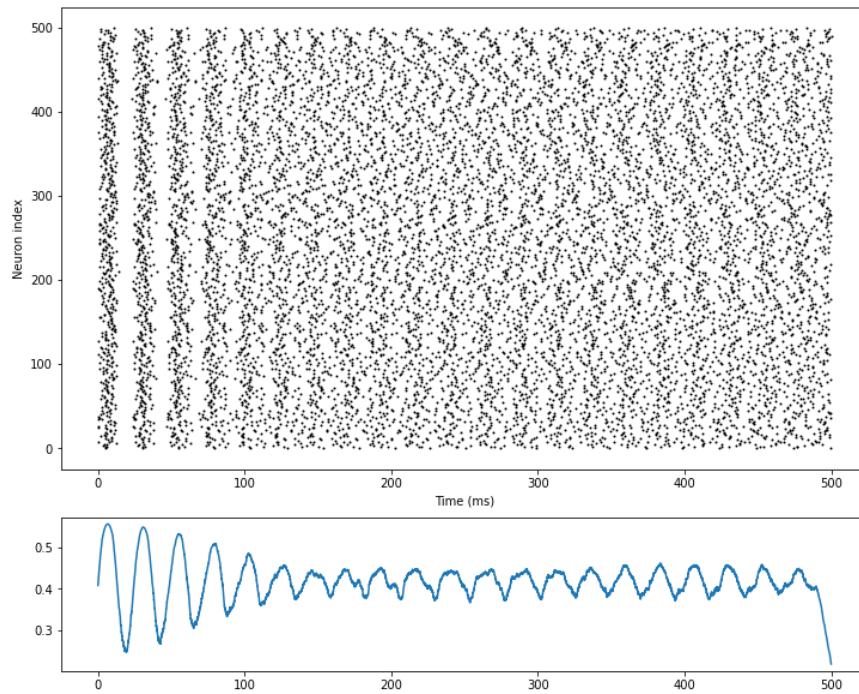


- 3) Inputs: All neurons in the network receive a constant external input current.



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.



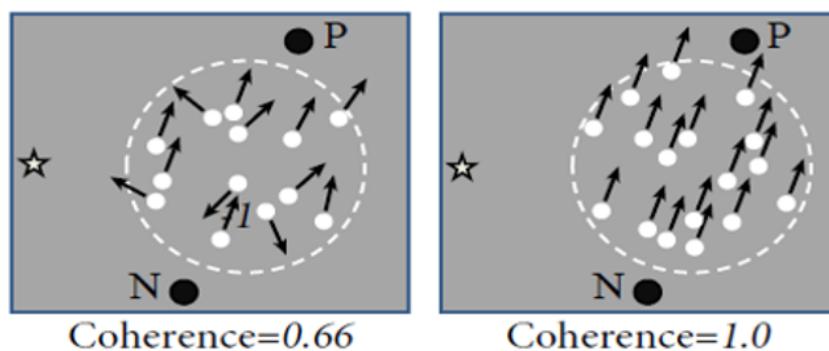


**Fig.3-3 E/I balanced net raster plot**

### 3.1.2 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.



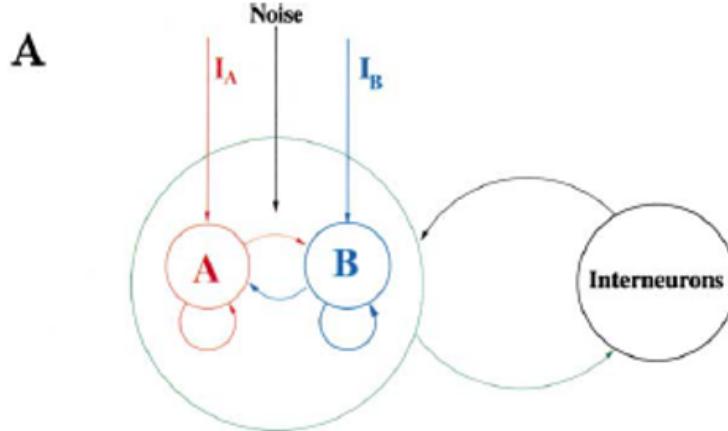
**Fig.3-4 Experimental Diagram**

Wang (2002) proposed a decision making network to model the activity of rhesus LIP neurons during decision making period in the visual motion discrimination task.

## 1.1 Biological background

As shown in Fig. 3-5, 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. 3-5, and we call other excitatory neurons as non-selective neurons,  $N_{non} = (1 - 2 * 0.15)N_E$ .

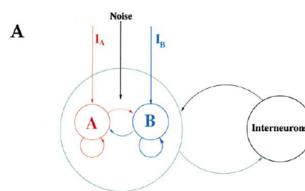


**Fig.3-5 structure of decision making network**

```

279 # def neurons
280 # def E neurons/pyramidal 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

```



## 1.1 Biological background

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 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}") } w- = 1 - f(x + -1)/1 - f
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, 0: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 conn.  
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

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  $\mu_x$  and variance  $\delta^2$ .

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

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

## 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.  
 $f_A \sim N(\mu_A, \delta^2)$   
 $f_B \sim N(\mu_B, \delta^2)$

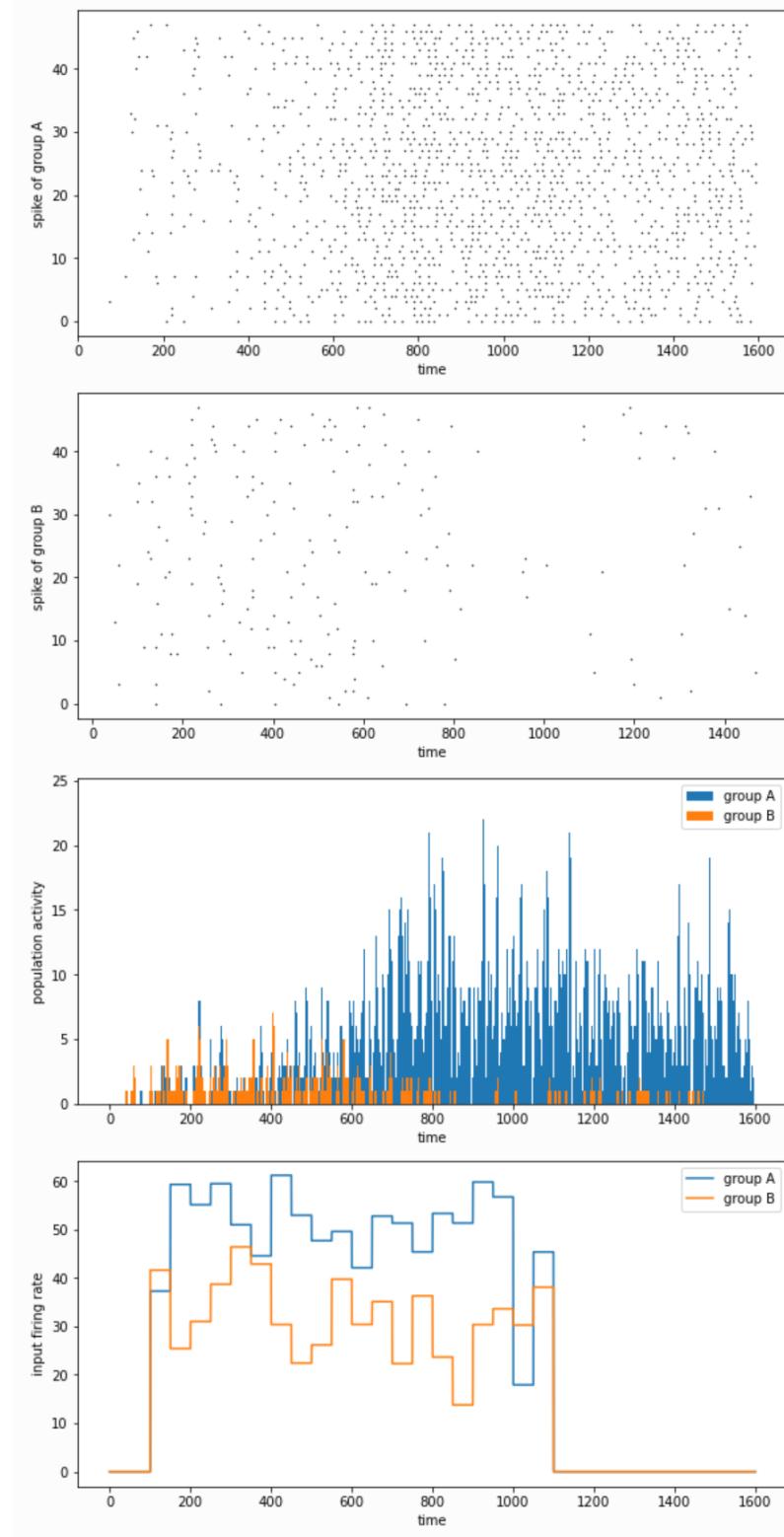
Generate Poisson inputs with the Poisson frequency of this neuron.

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

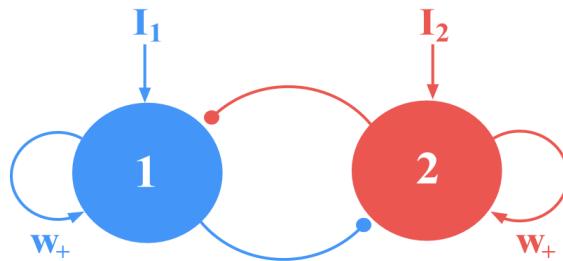


**Fig.3-6 decision making network**

## 3.2 Firing rate networks

### 3.2.1 Decision model

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)<sup>1</sup> 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. 3-1 Reduced decision model.** (From Wong & Wang, 2006<sup>1</sup>)

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. 3-1),

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

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

where  $I_0$  is the background current, and the external inputs  $I_1, I_2$  are determined by the total input strength  $\mu_0$  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:

## 1.1 Biological background

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

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

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.

```

1  from collections import OrderedDict
2  import brainpy as bp
3  bp.backend.set(backend='numba', dt=0.1)
4
5  class Decision(bp.NeuGroup):
6      target_backend = ['numpy', 'numba']
7
8      @staticmethod
9      def derivative(s1, s2, t, I, coh,
10                     JAext, J_rec, J_inh, I_0,
11                     a, b, d, tau_s, gamma):
12          I1 = JAext * I * (1. + coh)   ]   I1 =  $J_{A, \text{ext}} \mu_0 (1 + c')$ 
13          I2 = JAext * I * (1. - coh)   ]   I2 =  $J_{A, \text{ext}} \mu_0 (1 - c')$ 
14
15          I_syn1 = J_rec * s1 - J_inh * s2 + I_0 + I1
16          r1 = (a * I_syn1 - b) / (1. - bp.ops.exp(-d * (a * I_syn1 - b)))   ]    $I_{\text{syn},1} = J_{11} S_1 - J_{12} S_2 + I_0 + I_1$ 
17          ds1dt = - s1 / tau_s + (1. - s1) * gamma * r1   ]    $r_1 = \frac{a I_{\text{syn},1} - b}{1 - \exp(-d(a I_{\text{syn},1} - b))}$ 
18
19          I_syn2 = J_rec * s2 - J_inh * s1 + I_0 + I2
20          r2 = (a * I_syn2 - b) / (1. - bp.ops.exp(-d * (a * I_syn2 - b)))   ]    $I_{\text{syn},2} = J_{22} S_2 - J_{21} S_1 + I_0 + I_2$ 
21          ds2dt = - s2 / tau_s + (1. - s2) * gamma * r2   ]    $r_2 = \frac{a I_{\text{syn},2} - b}{1 - \exp(-d(a I_{\text{syn},2} - b))}$ 
22
23          return ds1dt, ds2dt   ]    $\frac{ds_1}{dt} = -S_1/\tau + (1 - S_1)\gamma r_1$ 
24
25      def __init__(self, size, coh, JAext=.00117, J_rec=.3725, J_inh=.1137,
26                  I_0=.3297, a=270., b=108., d=0.154, tau_s=.06, gamma=0.641,
27                  **kwargs):
28          # parameters
29          self.coh = coh
30          self.JAext = JAext
31          self.J_rec = J_rec
32          self.J_inh = J_inh
33          self.I0 = I_0
34          self.a = a
35          self.b = b
36          self.d = d
37          self.tau_s = tau_s
38          self.gamma = gamma
39
40          # variables
41          self.s1 = bp.ops.ones(size) * .06
42          self.s2 = bp.ops.ones(size) * .06
43          self.input = bp.ops.zeros(size)
44
45          self.integral = bp.odeint(f=self.derivative, method='rk4', dt=0.01)   ]
46
47      super(Decision, self).__init__(size=size, **kwargs)
48
49      def update(self, _t):
50          for i in range(self.size):
51              self.s1[i], self.s2[i] = self.integral(self.s1[i], self.s2[i], _t,
52                                                     self.input[i], self.coh,
53                                                     self.JAext, self.J_rec,
54                                                     self.J_inh, self.I0,
55                                                     self.a, self.b, self.d,
56                                                     self.tau_s, self.gamma)
57
58          self.input[i] = 0.

```

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

## 1.1 Biological background

```

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

We use numerical solution
rather than analytic solution
because the equations are
so complex. Thus, we don't
need to use the sympy
solver.

```

Let's first look at the case when there is no external input. At this time,  $\mu_0 = 0$ .

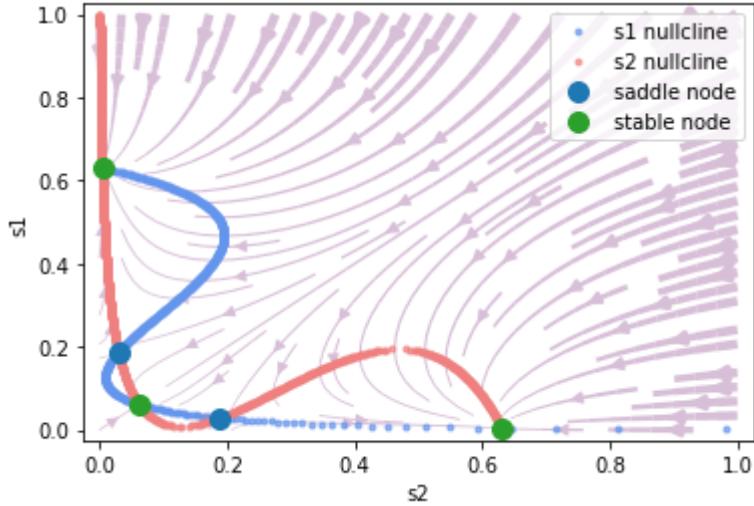
```
phase_analyze(I=0., coh=0.)
```

Output:

```

plot nullcline ...
plot fixed point ...
Fixed point #1 at s2=0.06176109215560733, s1=0.061761097890810475 is a stable r
Fixed point #2 at s2=0.029354239100062428, s1=0.18815448592736211 is a saddle r
Fixed point #3 at s2=0.0042468423702408655, s1=0.6303045696241589 is a stable r
Fixed point #4 at s2=0.6303045696241589, s1=0.004246842370235128 is a stable nc
Fixed point #5 at s2=0.18815439944520335, s1=0.029354240536530615 is a saddle r
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.

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.

## 1.1 Biological background

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.

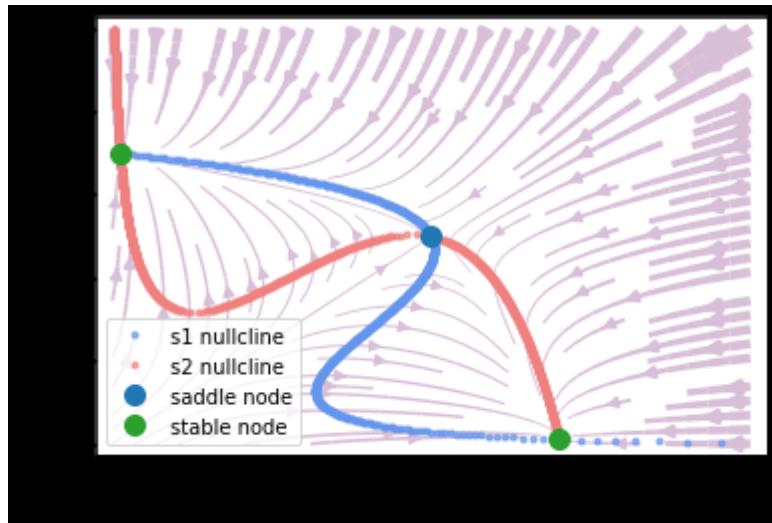
Now let's look at the phase plane under different coherences when we fix the external input strength to 30.

```
# 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)

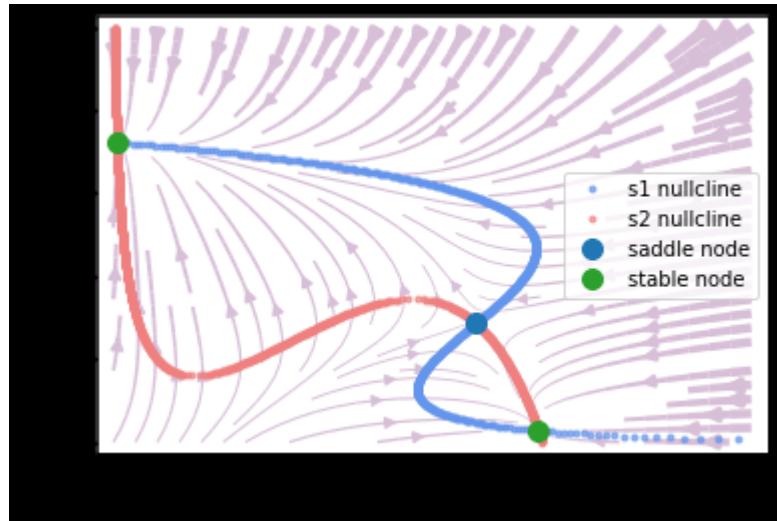
# coherence = 100%
print("coherence = 100%")
phase_analyze(I=30., coh=1.)
```

```
coherence = 0%
plot nullcline ...
plot fixed point ...
Fixed point #1 at s2=0.6993504413889349, s1=0.011622049526766405 is a stable node
Fixed point #2 at s2=0.49867489858358865, s1=0.49867489858358865 is a saddle node
Fixed point #3 at s2=0.011622051540013889, s1=0.6993504355529329 is a stable node
plot vector field ...
```

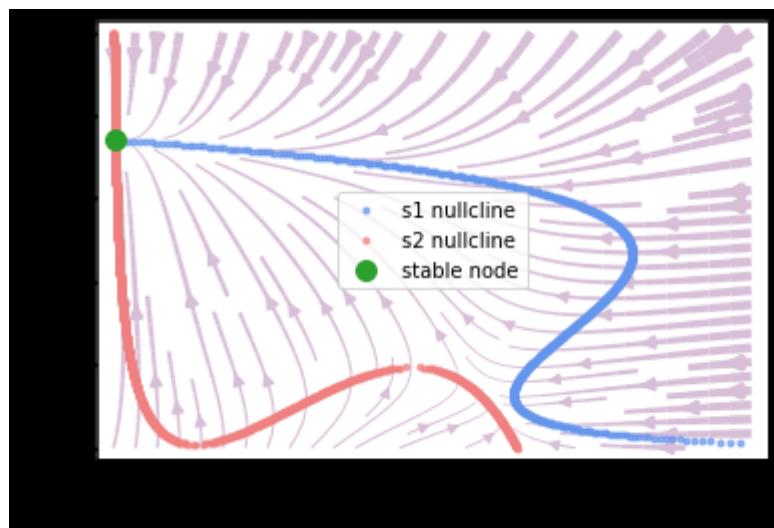


```
coherence = 51.2%
plot nullcline ...
plot fixed point ...
Fixed point #1 at s2=0.5673124813731691, s1=0.2864701069327971 is a saddle node
Fixed point #2 at s2=0.6655747347157656, s1=0.027835279565912054 is a stable node
Fixed point #3 at s2=0.005397687847426814, s1=0.7231453520305031 is a stable node
plot vector field ...
```

## 1.1 Biological background

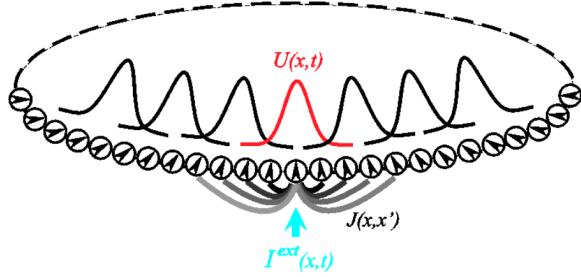


```
coherence = 100%
plot nullcline ...
plot fixed point ...
Fixed point #1 at s2=0.0026865954387078755, s1=0.7410985604497689 is a stable r
plot vector field ...
```



### 3.2.2 CANN

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



**Fig. 3-2 Structure of CANN.** (From Wu et al., 2008 <sup>2</sup>)

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 } Distances on the ring:  
z_range denotes the range of x
34

```

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) } Distances on the ring:  
x is the position of the ring, so we have:  
x ∈ [-π, π]  
z_range denotes the range of x (= 2π).  
distance on the ring: d = |x - x'| < π
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)) / ( } d=|x-x'|  
56             np.sqrt(2 * np.pi) * self.a) } Get a matrix of d for  
all positions } J(x, x') =  $\frac{e^{-\frac{(x-x')^2}{2a^2}}}{\sqrt{2\pi a}}$ 
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 - } I_{ek} = A e^{-\frac{|x-z(t)|^2}{4a^2}}
61                     pos) / self.a)) } I_{ek} = A e^{-\frac{|x-z(t)|^2}{4a^2}}
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.

```

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  $\pi$ , so the range of  $|x - x'|$  is  $2\pi$ , and  $-\pi$  and  $\pi$  are the same points (the actual maximum value is  $\pi$ , 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.

## 1.1 Biological background

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:

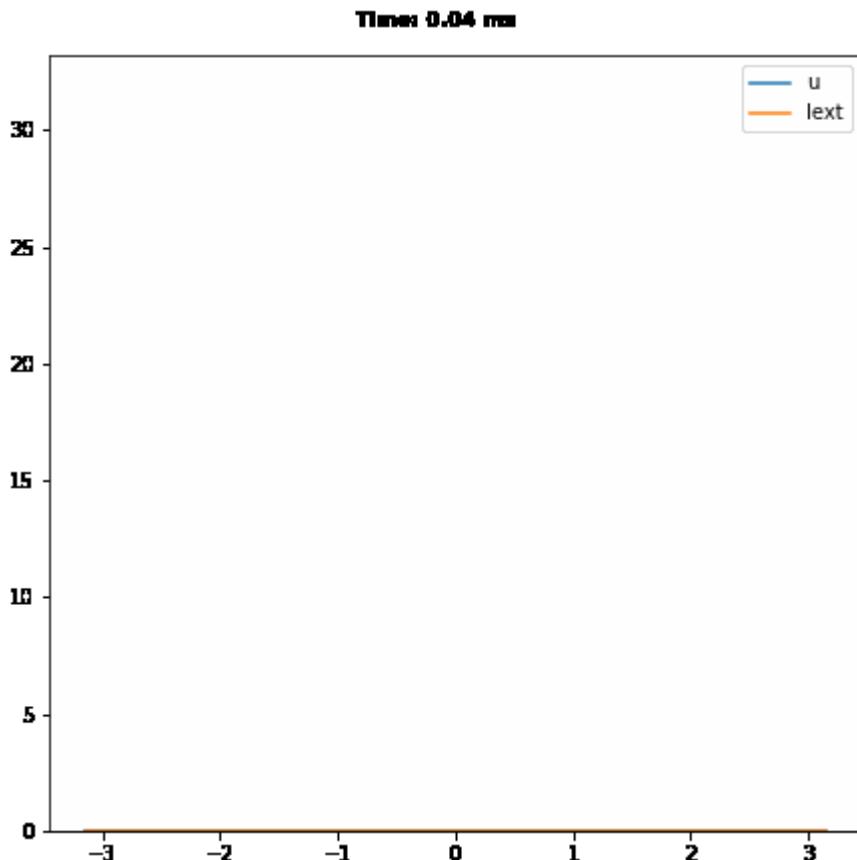
```
cann = CANN1D(num=512, k=0.1, monitors=['u'])

I1 = cann.get_stimulus_by_pos(0.)
Iext, duration = bp.inputs.constant_current([(0., 1.), (I1, 8.), (0., 8.)])
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.mon.u, 'xs': cann.x,
                                              'legend': 'u'}, {'ys': Iext,
                                              'xs': cann.x, 'legend': 'Iext'}],
                           frame_step=frame_step, frame_delay=frame_delay,
                           show=True)

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



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

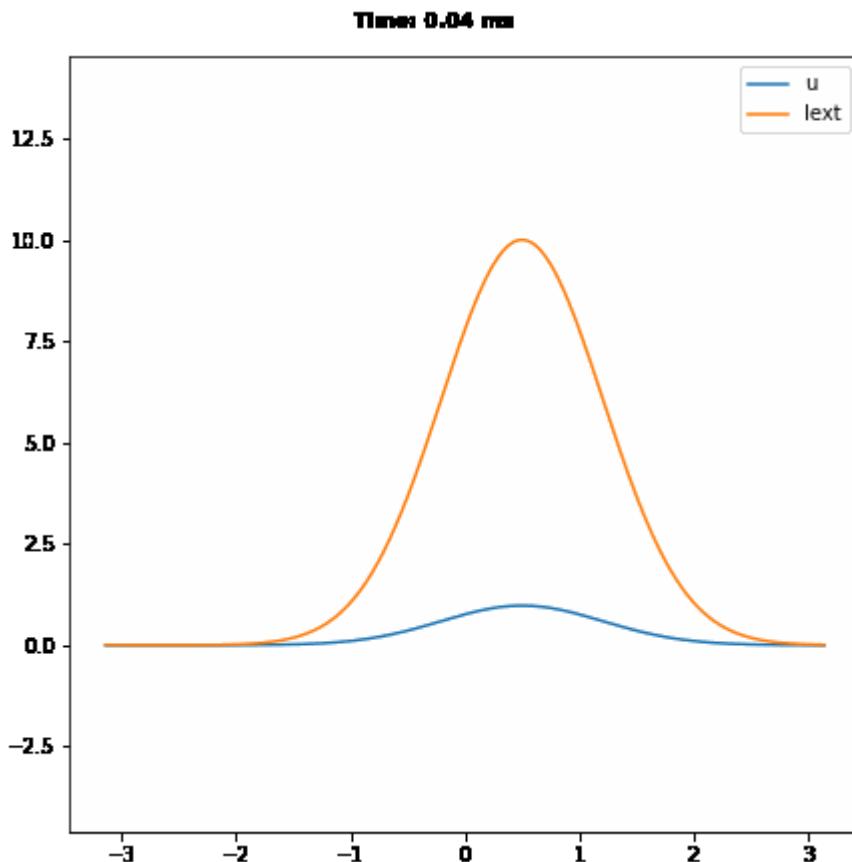
## 1.1 Biological background

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(num2, *cann.size)
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.

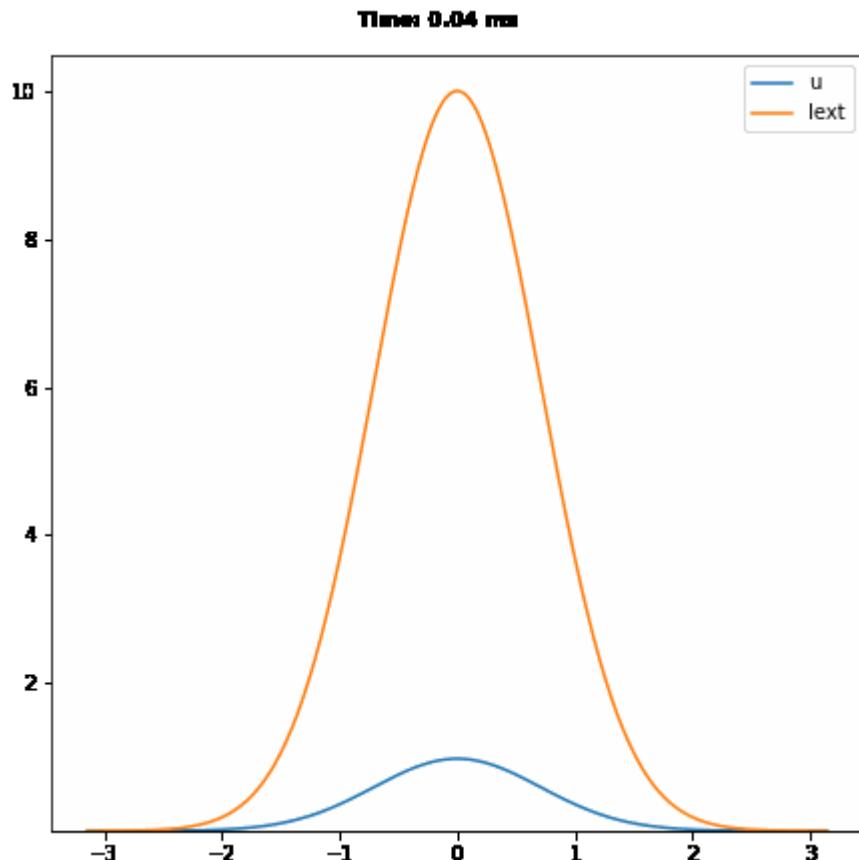
Now let's give a moving input, we vary the position of the input with `np.linspace`, we will see that the  $u$  will follow the input, i.e., smooth tracking.

## 1.1 Biological background

```
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

- <sup>1</sup>. Wong, K.-F. & Wang, X.-J. A Recurrent Network Mechanism of Time Integration in Perceptual Decisions. *J. Neurosci.* 26, 1314–1328 (2006). ↪
- <sup>2</sup>. Si Wu, Kosuke Hamaguchi, and Shun-ichi Amari. "Dynamics and computation of continuous attractors." *Neural computation* 20.4 (2008): 994-1025. ↪