Table of Contents

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.

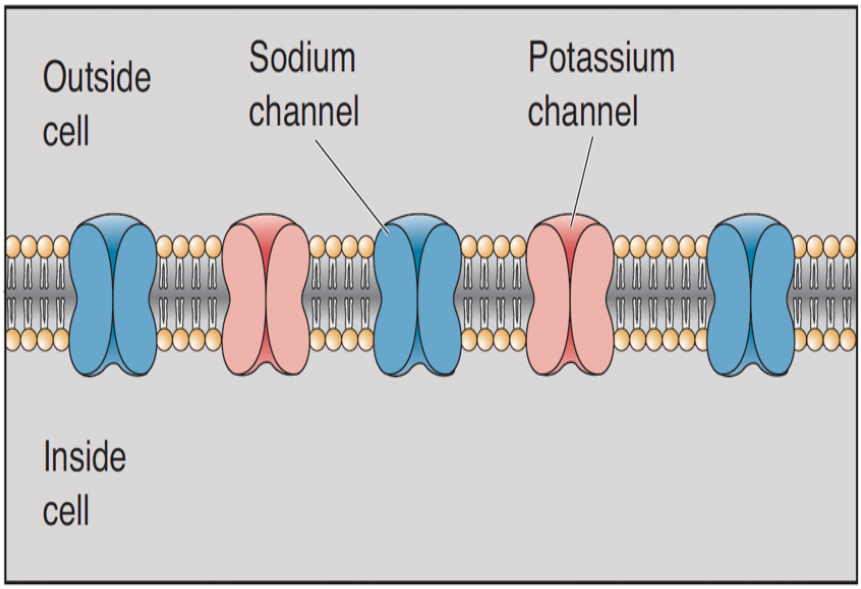


Fig. 1-1 Neuron membrane diagram (*Bear et al., 2014* [1])

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.

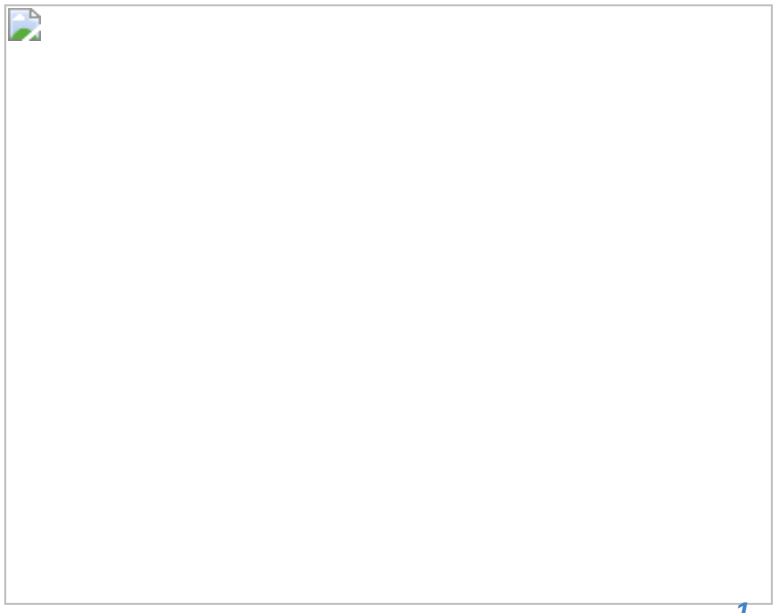


Fig. 1-2 Action Potential (Adapted from *Bear et al., 2014* [1])

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

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

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.

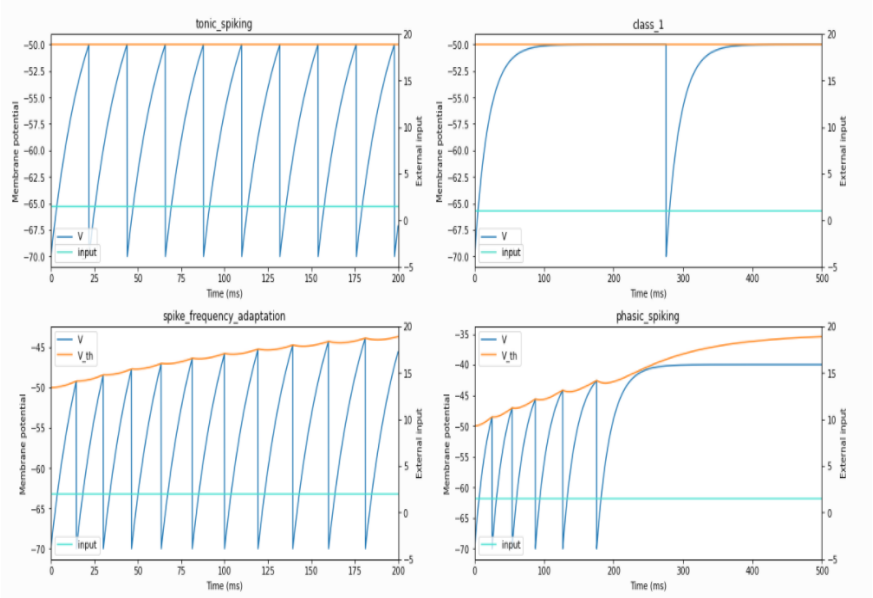


Figure 1-3 Some firing patterns

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

> [1]. Bear, Mark, Barry Connors, and Michael A. Paradiso. *Neuroscience: Exploring the brain*. Jones & Bartlett Learning, LLC, 2020. ↩

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 **Hodgin-Huxley model** (**HH model**).

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

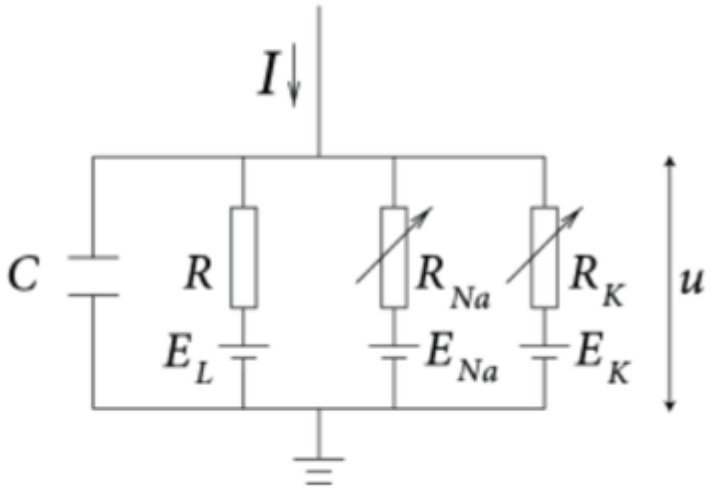


Fig. 1-4 Equivalent circuit diagram (*Gerstner et al., 2014* [1])

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    return dVdt, dmdt, dhdt, dndt

  def __init__(self, size, ENa=50., gNa=120., EK=-77., gK=36.,
               EL=-54.387, gL=0.03, V_th=20., C=1.0, **kwargs):
    # parameters
    self.ENa = ENa
    self.EK = EK
    self.EL = EL
    self.gNa = gNa
    self.gK = gK
    self.gL = gL
    self.C = C
    self.V_th = V_th

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

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

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

bp.NeuGroup class: Group of neurons

Set backend for model.

Call `bp.odeint` to integrate ODEs. Set parameter `method` to choose numerical integration methods.

$\alpha_m(V) = (0.1(V+40))/(1-\exp(-(V+40)/10))$
$\beta_m(V) = 4.0\exp(-(V+65)/18)$
$dx/dt = \alpha(1-x) - \beta x, \quad x = m$

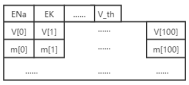
$\alpha_h(V) = 0.07\exp(-(V+65)/20)$
$\beta_h(V) = 1/(1+\exp(-(V+35)/10))$
$dx/dt = \alpha(1-x) - \beta x, \quad x = h$

$\alpha_n(V) = (0.01(V+55))/(1-\exp(-(V+55)/10))$
$\beta_n(V) = 0.125\exp(-(V+65)/80)$
$dx/dt = \alpha(1-x) - \beta x, \quad x = n$

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

Model parameters saved as floating point numbers.

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



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

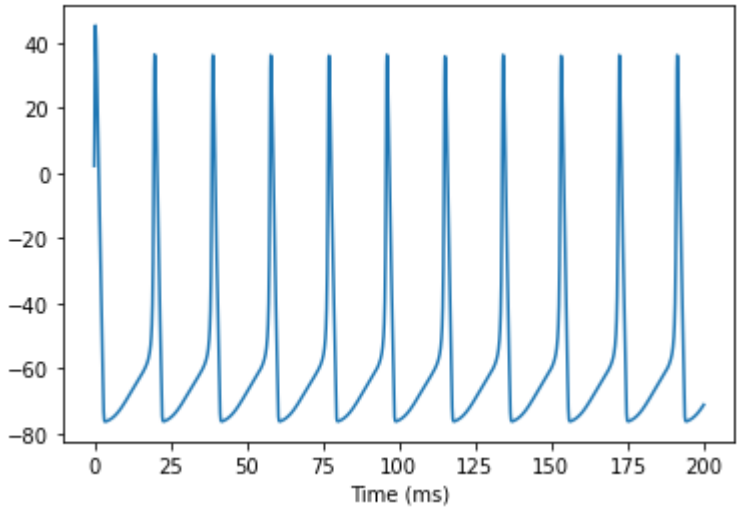
Update variables with numerical integration in vector form.

Judge if the neuron spikes.

Reset external input for this time step.

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

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



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

1.3 Reduced models

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

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

1.3.1 Leaky Integrate-and-Fire model

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

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

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

$$V \leftarrow V_{reset}$$

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

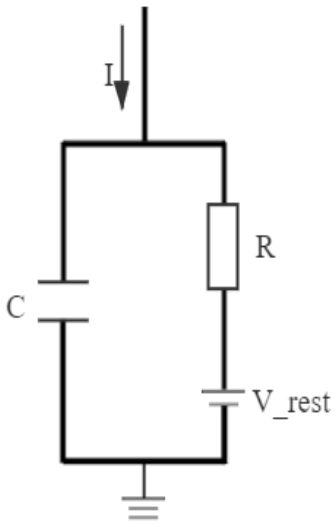


Fig1-4 Equivalent circuit of LIF model

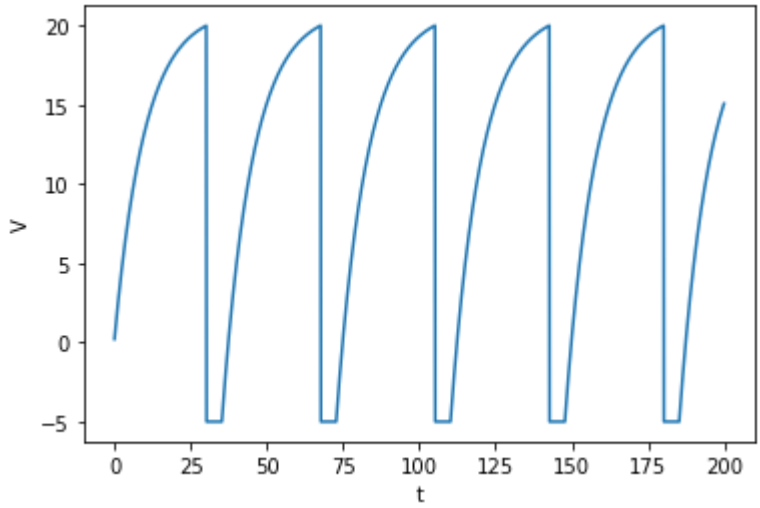
Compared with HH model, LIF model does not model the shape of action potentials, which means, the membrane potential does not burst before a spike. Also, the refractory period is overlooked in the original model, and in order to generate it, another conditional judgment must be added:

If

$$t - t_{lastspike} <= refractoryperiod$$

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

```
1   class LIF(bp.NeuGroup):                                  # bp.NeuGroup class: Group of neurons
2     target_backend = ['numpy', 'numba', 'numba-parallel', 'numba-cuda']
3                                                            # Users may also set backend with a list of backend name.
4     @staticmethod
5     def derivative(V, t, Iext, V_rest, R, tau):
6       dvdt = (-V + V_rest + R * Iext) / tau                # τ 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                     # Model parameters saved as floating point numbers.
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                   # Model variables saved as vectors of floating point numbers.
26
27      self.integral = bp.odeint(self.derivative)           # Call `bp.odeint` to integrate ODEs. Set parameter `method` to choose numerical integration methods.
28      super(LIF, self).__init__(size=size, **kwargs)
29                                                           # Pass `size` and `**kwargs` to superclass bp.NeuGroup's constrctor.
30    def update(self, _t):
31      for i in prange(self.size[0]):                       # For each neuron in neuron group.
32        spike = 0.
33        refractory = (_t - self.t_last_spike[i] <= self.t_refractory)   # Check if neuron is in refractory period.
34        if not refractory:
35          V = self.integral(self.V[i], _t, self.input[i],  # Update variables with numerical integration one by one.
36                            self.V_rest, self.R, self.tau)
37        spike = (V >= self.V_th)                           # Check if neuron spikes.
38        if spike:
39          V = self.V_reset                                 # Reset neuron.
40          self.t_last_spike[i] = _t
41        self.V[i] = V
42        self.spike[i] = spike
43        self.refractory[i] = refractory or spike
44        self.input[i] = 0.                                 # Reset external input 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** (**QuaIF model**), in which they add a second order term in differential equation so the neurons can generate spike better.

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

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

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

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

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

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

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

bp.NeuGroup class: Group of neurons

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

Model parameters saved as floating point numbers.

Model variables saved as vectors of floating point numbers.

Call `bp.odeint` to integrate ODEs. Set parameter `method` to choose numerical integration methods.

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

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

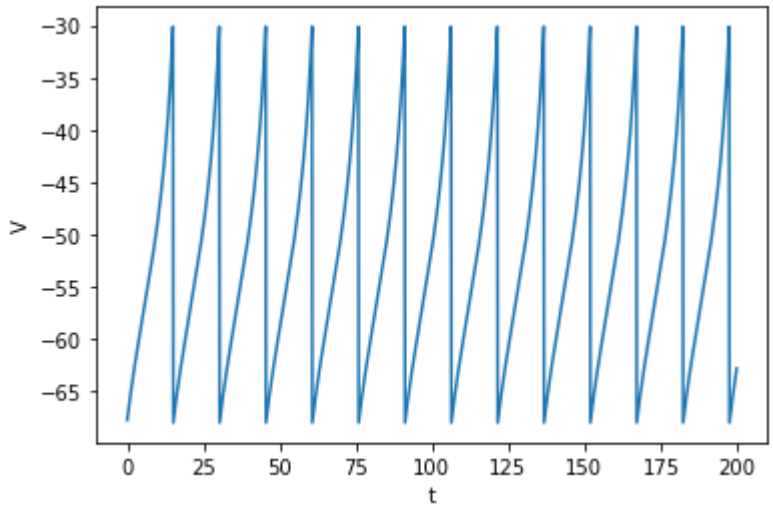
For each neuron in neuron group.

Check if neuron is in refractory period.

Update variables with numerical integration one by one.

Check if neuron spikes.

Reset neuron.



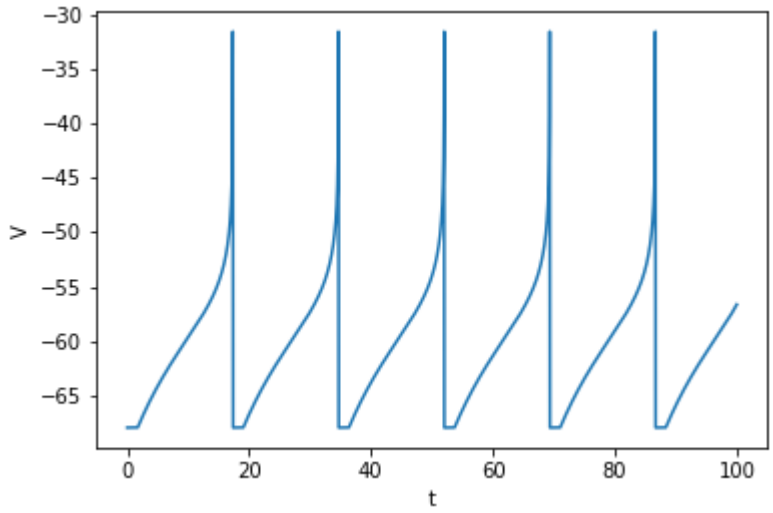
1.3.3 Exponential Integrate-and-Fire model

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

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

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

```python
class ExpIF(bp.NeuGroup):                                   # bp.NeuGroup class: Group of neurons
  target_backend = 'general'

  @staticmethod
  def derivative(V, t, I_ext, V_rest, delta_T, V_T, R, tau):
    exp_term = bp.ops.exp((V - V_T) / delta_T)
    dvdt = (-(V-V_rest) + delta_T*exp_term + R*I_ext) / tau  # τ dV/dt = -(V - V_rest) + Δ_T e^{(V-V_T)/Δ_T} + RI(t)
    return dvdt

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

    # variables                                             # Model variables saved as vectors of floating point numbers.
    self.V = bp.ops.ones(size) * V_rest
    self.input = bp.ops.zeros(size)
    self.spike = bp.ops.zeros(size, dtype=bool)
    self.refractory = bp.ops.zeros(size, dtype=bool)
    self.t_last_spike = bp.ops.ones(size) * -1e7

    self.integral = bp.odeint(self.derivative)              # Call `bp.odeint` to integrate ODEs. Parameter `method` is set to default value `euler`.
    super(ExpIF, self).__init__(size=size, **kwargs)        # Pass `size` and `**kwargs` to superclass bp.NeuGroup's constrctor.

  def update(self, _t):
    for i in prange(self.num):                              # For each neuron in neuron group.
      spike = 0.
      refractory = (_t - self.t_last_spike[i] <= self.t_refractory)   # Check if neuron is in refractory period.
      if not refractory:
        V = self.integral(
          self.V[i], _t, self.input[i], self.V_rest,        # Update variables with numerical integration one by one.
          self.delta_T, self.V_T, self.R, self.tau
        )
        spike = (V >= self.V_th)                            # Check if neuron spikes.
        if spike:                                           # Reset neuron.
          V = self.V_reset
          self.t_last_spike[i] = _t
        self.V[i] = V
      self.spike[i] = spike
      self.refractory[i] = refractory or spike
    self.input[:] = 0.
```



1.3.4 Adaptive Exponential Integrate-and-Fire model

While facing a constant stimulus, the response generated by a single neuron will sometimes decreases 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, QuaIF and ExpIF models. Here we introduce a typical one: **Adaptive Exponential Integrate-and-Fire model** (**AdExIF model**) (Gerstner et al., 2014).

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

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

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

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

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

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

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

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

    return dVdt, dwdt

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

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

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

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

  def update(self, _t):
    for i in prange(self.size[0]):
      spike = 0.
      refractory = (_t - self.t_last_spike[i] <= self.t_refractory)
      if not refractory:
        V, w = self.integral(self.V[i], self.w[i], _t, self.input[i],
                             self.V_rest, self.delta_T,
                             self.V_T, self.R, self.tau, self.tau_w, self.a)
        spike = (V >= self.V_th)
        if spike:
          V = self.V_rest
          w += self.b
          self.t_last_spike[i] = _t
        self.V[i] = V
        self.w[i] = w
      self.spike[i] = spike
      self.refractory[i] = refractory or spike
      self.input[i] = 0.
```

bp.NeuGroup class: Group of neurons

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

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

Model parameters saved as floating point numbers.

Model variables saved as vectors of floating point numbers.

Call `bp.odeint` to integrate ODEs. Set parameter `method` to choose numerical integration methods.

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

For each neuron in neuron group.

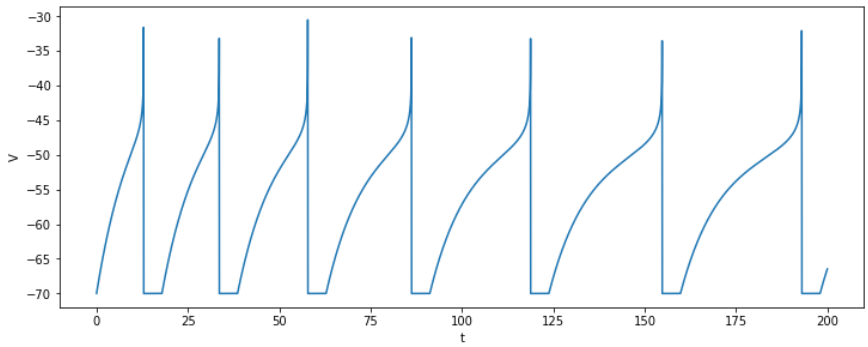
Check if neuron is in refractory period.

Update variables with numerical integration one by one.

Check if neuron spikes.

Reset membrane potential, add the Dirac δ term of w:

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



1.3.5 Hindmarsh-Rose model

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.

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

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

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

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

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

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

bp.NeuGroup class: Group of neurons

$dV/dt = y - aV^3 + bV^2 - z + I$
$dy/dt = c - dV^2 - y$
$dz/dt = r(s(V - V_{rest}) - z)$

Model parameters saved as floating point numbers.

Model variables saved as vectors of floating point numbers.

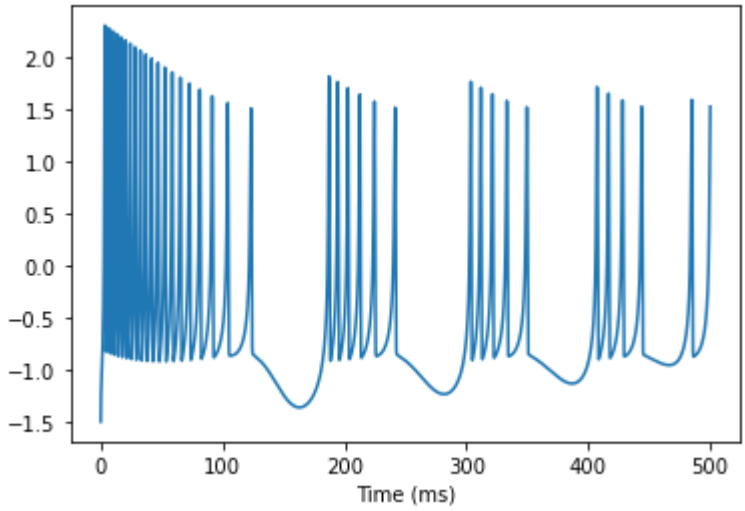
Call `bp.odeint` to integrate ODEs. Parameter `method` is set to default value `euler`.

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

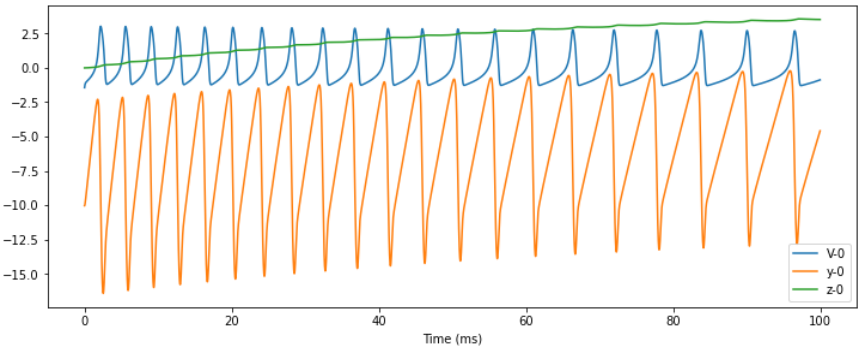
For each neuron in neuron group.

Update variables with numerical integration one by one.

Reset external input for this time step.

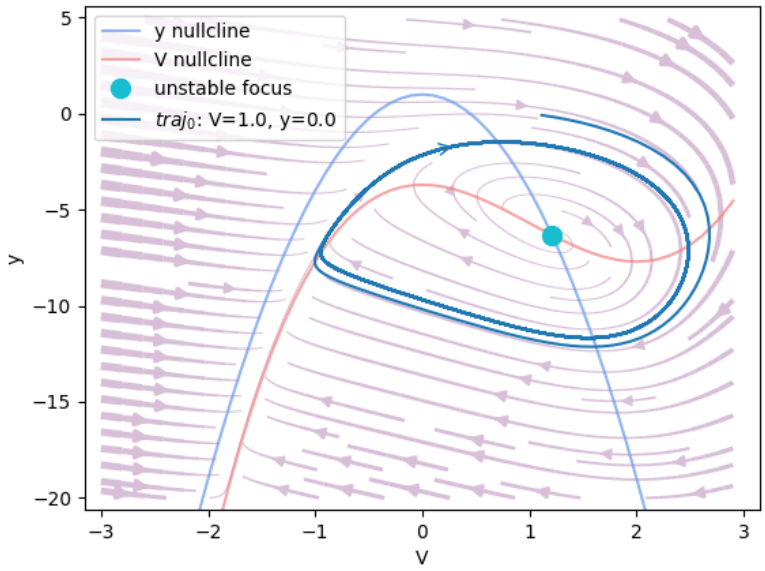


In the variable-t plot painted below, we may see that the slow variable z changes much slower than V and y. Also, V and y are changing periodically during the simulation.



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.

```
# Phase plane analysis
phase_plane_analyzer = bp.analysis.PhasePlane(
    neu.integral,                                          → Dynamic system to be analyzed.
    target_vars={'V': [-3., 3.], 'y': [-20., 5.]},         → Variables to be showed in phase plane.
    fixed_vars={'z': 0.},                                  → Variables to be fixed in phase plane.
    pars_update={'I_ext': param[mode][1], 'a': 1., 'b': 3.,
                 'c': 1., 'd': 5., 'r': 0.01, 's': 4.,     } Other parameters to be fixed.
                 'V_rest': -1.6}
)
phase_plane_analyzer.plot_nullcline()                      → Plot nullcline.
phase_plane_analyzer.plot_fixed_point()                    → Plot fixed points.
phase_plane_analyzer.plot_vector_field()                   → Plot vector field.
phase_plane_analyzer.plot_trajectory(
    [{'V': 1., 'y': 0., 'z': -0.}],                        } Plot trajectory.
    duration=100.,                                         } Define start point of trajectory and
    show=True                                              } 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:

$$I_j \leftarrow R_j I_j + A_j$$

$$V \leftarrow V_{reset}$$

$$V_{th} \leftarrow max(V_{th_{reset}}, V_{th})$$

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

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

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

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

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

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

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

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

bp.NeuGroup class: Group of neurons

$$dI_1/dt = -k_1 I_1$$
$$dI_2/dt = -k_2 I_2$$
$$dV_{th}/dt = a(V - V_{rest}) - b(V_{th} - V_{th_\infty})$$
$$\tau dV/dt = -(V - V_{rest}) + R\sum I_j + RI(t), \quad j = 1,2$$

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 constrctor.

```
46  def update(self, _t):
47    for i in prange(self.size[0]):
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
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
```

For each neuron in neuron group.

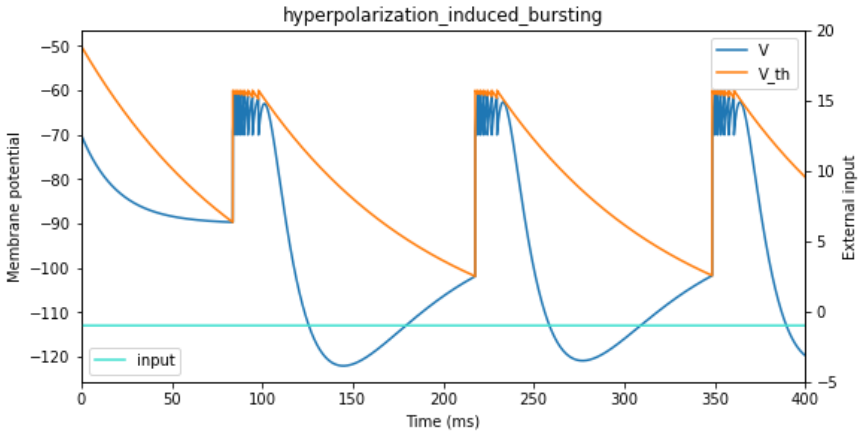
Update variables with numerical integration one by one.

Check if neuron spikes.

Reset membrane potential and threshold potential, update I1, I2.

$V = V_{reset}$
$I_1 = R_1 I_1 + A_1$
$I_2 = R_2 I_2 + A_2$
$V_{th} = \max(V_{th}, V_{th_{reset}})$

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 ν). 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.

$$\tau_e \frac{da_e(t)}{dt} = -a_e(t) + (k_e - r_e * a_e(t)) * \mathcal{S}(c_1 a_e(t) - c_2 a_i(t) + I_{ext_e}(t))$$

$$\tau_i \frac{da_i(t)}{dt} = -a_i(t) + (k_i - r_i * a_i(t)) * \mathcal{S}(c_3 a_e(t) - c_4 a_i(t) + I_{ext_i}(t))$$

$$\mathcal{S}(x) = \frac{1}{1 + exp(-a(x - \theta))} - \frac{1}{1 + exp(a\theta)}$$

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

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

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

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

$$S(x) = \frac{1}{1+\exp(-a(x-\theta))} - \frac{1}{1+\exp(a\theta)}$$

$$\tau_e \frac{da_e}{dt} = -a_e + (k_e - r_e * a_e) * \mathrm{S_e}(\mathrm{c_1 a_e} - \mathrm{c_2 a_i} + \mathrm{I_{ext_e}})$$

$$\tau_i \frac{da_i}{dt} = -a_i + (k_i - r_i * a_i) * \mathrm{S_i}(\mathrm{c_1 a_e} - \mathrm{c_2 a_i} + \mathrm{I_{ext_i}})$$

```
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)
48          super(FiringRateUnit, self).__init__(size=size, **kwargs)
49
50      def update(self, _t):
51          self.a_e, self.a_i = self.integral(
52              self.a_e, self.a_i, _t,
53              self.k_e, self.r_e, self.c1, self.c2,
54              self.input_e, self.slope_e,
55              self.theta_e, self.tau_e,
56              self.k_i, self.r_i, self.c3, self.c4,
57              self.input_i, self.slope_i,
58              self.theta_i, self.tau_i)
59          self.input_e[:] = 0.
60          self.input_i[:] = 0.
```

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 constrctor.

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

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

2.1.1 Chemical Synapses

Biological Background

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

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

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

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

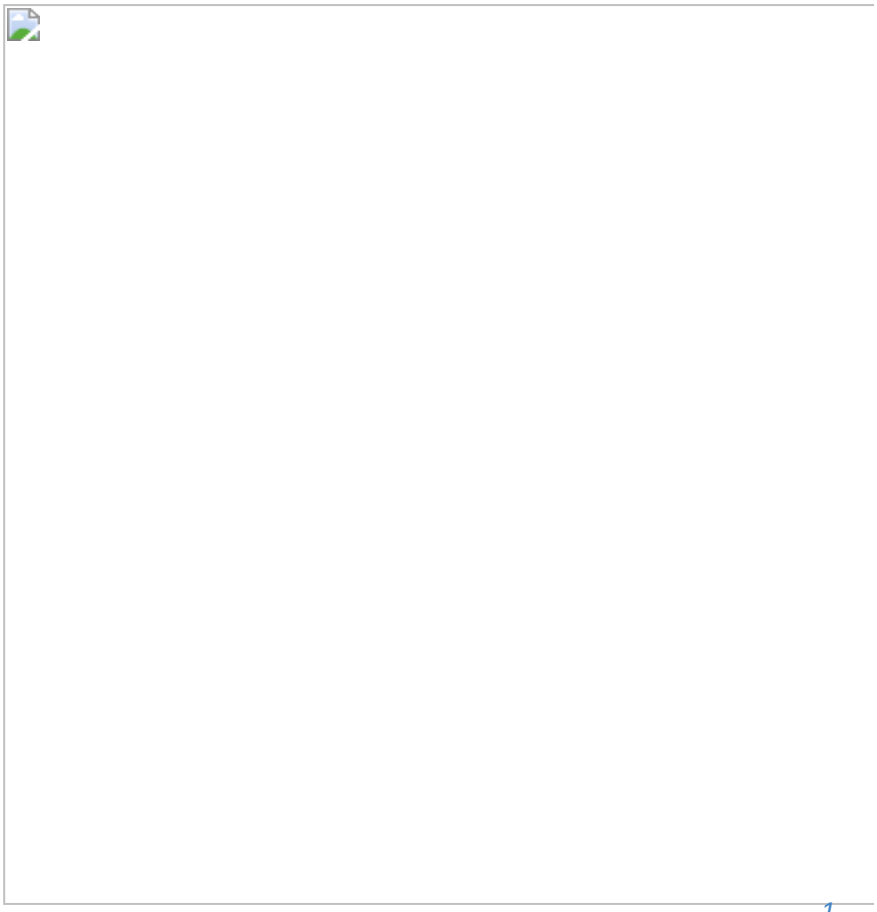


Fig. 2-1 Biological Synapse (Adapted from *Gerstner et al., 2014* [1])

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

AMPA Synapse

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

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

Fig. 2-2 Markov process of channel dynamics

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

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

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

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

```python
import brainpy as bp


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

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

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

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

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

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

```

bp.TwoEndConn class:
Connections between two neuron groups

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

Regard [T] as a constant T, last for T_duration

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

We update s by an `update` function.

```
32    def update(self, _t):
33      for i in range(self.size):
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])
40              < self.T_duration) * self.T
41        self.s[i] = self.int_s(self.s[i], _t, TT, self.alpha, self.beta)
42
```

For each single synapse

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

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

```
43
44  import brainmodels as bm
45
46  bp.backend.set(backend='numba', dt=0.1)
47
48
49  def run_syn(syn_model, **kwargs):
50    neu1 = bm.neurons.LIF(2, monitors=['V'])
51    neu2 = bm.neurons.LIF(3, monitors=['V'])
52
53    syn = syn_model(pre=neu1, post=neu2, conn=bp.connect.All2All(),
54                    monitors=['s'], **kwargs)
55
56    net = bp.Network(neu1, syn, neu2)
57    net.run(30., inputs=(neu1, 'input', 35.))
58    bp.visualize.line_plot(net.ts, syn.mon.s, ylabel='s', show=True)
59
60
61  run_syn(AMPA, T_duration=3.)
62
```

Set numba backend

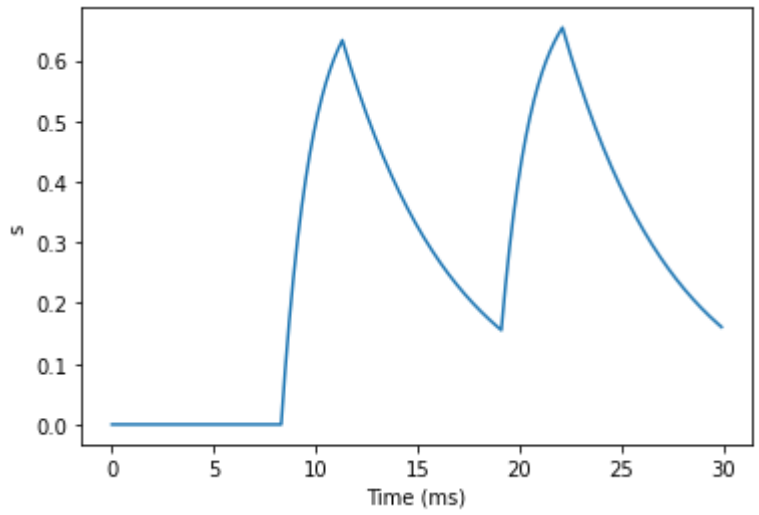
A method to run synapse

Get two LIF neuron groups from brainmodels package

Specify pre and post neurons and there connections. Here we use all to all connections provided by BrainPy.

Run AMPA synapse and set parameter 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.

NMDA Synapse

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

conductance g is given by,

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

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

where g_∞ represents the effect of magnesium ion concentration, and its value decreases as the magnesium ion concentration increases. With the increase of the voltage V, the effect of c_{Mg} on g_∞ decreases. α, β and $\bar{g}$ are some constants。 The dynamic of gating variable s is similar with the AMPA synapse, which is given by,

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

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

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

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

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

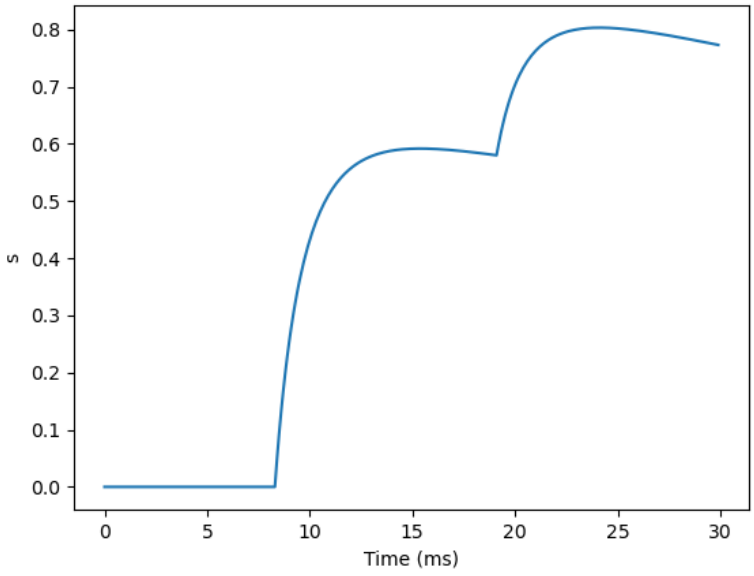
```
class NMDA(bp.TwoEndConn):
  target_backend = ['numpy', 'numba']

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

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

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

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

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

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

$$\left.\begin{aligned} \frac{ds}{dt} &= -s/\tau_{decay} + ax(1-s) \\ \frac{dx}{dt} &= -x/\tau_{rise} \end{aligned}\right\}$$

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

    self.x[i] += self.pre.spike[pre_id]                    ——→ if (pre spike), then x = x + 1
    self.s[i], self.x[i] = self.integral(self.s[i], self.x[i], _t,
                                         self.tau_rise, self.tau,
                                         self.a)

    # output
    g_inf_exp = bp.ops.exp(-self.alpha * self.post.V[post_id])
    g_inf = 1 + g_inf_exp * self.cc_Mg / self.beta          } g_∞ = (1 + e^{-αV} c_M g/β)^{-1}

    self.g.push(i, self.g_max * self.s[i] / g_inf)          ——→ g = ḡ g_∞ s

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

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

```
run_syn(NMDA)
```



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

$GABA_B$ Synapse

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

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

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

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

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

Here are the codes of the BrainPy implementation.

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

  @staticmethod
  def derivative(R, G, t, k3, TT, k4, k1, k2):
    dRdt = k3 * TT * (1 - R) - k4 * R
    dGdt = k1 * R - k2 * G
    return dRdt, dGdt

  def __init__(self, pre, post, conn, delay=0., g_max=0.02, E=-95.,
               k1=0.18, k2=0.034, k3=0.09, k4=0.0012, kd=100., T=0.5,
               T_duration=0.3, **kwargs):
    # params
    self.g_max = g_max
    self.E = E
    self.k1 = k1
    self.k2 = k2
    self.k3 = k3
    self.k4 = k4
    self.kd = kd
    self.T = T
    self.T_duration = T_duration

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

    # data
    self.R = bp.ops.zeros(self.size)
    self.G = bp.ops.zeros(self.size)
    self.t_last_pre_spike = bp.ops.ones(self.size) * -1e7
    self.s = bp.ops.zeros(self.size)
    self.g = self.register_constant_delay('g', size=self.size,
                                          delay_time=delay)

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


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

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

      self.R[i], G = self.integral(self.R[i], self.G[i], _t, self.k3,
                                   TT, self.k4, self.k1, self.k2)
      self.s[i] = G ** 4 / (G ** 4 + self.kd)
      self.G[i] = G

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

$$\begin{cases} \frac{d[R]}{dt} = -k_3[T](1-[R]) - k_4[R] \\ \frac{d[G]}{dt} = -k_1[R] - k_2[G] \end{cases}$$

Regard [T] as a constant T, last for T_duration

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

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

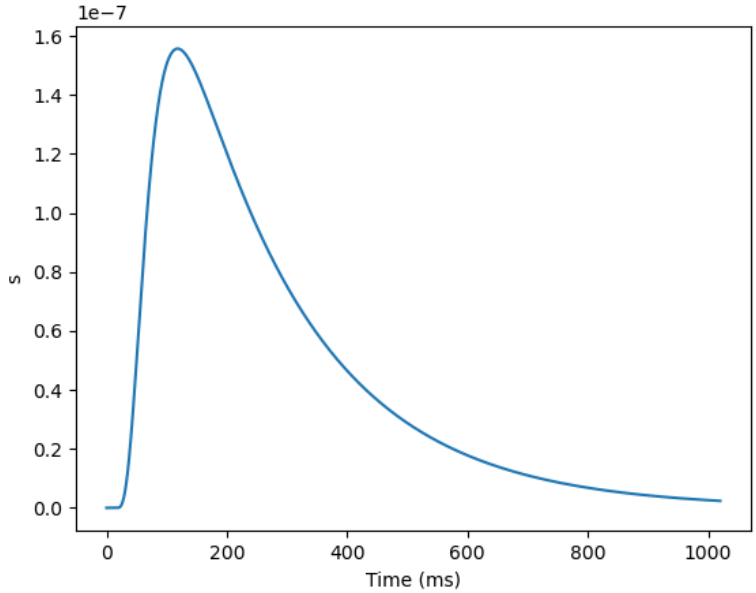
$g = \overline{g}\, s$

$I = g(V-E)$

Since the dynamics of $GABA_B$ is very slow, we no longer use the `run_syn` function here, but use `bp.inputs.constant_current` to give stimulation for 20ms, and then look at the decay at the subsequent

1000ms without external inputs.

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

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

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



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

Current-based and Conductance-based synapses

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

(1) Current-based

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

$$I \propto s$$

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

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

```
def __init__(self, pre, post, conn, delay, **kwargs):
  # ...
  self.s = bp.ops.zeros(self.size)
  self.w = bp.ops.ones(self.size) * .2
  self.I_syn = self.register_constant_delay('I_syn', size=self.size,
                                            delay_time=delay)


def update(self, _t):
  for i in range(self.size):
    # ...
    self.I_syn.push(i, self.w[i] * self.s[i])
    self.post.input[post_id] += self.I_syn.pull(i)
```

→ set delay time before changing synaptic current by using the register_constant_delay method

} use push and pull to set delay before applying synaptic current into postsynatic 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`.

```
def __init__(self, pre, post, conn, g_max, E, delay, **kwargs):
  self.g_max = g_max
  self.E = E
  # ...
  self.s = bp.ops.zeros(self.size)
  self.g = self.register_constant_delay('g', size=self.size, delay_time=delay)


def update(self, _t):
  for i in range(self.size):
    # ...
    self.g.push(i, self.g_max * self.s[i])
    self.post.input[post_id] -= self.g.pull(i) * (self.post.V[post_id] - self.E)

```

→ set delay time before changing the conductance

→ $g = \bar{g}s$

→ $I = g(V - E)$

Now let's review the NMDA and $GABA_B$ synapses we just implemented. They are both conductance-based models. Excitatory currents are induced by $E = 0$ in the NMDA synapse; while inhibitory currents are produced by $E = -95$ in the $GABA_B$ synapse. You can find the complete code to the above models from our BrainModels GitHub repository.

Abstract reduced models

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

(1) Differences of two exponentials

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

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

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

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

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

$$\frac{dx}{dt} = -\frac{\tau_1 + \tau_2}{\tau_1 \tau_2}x - \frac{s}{\tau_1 \tau_2}$$

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

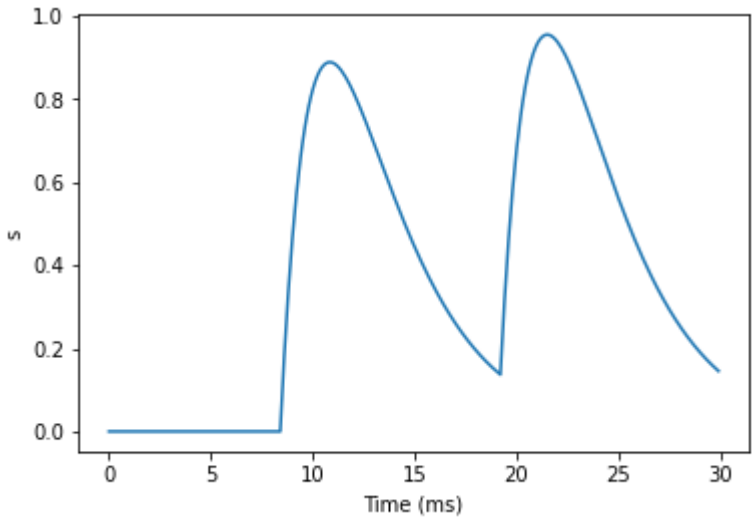
Here we specify the logic of increment of x in the `update` function when the presynaptic neurons fire. The code is as follows:

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

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

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

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

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

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

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

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

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


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

$$\left.\begin{aligned}\frac{dx}{dt} &= (-(\tau_1+\tau_2)x - s)/(\tau_1\tau_2)\\ \frac{ds}{dt} &= x\end{aligned}\right\}$$

→ if (pre spike), then x = x + 1

Then we expect to see the following result:



(2) Alpha synapse

Dynamics of `Alpha synapse` is given by,

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

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

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

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

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

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

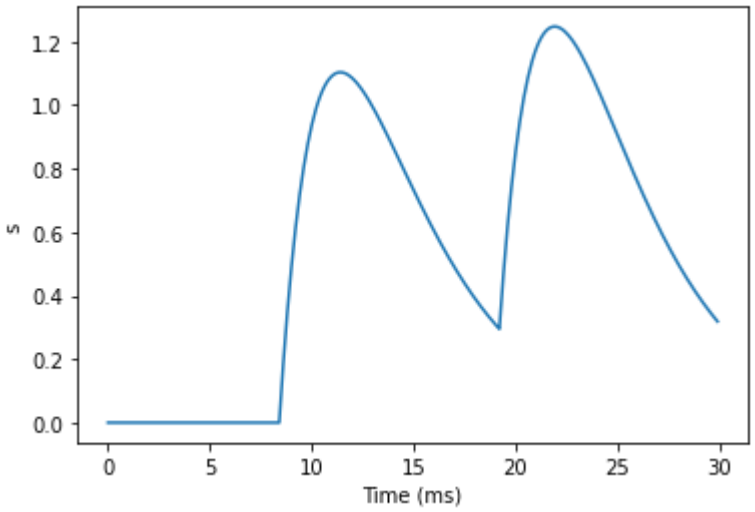
Code implementation is similar:

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

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

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

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

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

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

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

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

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

run_syn(Alpha)
```

$\frac{dx}{dt} = (-2\tau x - s)/\tau^2$, $\frac{ds}{dt} = x$

→ if (pre spike), then x = x + 1

Then we expect to see the following result:



(3) Single exponential decay

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

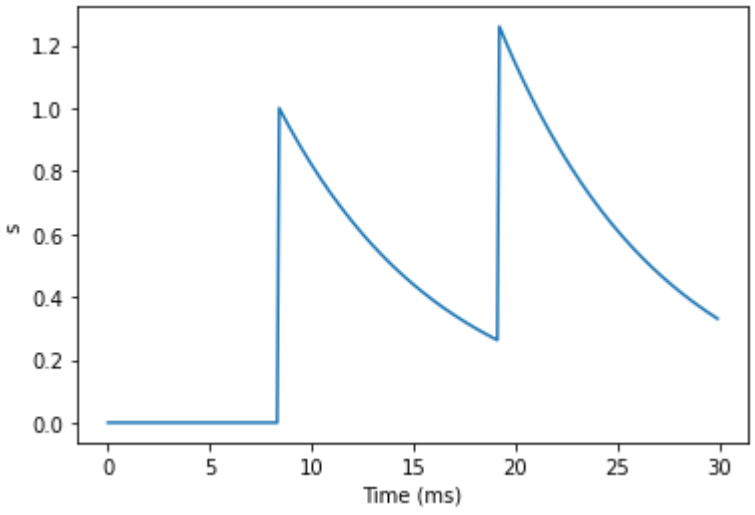
$$\frac{ds}{dt} = -\frac{s}{\tau_{decay}}$$

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

The implementing code is given by:

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

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

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

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

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

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

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

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

      self.s[i] = self.integral(self.s[i], _t, self.tau)
      self.s[i] += self.pre.spike[pre_id]      ----------> if (pre spike), then s = s + 1


run_syn(Exponential)
```

Then we expect to see the following result:



(4) Voltage jump

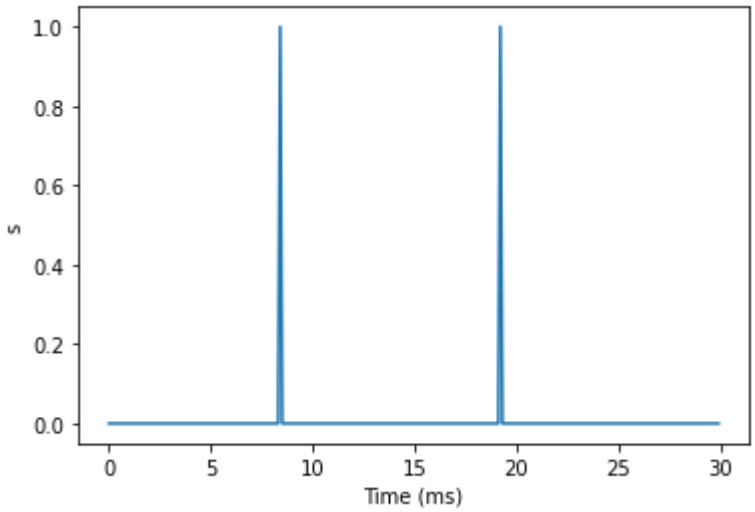
Sometimes even the decay process can be ignored, so there is a `voltage jump` model, which is given by:

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

The code is as follows:

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

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

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

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

  def update(self, _t):
    for i in range(self.size):
      pre_id = self.pre_ids[i]
      self.s[i] = self.pre.spike[pre_id]      ------------------>  if (pre spike), then s = s + 1


run_syn(Voltage_jump)
```

Then we expect to see the following result:



2.1.2 Electrical Synapses

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

(a) **(b)**

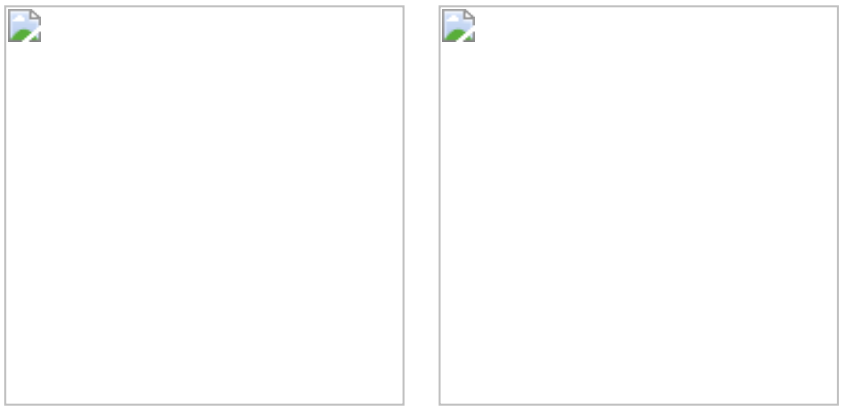


Fig. 2-3 (a) Gap junction connection between two cells. **(b)** An equivalent diagram.
(From *Sterratt et al., 2011* [2])

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.

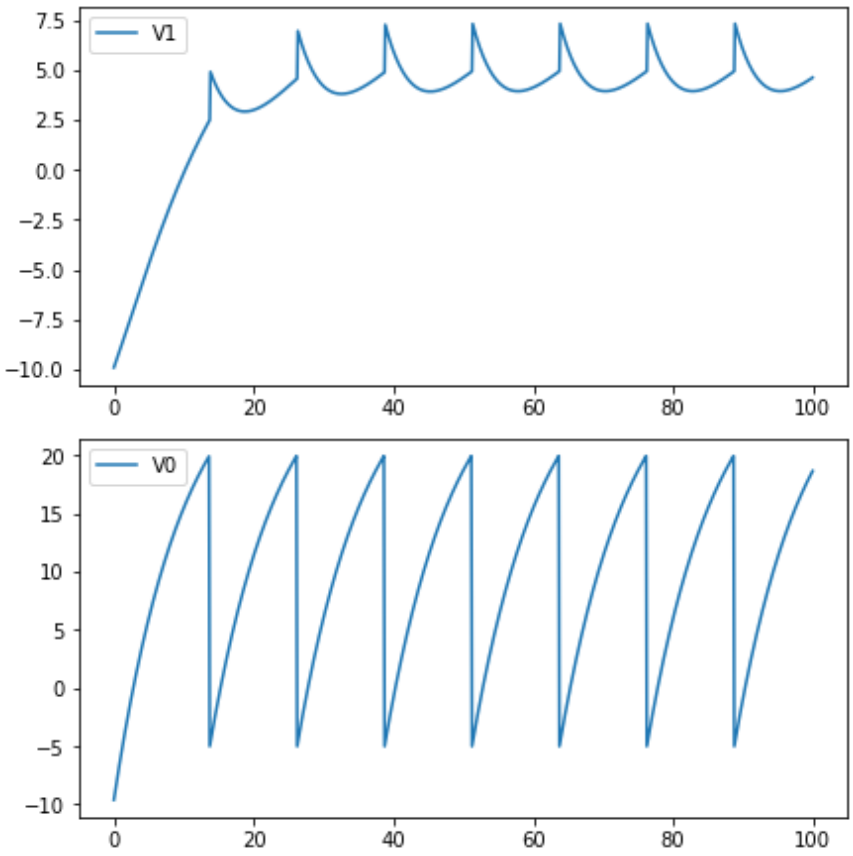
```
class Gap_junction(bp.TwoEndConn):
  target_backend = ['numpy', 'numba']

  def __init__(self, pre, post, conn, delay=0., k_spikelet=0.1,
               post_refractory=False, **kwargs):
    self.delay = delay
    self.k_spikelet = k_spikelet
    self.post_has_refractory = post_refractory

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

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

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

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

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

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

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

```

$I_{post} = w\,(V_{pre} - V_{post})$

Then we can run a simulation.

```
import matplotlib.pyplot as plt

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

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

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

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

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

```
plt.legend()
plt.show()
```



References

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

2. Sterratt, David, et al. Principles of computational 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 firing with short intervals, the response of the postsynaptic neurons becomes weaker and weaker, showing a short term depression. But the response recovers after a short period of time, so this plasticity is short-term.

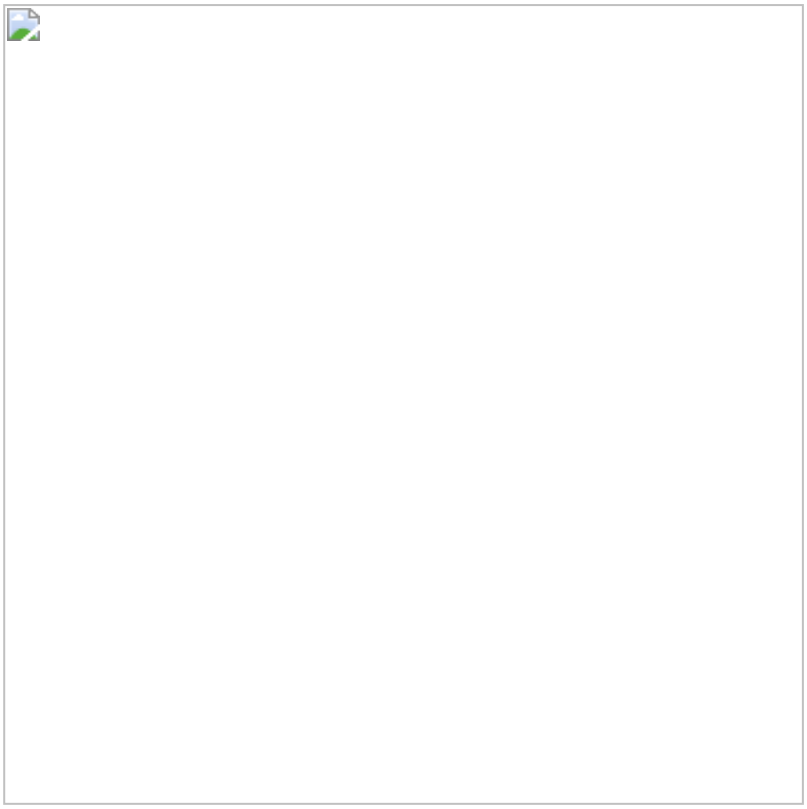


Fig. 2-1 Short-term plasticity. (Adapted from *Gerstner et al., 2014* [1])

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,

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

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

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

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

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. τ_f and τ_d are time constants of u and x, respectively.

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

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

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

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

  def __init__(self, pre, post, conn, delay=0., U=0.15, tau_f=1500.,
               tau_d=200., tau=8., **kwargs):
    # parameters
    self.tau_d = tau_d
    self.tau_f = tau_f
    self.tau = tau
    self.U = U
    self.delay = delay

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

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

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

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


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

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

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

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

$$\begin{cases} \frac{ds}{dt} = -s/\tau \\ \frac{du}{dt} = -u/\tau_f \\ \frac{dx}{dt} = (1-x)/\tau_d \end{cases}$$

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

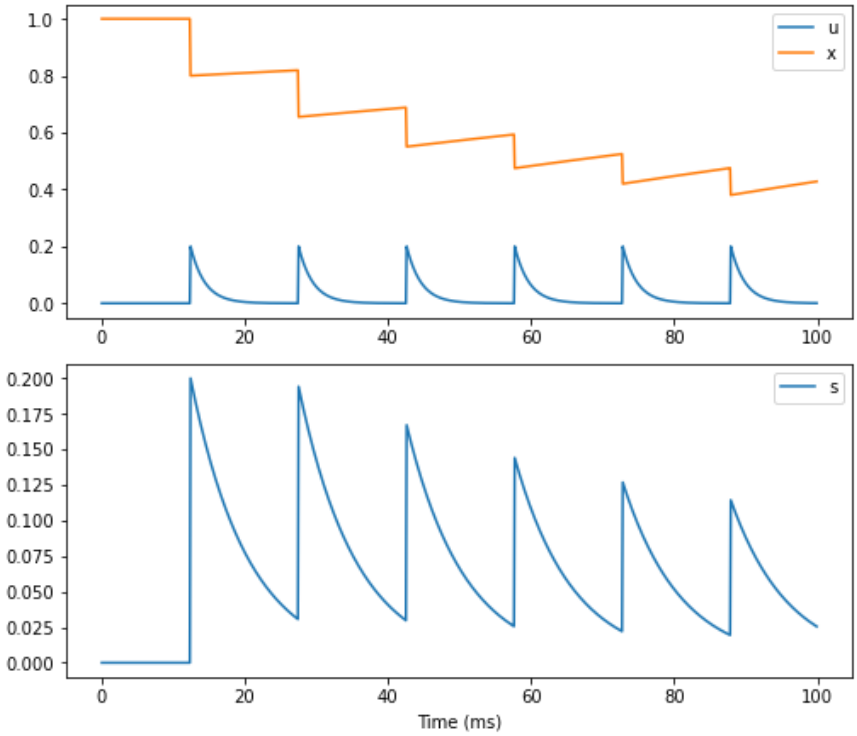
Then let's define a function to run the code. Like synapse models, we need two neuron groups to be connected. Besides the dynamic of s, we also want to see how u and x changes over time, so we monitor 's', 'u'

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` .

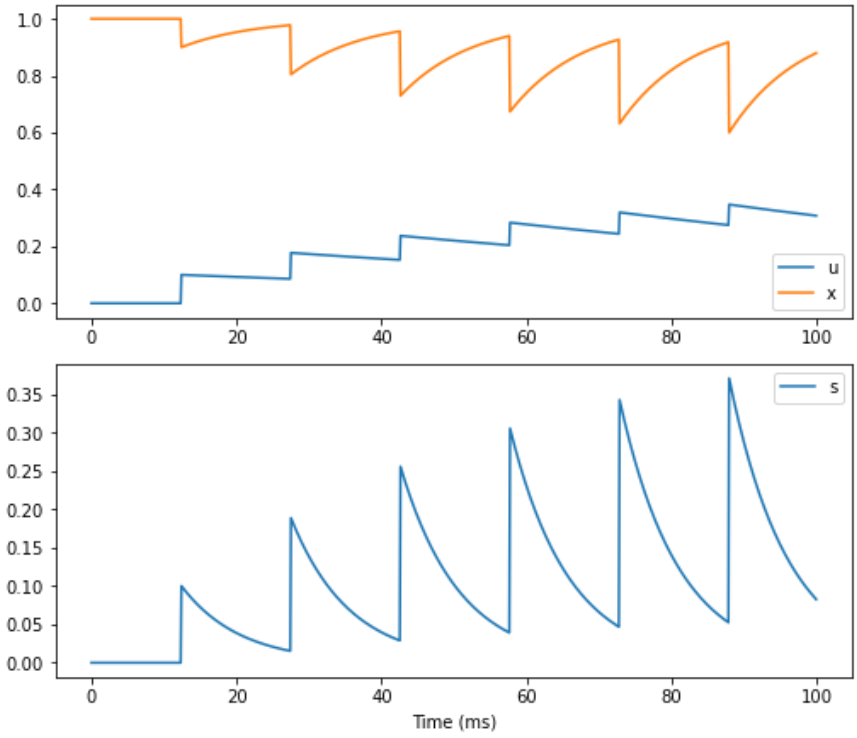
```
run_stp(U=0.2, tau_d=150., tau_f=2.)
```



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* [2])

The computational model of STDP is given by,

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

      self.s[i], A_s, A_t = self.integral(self.s[i], self.A_s[i],
                                          self.A_t[i], _t, self.tau,
                                          self.tau_s, self.tau_t)

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

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

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

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

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

$$\begin{cases} \frac{ds}{dt} = -s/\tau \\ \frac{dA_s}{dt} = -A_s/\tau_s \\ \frac{dA_t}{dt} = -A_t/\tau_t \end{cases}$$

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

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

Limit w within the boundary

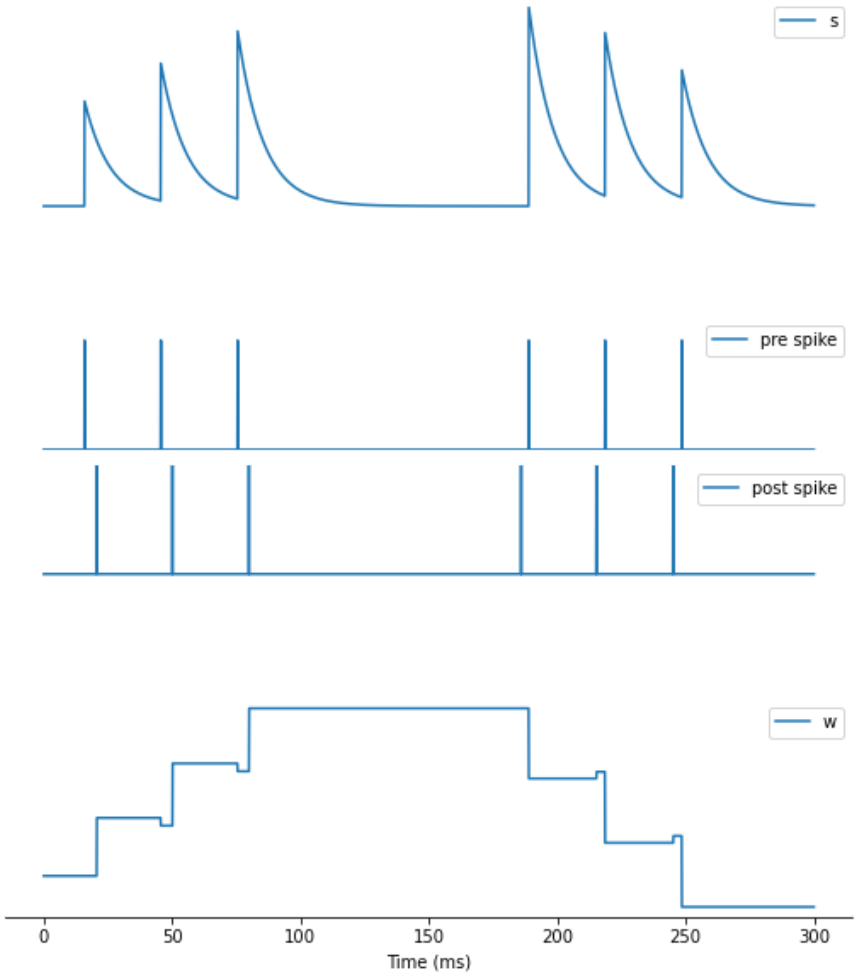
We control the spike timing by varying the input current of the presynaptic group and postsynaptic group. We apply the first input to the presynaptic group starting at $t = 5ms$ (with amplitude of 30 μA, lasts for 15 ms to ensure to induce a spike with LIF neuron model), then start to stimulate the postsynaptic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

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

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

Then let's run the simulation.

```
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, 'in

# 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.

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

The 6th term contains r_ir_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.

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

γ represents the learning rate.

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

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

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

  def __init__(self, pre, post, conn, gamma=.005, w_max=1., w_min=0., **kwargs):
    # params
    self.gamma = gamma
    self.w_max = w_max
    self.w_min = w_min
    # no delay in firing rate models

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

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

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

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

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

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

bp.TwoEndConn class: Learning rule occur between two neuron groups

$\frac{dw}{dt} = \gamma(r_ir_j - w\,r_i^2)$

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

Rate model: $r_i = \sum_j w_{ij}\ r_j$

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

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

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

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

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

We need a firing rate neuron with parameter r

We only need a simple unit to test oja's rule.

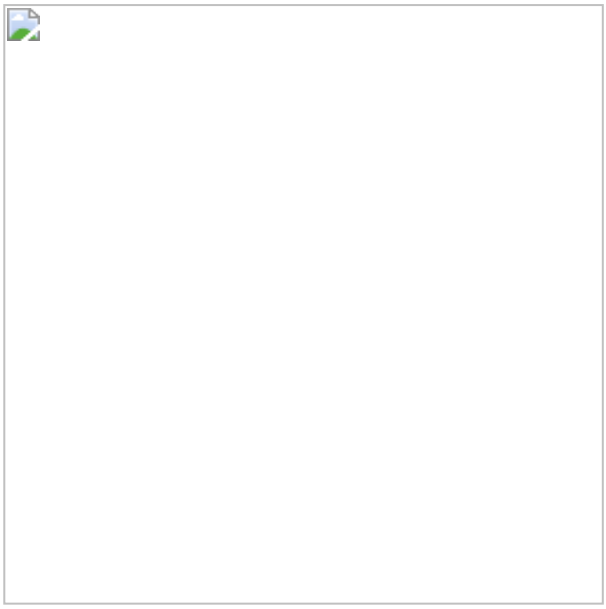


Fig. 2-3 Connection of neuron groups.

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.

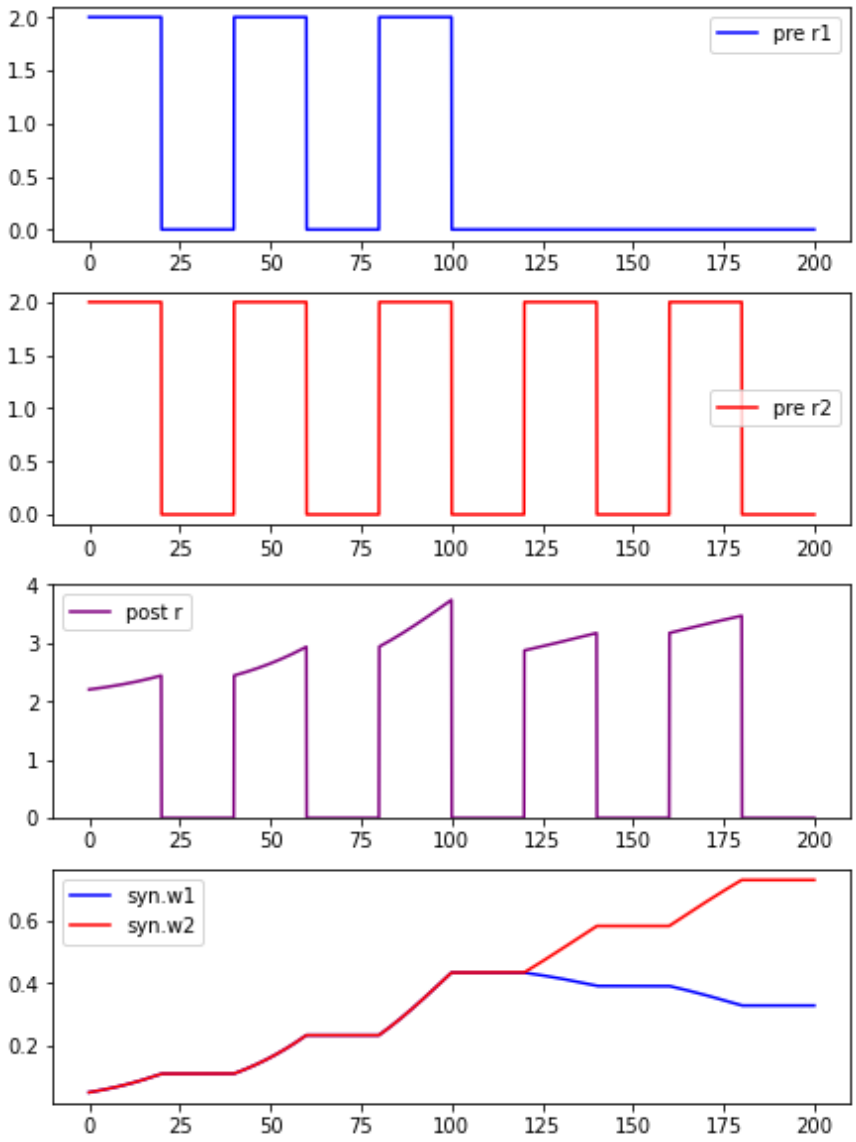
```python
# create input
current1, _ = bp.inputs.constant_current([(2., 20.), (0., 20.)] * 3 +
                                         [(0., 20.), (0., 20.)] * 2)
current2, _ = bp.inputs.constant_current([(2., 20.), (0., 20.)] * 5)
current3, _ = bp.inputs.constant_current([(2., 20.), (0., 20.)] * 5)
current_pre = np.vstack((current1, current2))
current_post = np.vstack((current3, current3))

# simulate
neu_pre = neu(2, monitors=['r'])
neu_post = neu(2, monitors=['r'])
syn = Oja(pre=neu_pre, post=neu_post, conn=bp.connect.All2All(), monitors=['w'])
net = bp.Network(neu_pre, syn, neu_post)
net.run(duration=200., inputs=[(neu_pre, 'r', current_pre.T, '='),
                               (neu_post, 'r', current_post.T)])

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

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

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

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

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

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.

index [:, 0], group 1

index [:, 1], group 2



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 η represents the learning rate, and r_θ 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_θ.

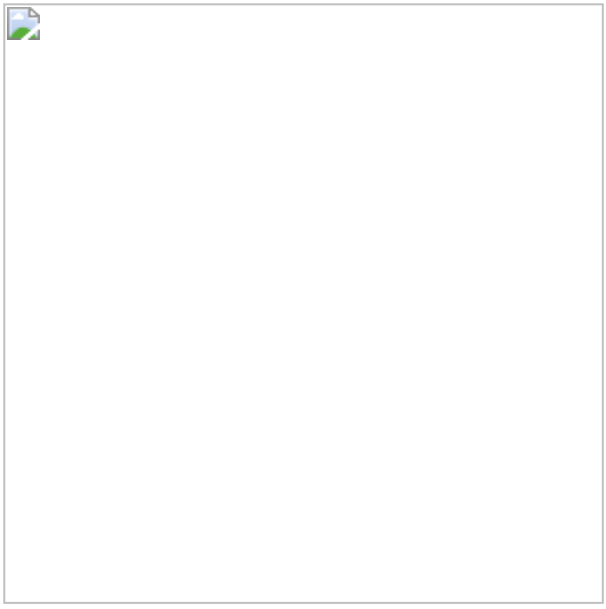


Fig. 2-4 BCM rule (From *Gerstner et al., 2014* [1])

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.

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

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

  def __init__(self, pre, post, conn, lr=0.005, w_max=1., w_min=0., **kwargs):
    # parameters
    self.lr = lr
    self.w_max = w_max
    self.w_min = w_min
    self.dt = bp.backend.get_dt()

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

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

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

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

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

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

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

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

$$\frac{dw}{dt} = \eta r_i (r_i - r_\theta) r_j$$

$$r_\theta = \frac{\sum_t dt r_i}{T}$$

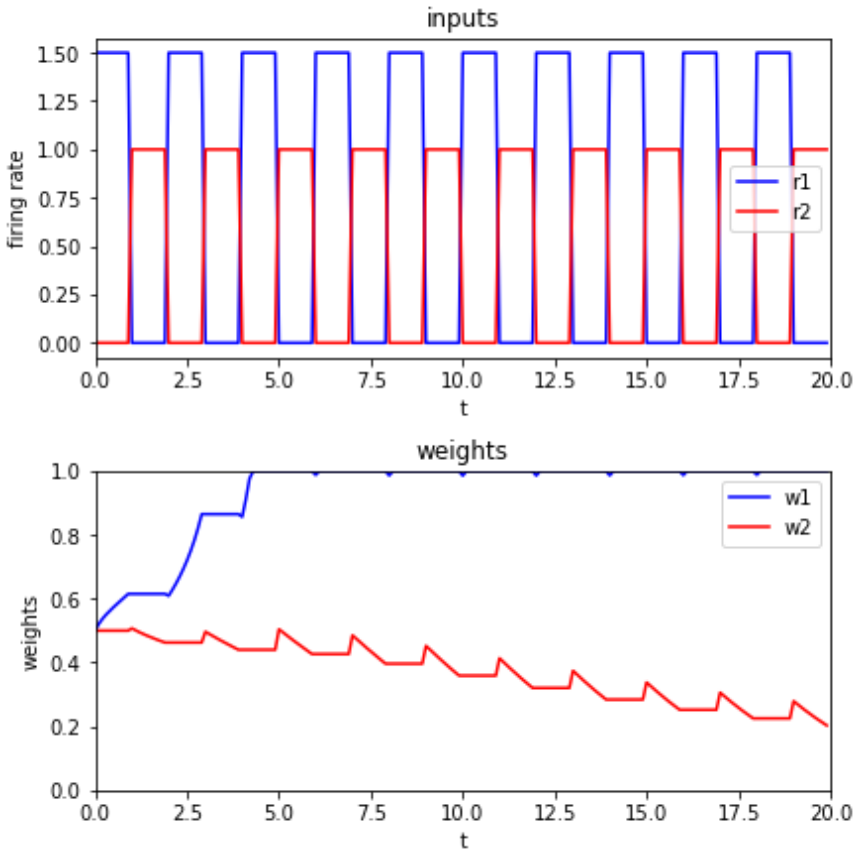
self.dt +1 since self.dt begin with 0

Limit w within the boundary

Rate model: $r_i = \sum_j w_{ij}\ r_j$

Then we can run the simulation with the following code.

```
n_post = 1
n_pre = 20

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

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

> [2]. 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 there 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.

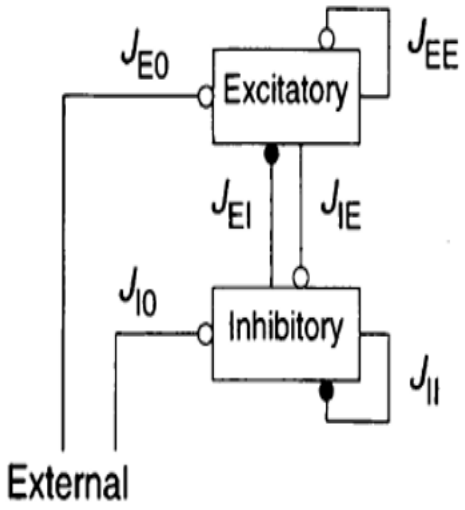


Fig.3-1 Structure of E/I balanced network (*Vreeswijk and Sompolinsky, 1996* [1])

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

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

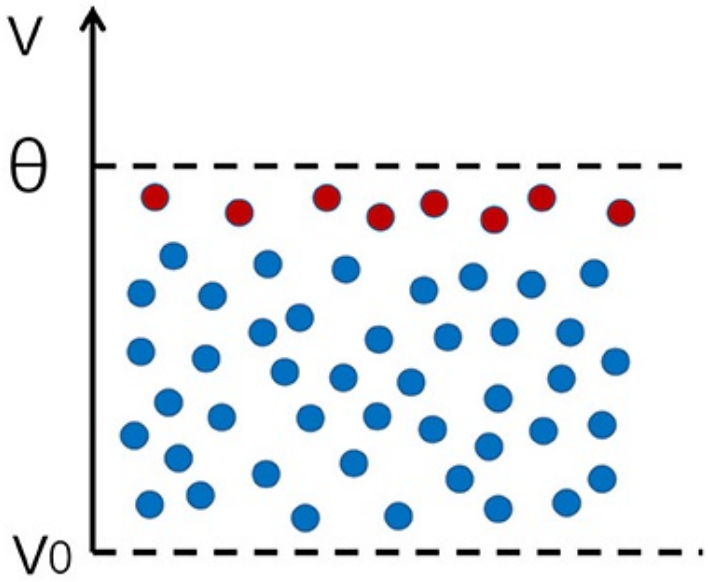


Fig.3-2 Distribution of neuron membrane potentials in E/I balanced network (*Tian et al.，2020* [2])

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.

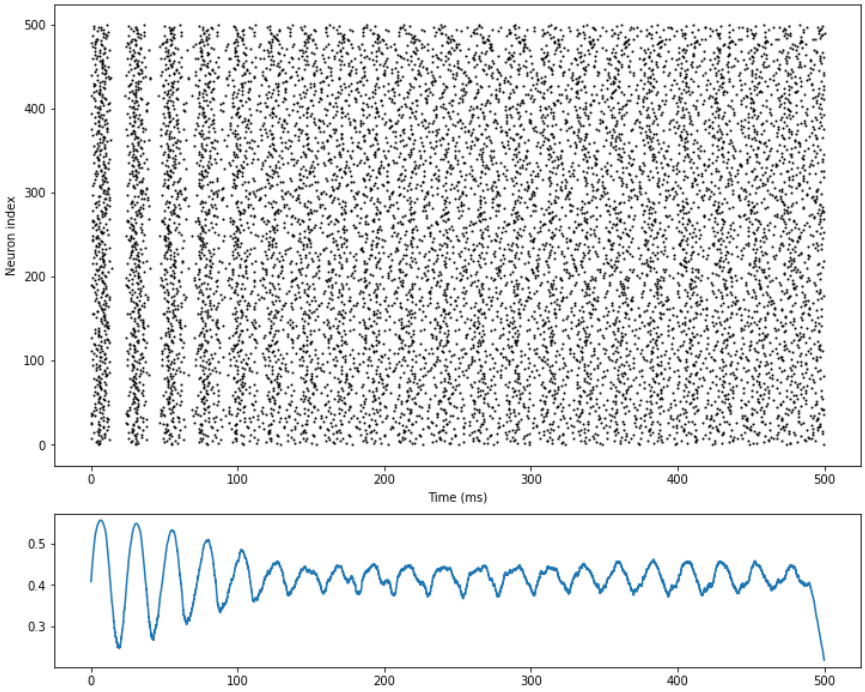


Fig.3-3 E/I balanced net raster plot

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.

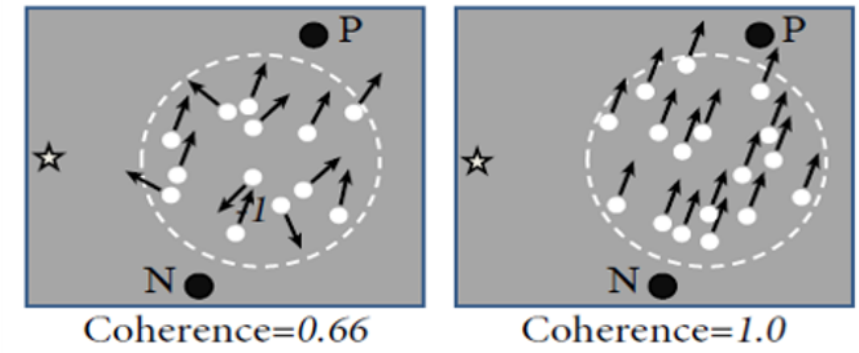


Fig.3-4 Experimental Diagram (*Gerstner et al., 2014*[3])

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.

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

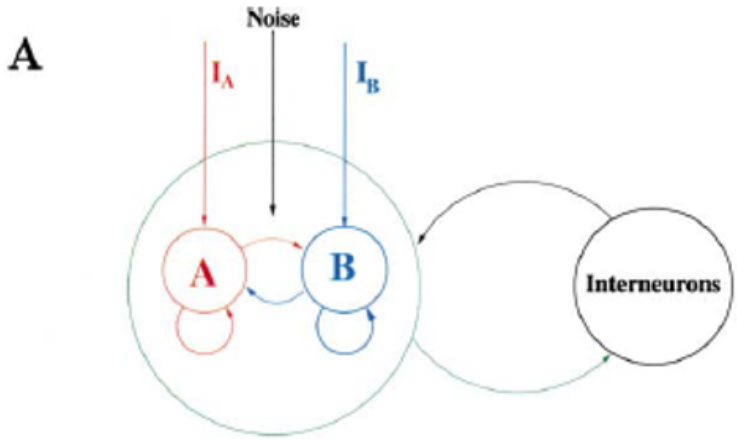
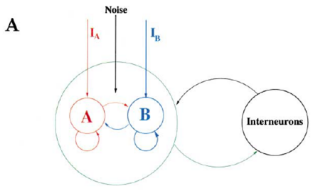


Fig.3-5 structure of decision making network (*Wang，2002*[4])

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

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

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

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

```



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.

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.

```
488  # def background poisson input
489  class PoissonInput(bp.NeuGroup):
490    target_backend = 'general'
491
492    def __init__(self, size, freqs, dt, **kwargs):
493      self.freqs = freqs
494      self.dt = dt
495
496      self.spike = bp.ops.zeros(size, dtype=bool)
497
498      super(PoissonInput, self).__init__(size=size, **kwargs)
499
500    def update(self, _t):
501      self.spike = np.random.random(self.size) \
502                   < self.freqs * self.dt / 1000.
503
504  # poisson_freq = 2400Hz
505  neu_poisson_A = PoissonInput(N_A, freqs=poisson_freq, dt=dt)
506  neu_poisson_B = PoissonInput(N_B, freqs=poisson_freq, dt=dt)
507  neu_poisson_non = PoissonInput(N_non, freqs=poisson_freq, dt=dt)
508  neu_poisson_I = PoissonInput(N_I, freqs=poisson_freq, dt=dt)
509
510  syn_back2A_AMPA = AMPA(pre=neu_poisson_A, post=neu_A,
511                         conn=bp.connect.One2One())
512  syn_back2B_AMPA = AMPA(pre=neu_poisson_B, post=neu_B,
513                         conn=bp.connect.One2One())
514  syn_back2non_AMPA = AMPA(pre=neu_poisson_non, post=neu_non,
515                           conn=bp.connect.One2One())
516
517  syn_back2I_AMPA = AMPA(pre=neu_poisson_I, post=neu_I,
518                         conn=bp.connect.One2One())
519
```

Generate spikes with a given Poisson frequency.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  def update(self, _t):
    if self.stim_start_t < _t < self.stim_end_t:
      if self.stim_change_freq_interval <= _t - self.t_last_change_freq:
        self.freq = np.random.normal(self.mean_freq, self.var_freq)
        self.freq = max(self.freq, 0)
        self.t_last_change_freq = _t
      self.spike = np.random.random(self.size) < (self.freq * self.dt / 1000)
    else:
      self.freq = 0.
      self.spike[:] = False


neu_input2A = PoissonStim(N_A, dt=dt, t_start=pre_period,
                          t_end=pre_period + stim_period,
                          t_interval=50., mean_freq=mu_A, var_freq=10.,
                          monitors=['freq'])
neu_input2B = PoissonStim(N_B, dt=dt, t_start=pre_period,
                          t_end=pre_period + stim_period,
                          t_interval=50., mean_freq=mu_B, var_freq=10.,
                          monitors=['freq'])

syn_input2A_AMPA = AMPA(pre=neu_input2A, post=neu_A,
                        conn=bp.connect.One2One())

syn_input2B_AMPA = AMPA(pre=neu_input2B, post=neu_B,
                        conn=bp.connect.One2One())
```

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.

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

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

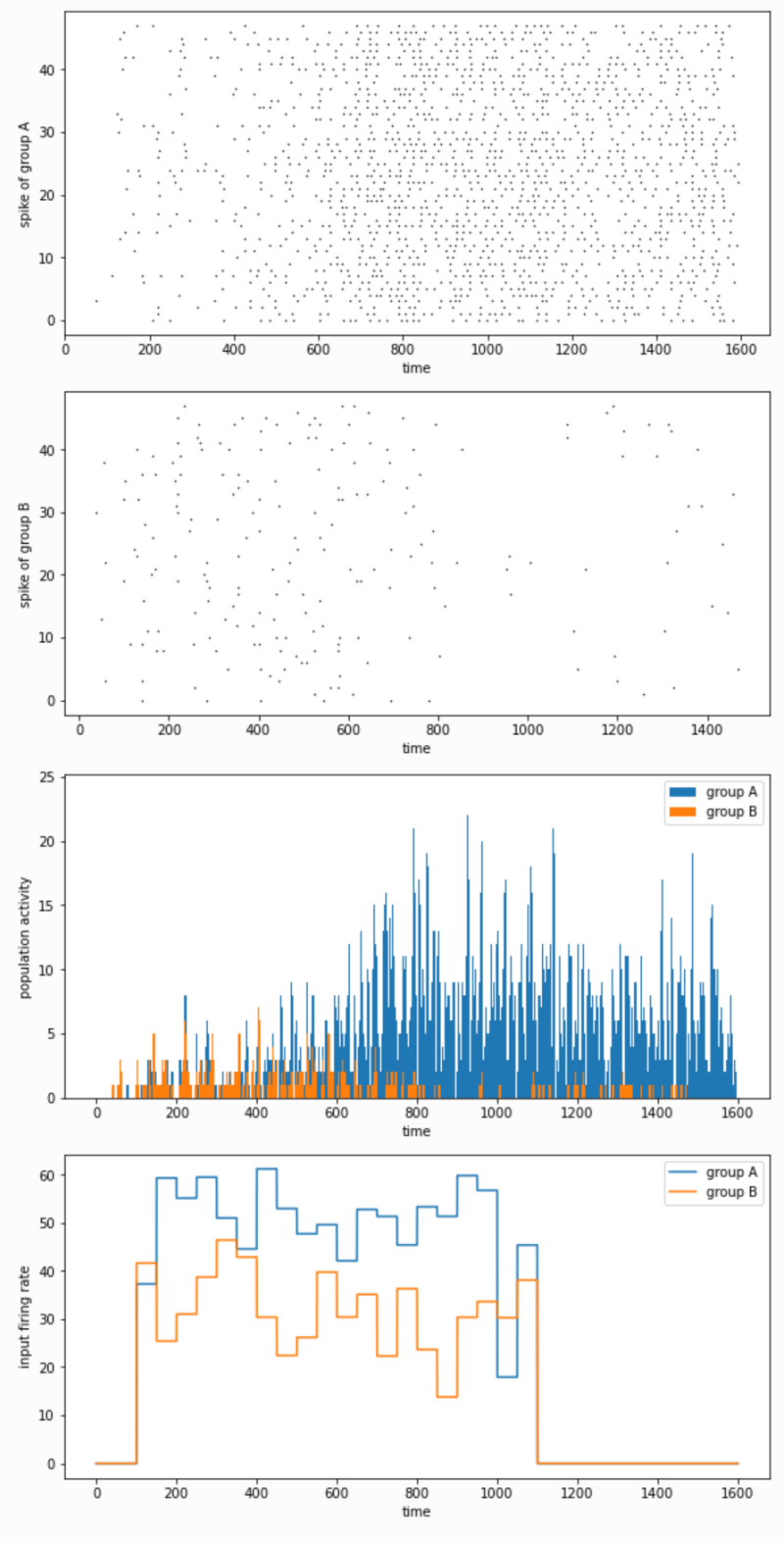
Generate Poisson inputs with the Poisson frequency of this neuron.

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. Van Vreeswijk, Carl, and Haim Sompolinsky. "Chaos in neuronal networks with balanced excitatory and inhibitory activity." *Science* 274.5293 (1996): 1724-1726. ↩

[2] . Tian, Gengshuo, et al. "Excitation-Inhibition Balanced Neural Networks for Fast Signal Detection." *Frontiers in Computational Neuroscience* 14 (2020): 79. ↩

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

[4] . Wang, Xiao-Jing. "Probabilistic decision making by slow reverberation in cortical circuits." *Neuron* 36.5 (2002): 955-968. ↩

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)[1] 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.

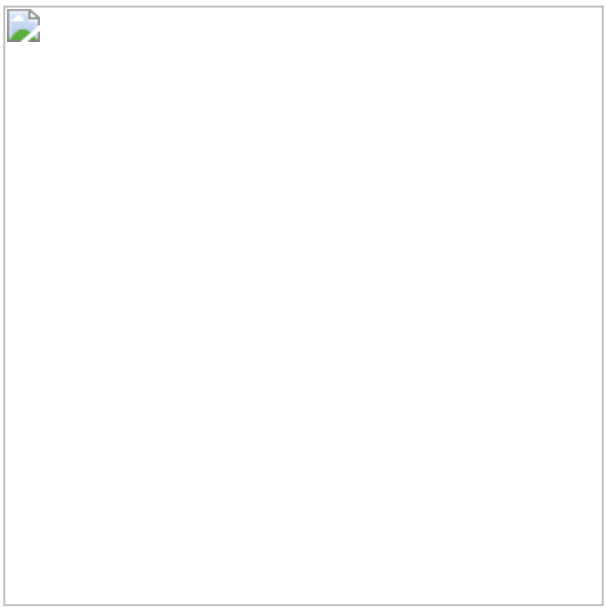


Fig. 3-1 Reduced decision model. (From *Wong & Wang, 2006* [1])

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 μ_0 and a coherence c'. The higher the coherence, the more definite S_1 is the correct answer, while the lower the coherence, the more random it is. The formula is as follows:

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

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

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.

```python
from collections import OrderedDict
import brainpy as bp

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


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

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

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

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

    return ds1dt, ds2dt
```

$I_1 = J_{Aext}\mu_0(1 + c')$
$I_2 = J_{Aext}\mu_0(1 - c')$

$I_{syn,1} = J_{11}S_1 - J_{12}S_2 + I_0 + I_1$
$r_1 = \frac{aI_{syn,1} - b}{1 - exp(-d(aI_{syn,1} - b))}$
$\frac{dS_1}{dt} = -S_1/\tau + (1 - S_1)\gamma r_1$

$I_{syn,2} = J_{22}S_2 - J_{21}S_1 + I_0 + I_2$
$r_2 = \frac{aI_{syn,2} - b}{1 - exp(-d(aI_{syn,2} - b))}$
$\frac{dS_2}{dt} = -S_2/\tau + (1 - S_2)\gamma r_2$

```
  def __init__(self, size, coh, JAext=.00117, J_rec=.3725, J_inh=.1137,
               I_0=.3297, a=270., b=108., d=0.154, tau_s=.06, gamma=0.641,
               **kwargs):
    # parameters
    self.coh = coh
    self.JAext = JAext
    self.J_rec = J_rec
    self.J_inh = J_inh
    self.I0 = I_0
    self.a = a
    self.b = b
    self.d = d
    self.tau_s = tau_s
    self.gamma = gamma

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

    self.integral = bp.odeint(f=self.derivative, method='rk4', dt=0.01)

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

  def update(self, _t):
    for i in range(self.size):
      self.s1[i], self.s2[i] = self.integral(self.s1[i], self.s2[i], _t,
                                             self.input[i], self.coh,
                                             self.JAext, self.J_rec,
                                             self.J_inh, self.I0,
                                             self.a, self.b, self.d,
                                             self.tau_s, self.gamma)
      self.input[i] = 0.
```

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

```
def phase_analyze(I, coh):
  decision = Decision(1, coh=coh)

  phase = bp.analysis.PhasePlane(decision.integral,
                                 target_vars=OrderedDict(s2=[0., 1.],
                                                         s1=[0., 1.]),
                                 fixed_vars=None,
                                 pars_update=dict(I=I, coh=coh,
                                                  JAext=.00117, J_rec=.3725,
                                                  J_inh=.1137, I_0=.3297,
                                                  a=270., b=108., d=0.154,
                                                  tau_s=.06, gamma=0.641),
                                 numerical_resolution=.001,
                                 options={'escape_sympy_solver': True})

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

Functions defining the differential equations (from decision model)

Variables to be showed in phase plane.

Specify values of the parameters.

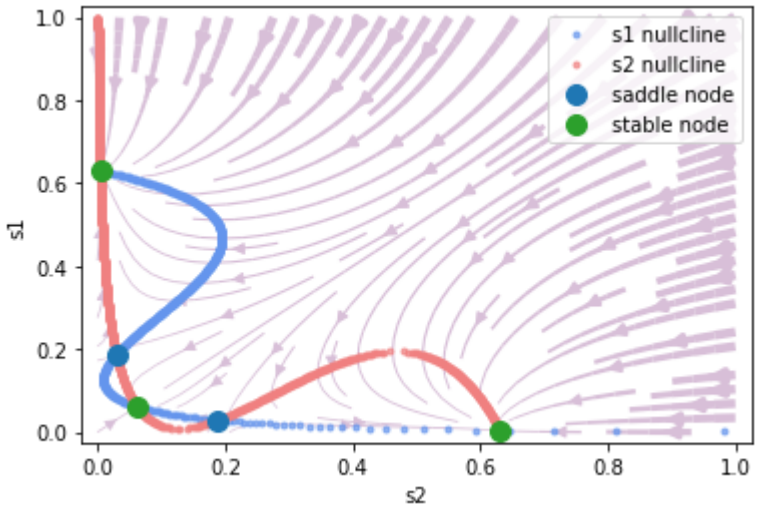
Since the differential equations in the model are very complicated, here we use numerical solutions, skip the analytical solutions provided by sympy, and set the numerical resolution.

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

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

Output:

```
plot nullcline ...
plot fixed point ...
Fixed point #1 at s2=0.06176109215560733, s1=0.061761097890
Fixed point #2 at s2=0.029354239100062428, s1=0.18815448592
Fixed point #3 at s2=0.0042468423702408655, s1=0.630304569
Fixed point #4 at s2=0.6303045696241589, s1=0.0042468423702
Fixed point #5 at s2=0.18815439944520335, s1=0.029354240536
plot vector field ...
```



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

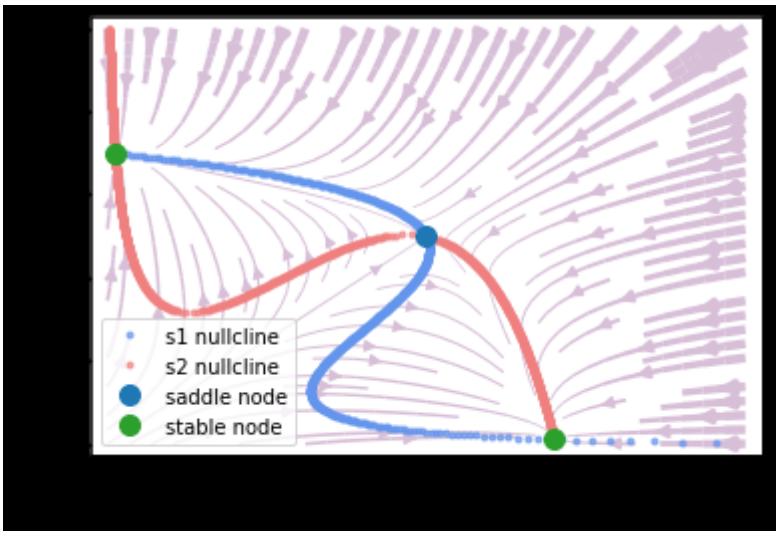
Here, the x-axis is S_2 which represents choice 2, and the y-axis is S_1, which represents choice 1. As you can see, the upper-left fixed point represents choice 1, the lower-right fixed point represents choice 2, and the lower-left fixed point represents no choice.

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

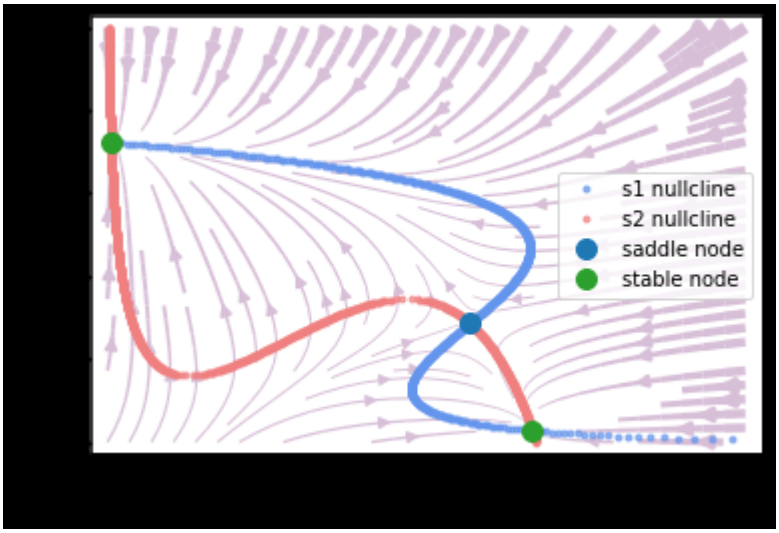
Now let's look at the phase plane under different coherences when we fix the external input strength to 30.

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

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

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

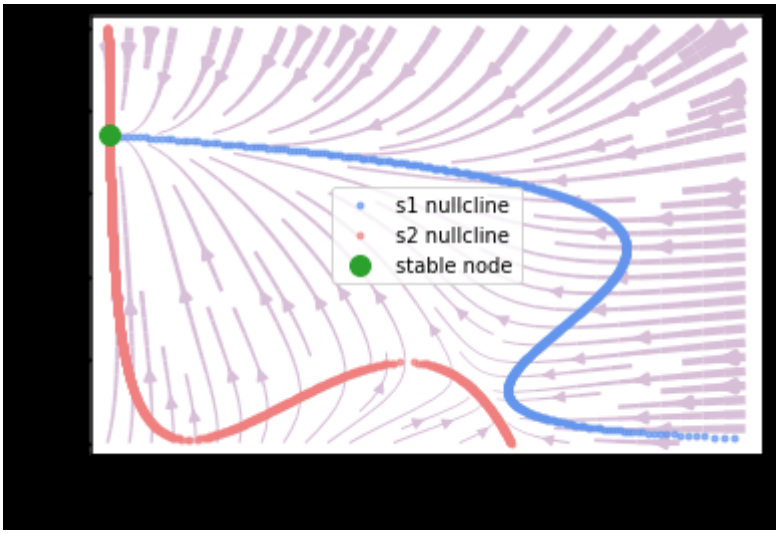
```
coherence = 0%
plot nullcline ...
plot fixed point ...
Fixed point #1 at s2=0.6993504413889349, s1=0.0116220495267
Fixed point #2 at s2=0.49867489858358865, s1=0.498674898583
Fixed point #3 at s2=0.011622051540013889, s1=0.69935043555
plot vector field ...
```




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




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



3.2.2 CANN

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

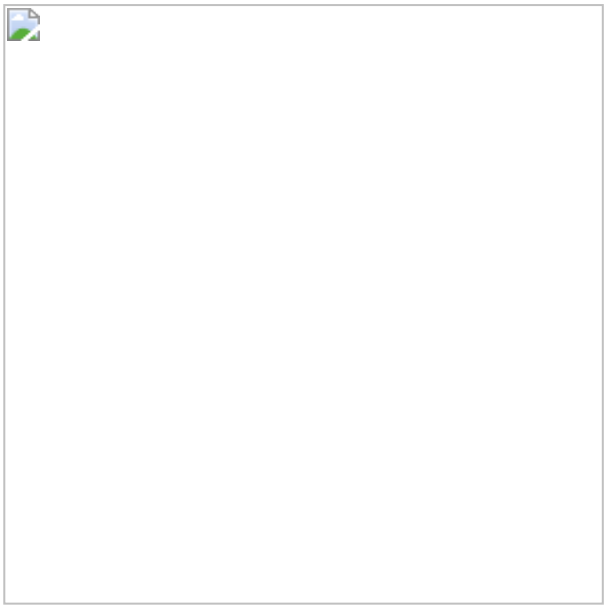


Fig. 3-2 Structure of CANN. (From *Wu et al., 2008* [2])

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:

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

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`.

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

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

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

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

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

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

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

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

```

z_range denotes the range of x.

$\rho \int dx'$

Then we define the functions.

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

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

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

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

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

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

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

Distances on the ring:
The farthest distance is half of z_range, so d should not exceed 0.5 * z_range

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

	x_1	x_2	...
x_1		d_{12}	
x_2	d_{21}		
...			

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

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

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

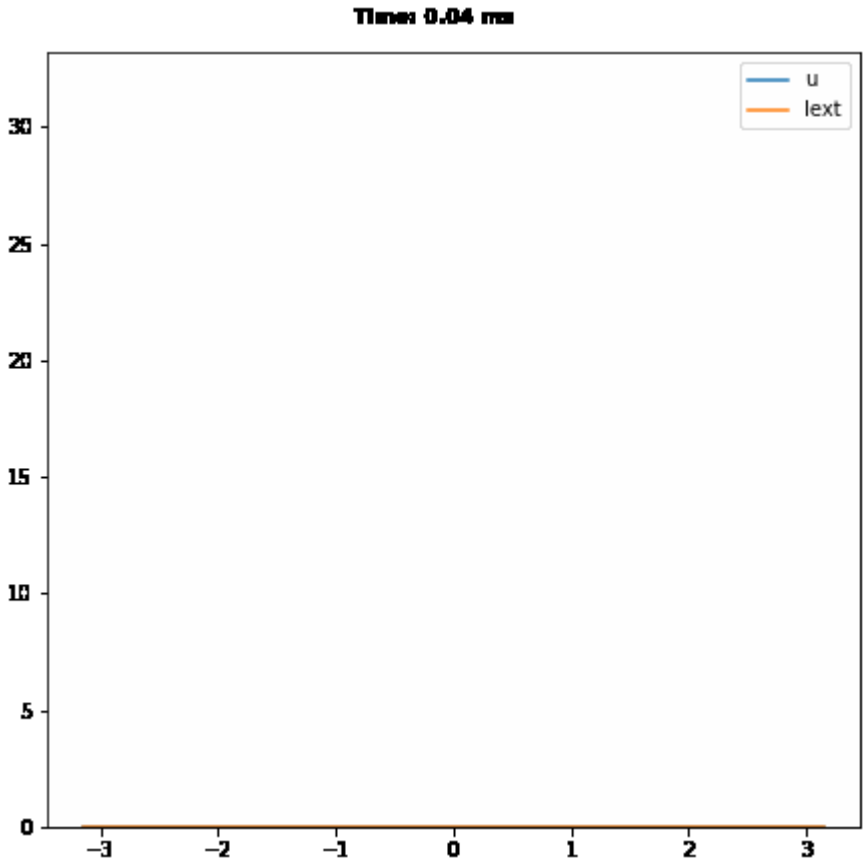
same points (the actual maximum value is π, that is, half of `z_range`, the distance exceeded needs to be subtracted from a `z_range`). We use the `dist` function to handle the distance on the ring.

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

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

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

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

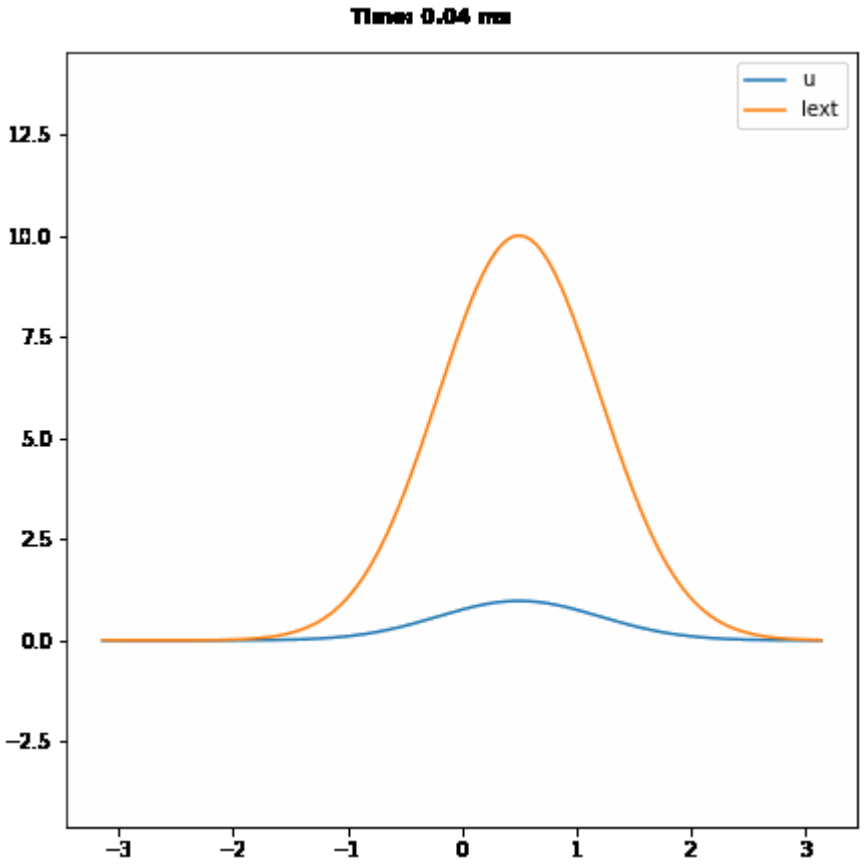
```
# define function
def plot_animate(frame_step=5, frame_delay=50):
    bp.visualize.animate_1D(dynamical_vars=[{'ys': cann.mor
                                             'legend': 'u'}
                                             'xs': cann.x,
                            frame_step=frame_step, frame_de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


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



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

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

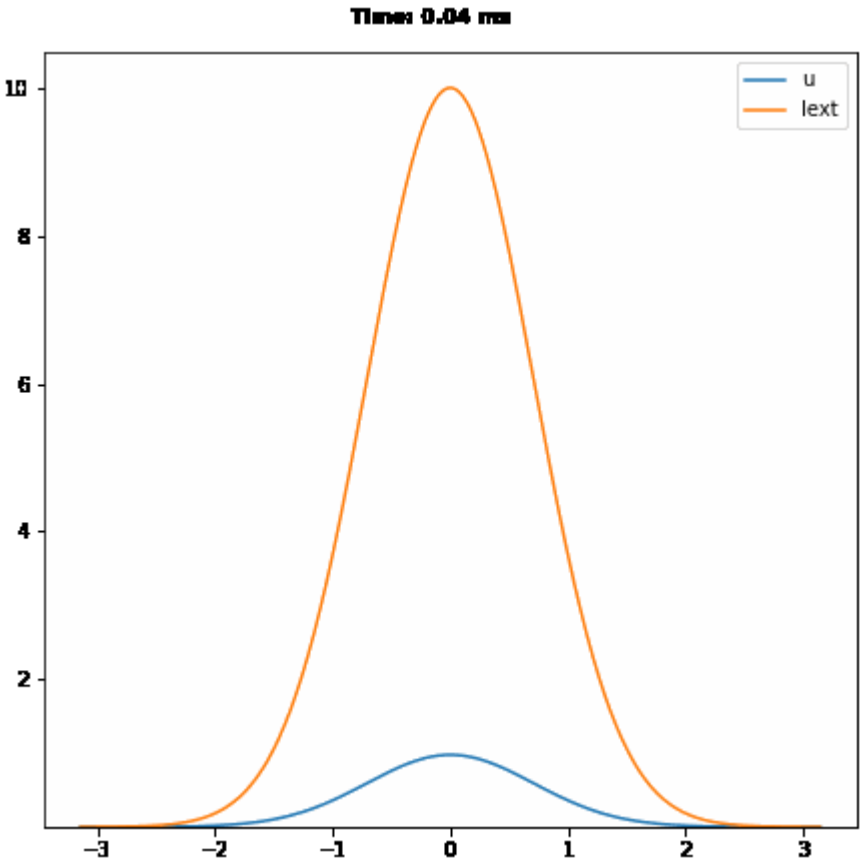
```
cann = CANN1D(num=512, k=8.1, monitors=['u'])

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

plot_animate()
```



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

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

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

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

plot_animate()
```



Reference

[1]. Wong, K.-F. & Wang, X.-J. A Recurrent Network Mechanism of Time Integration in Perceptual Decisions. J. Neurosci. 26, 1314–1328 (2006). ↩

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

Biophysical models

Hodgkin-Huxley model

```
import brainpy as bp
from numba import prange

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

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

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

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

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

    return dVdt, dmdt, dhdt, dndt

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

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

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

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

import brainpy as bp

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

Reduced models

LIF model

```
import brainpy as bp
from numba import prange

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

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

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

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

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

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

```
import brainpy as bp

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

QualF model

```
import brainpy as bp
from numba import prange

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

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

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

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

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

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

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


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

ExpIF model

```
import brainpy as bp
from numba import prange

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

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

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

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

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

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

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

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

AdExIF model

```
import brainpy as bp
from numba import prange

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

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

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

    return dVdt, dwdt

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

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

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

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

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

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

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

Hindmarsh-Rose model

```
import brainpy as bp
from numba import prange


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

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

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

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

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

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

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

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

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



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

GeneralizedIF model

```
import brainpy as bp
from numba import prange

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

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

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

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

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

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

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

# set parameters
num2mode = ["tonic_spiking", "class_1", "spike_frequency_ad
            "phasic_spiking", "accomodation", "threshold_va
            "rebound_spike", "class_2", "integrator",
            "input_bistability", "hyperpolarization_induced
            "tonic_bursting", "phasic_bursting", "rebound_b
            "mixed_mode", "afterpotentials", "basal_bistabi
            "preferred_frequency", "spike_latency"]

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

```
        "a": 0.005,
        "input": [(1.5, 100.), (0, 500.), (0.5, 100.),
                  (1., 100.), (1.5, 100.), (0., 100.)]
    },
    "threshold_variability": {
        "a": 0.005,
        "input": [(1.5, 20.), (0., 180.), (-1.5, 20.),
                  (0., 20.), (1.5, 20.), (0., 140.)]
    },
    "rebound_spike": {
        "a": 0.005,
        "input": [(0, 50.), (-3.5, 750.), (0., 200.)]
    },
    "class_2": {
        "a": 0.005,
        "input": [(2 * (1. + 1e-6), 200.)],
        "V_th": -30.
    },
    "integrator": {
        "a": 0.005,
        "input": [(1.5, 20.), (0., 10.), (1.5, 20.), (0., 2
                  (1.5, 20.), (0., 30.), (1.5, 20.), (0., 3
    },
    "input_bistability": {
        "a": 0.005,
        "input": [(1.5, 100.), (1.7, 400.),
                  (1.5, 100.), (1.7, 400.)]
    },
    "hyperpolarization_induced_spiking": {
        "V_th_reset": -60.,
        "V_th_inf": -120.,
        "input": [(-1., 400.)],
        "V_th": -50.
    },
    "hyperpolarization_induced_bursting": {
        "V_th_reset": -60.,
        "V_th_inf": -120.,
        "A1": 10.,
        "A2": -0.6,
        "input": [(-1., 400.)],
        "V_th": -50.
    },
    "tonic_bursting": {
        "a": 0.005,
        "A1": 10.,
        "A2": -0.6,
        "input": [(2., 500.)]
    },
```

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


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

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

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

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

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


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

Firing rate models

Firing Rate Unit model

```
import brainpy as bp
from numba import prange

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

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

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

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

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

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

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

Appendix: Synapses

Synapse models

AMPA

```
import brainpy as bp


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

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

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

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

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

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

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

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

```
import brainmodels as bm

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

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

  syn = syn_model(pre=neu1, post=neu2, conn=bp.connect.All2
                  monitors=['s'], **kwargs)

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

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

NMDA

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

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

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

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

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

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

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

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

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

      # output
```

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

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

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

```
run_syn(NMDA)
```

GABA_b

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

  @staticmethod
  def derivative(R, G, t, k3, TT, k4, k1, k2):
    dRdt = k3 * TT * (1 - R) - k4 * R
    dGdt = k1 * R - k2 * G
    return dRdt, dGdt

  def __init__(self, pre, post, conn, delay=0., g_max=0.02,
               k1=0.18, k2=0.034, k3=0.09, k4=0.0012, kd=10
               T_duration=0.3, **kwargs):
    # params
    self.g_max = g_max
    self.E = E
    self.k1 = k1
    self.k2 = k2
    self.k3 = k3
    self.k4 = k4
    self.kd = kd
    self.T = T
    self.T_duration = T_duration

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

    # data
    self.R = bp.ops.zeros(self.size)
    self.G = bp.ops.zeros(self.size)
    self.t_last_pre_spike = bp.ops.ones(self.size) * -1e7
    self.s = bp.ops.zeros(self.size)
    self.g = self.register_constant_delay('g', size=self.si
                                          delay_time=delay)

    self.integral = bp.odeint(f=self.derivative, method='rk
    super(GABAb, self).__init__(pre=pre, post=post, **kwarg

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

      if self.pre.spike[pre_id]:
        self.t_last_pre_spike[i] = _t
      TT = ((_t - self.t_last_pre_spike[i]) < self.T_durati
```

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

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

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

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

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

Differences of two exponentials

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

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

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

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

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

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

    super(Two_exponentials, self).__init__(pre=pre, post=pc

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

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

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

Alpha

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

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

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

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

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

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

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

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

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

```
run_syn(Alpha)
```

Single exponential decay

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

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

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

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

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

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

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

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

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

```
run_syn(Exponential)
```

Voltage jump

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

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

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

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

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

```
run_syn(Voltage_jump)
```

Gap junction

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

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

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

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

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

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

      self.post.input[post_id] += self.w[i] * (self.pre.V[p
                                               self.post.V[

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

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

```
import matplotlib.pyplot as plt

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

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

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

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

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

Synaptic plasticity

STP

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

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

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

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

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

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

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

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

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

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

```
        self.x[i] = x

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

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

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

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

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

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

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

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

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

STDP

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

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

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

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

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

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

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

      self.s[i], A_s, A_t = self.integral(self.s[i], self.A
                                          self.A_t[i], _t,
                                          self.tau_s, self.
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Oja's rule

```
import numpy as np

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


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

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

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

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

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

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

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

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

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

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

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

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

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

# simulate
neu_pre = neu(2, monitors=['r'])
neu_post = neu(2, monitors=['r'])
syn = Oja(pre=neu_pre, post=neu_post, conn=bp.connect.All2A
net = bp.Network(neu_pre, syn, neu_post)
net.run(duration=200., inputs=[(neu_pre, 'r', current_pre.T
                               (neu_post, 'r', current_post

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

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

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

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

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

BCM rule

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

plt.show()
```

Spiking neural networks

E/I balanced network

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

bp.backend.set('numba')

N_E = 500
N_I = 500
prob = 0.1

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

tau_decay = 2.

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

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

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

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

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

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

Decision making network

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

Wang, Xiao-Jing. "Probabilistic decision making by slow
reverberation in cortical circuits." Neuron 36.5 (2002): 95
"""
import brainpy as bp
import numpy as np
import matplotlib.pyplot as plt

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

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

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

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

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

```
print(f"R_E * C_E = {R_E * C_E} should be equal to tau_E =

V_rest_I = -70.  # mV
V_reset_I = -55.  # mV
V_th_I = -50.  # mV
g_I = 20. * 1e-3  # uS
R_I = 1 / g_I  # Mohm
C_I = 0.2  # nF
tau_I = 10.  # ms
t_refractory_I = 1.  # ms
print(f"R_I * C_I = {R_I * C_I} should be equal to tau_I =


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

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

  def __init__(self, size, V_rest=0., V_reset=0.,
               V_th=0., R=0., tau=0., t_refractory=0.,
               **kwargs):
    self.V_rest = V_rest
    self.V_reset = V_reset
    self.V_th = V_th
    self.R = R
    self.tau = tau
    self.t_refractory = t_refractory

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

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

  def update(self, _t):
    # update variables
    not_ref = (_t - self.t_last_spike > self.t_refractory)
    self.V[not_ref] = self.integral(
      self.V[not_ref], _t, self.input[not_ref],
      self.V_rest, self.R, self.tau)
    sp = (self.V > self.V_th)
    self.V[sp] = self.V_reset
```

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


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

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

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

delay_syn = 0.5  # ms


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

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

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

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

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

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

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

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


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

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

  def __init__(self, pre, post, conn, delay=0.,
               g_max=0.10, E=0., tau=2.0, **kwargs):
    # parameters
    self.g_max = g_max
    self.E = E
    self.tau = tau
    self.delay = delay

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

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

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

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


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

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

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

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

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

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

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

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


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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

syn_A2non_AMPA.g_max = g_max_E2E_AMPA
syn_A2non_NMDA.g_max = g_max_E2E_NMDA

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

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

syn_B2non_AMPA.g_max = g_max_E2E_AMPA
syn_B2non_NMDA.g_max = g_max_E2E_NMDA

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

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

syn_non2non_AMPA.g_max = g_max_E2E_AMPA
syn_non2non_NMDA.g_max = g_max_E2E_NMDA

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

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

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

```
                    delay=delay_syn)

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

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

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

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

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

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

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


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

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

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

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

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

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

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

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

for i in [syn_back2A_AMPA, syn_back2B_AMPA, syn_back2non_AM
  i.g_max = g_max_ext2E_AMPA
  i.E = E_AMPA
  i.tau_decay = tau_decay_AMPA

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

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

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

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

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

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

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


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

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

syn_input2A_AMPA = AMPA(pre=neu_input2A, post=neu_A,
                        conn=bp.connect.One2One())

syn_input2B_AMPA = AMPA(pre=neu_input2B, post=neu_B,
                        conn=bp.connect.One2One())

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

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

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

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

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


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


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

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

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

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

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

plt.show()
```

Firing rate networks

Decision model

```
from collections import OrderedDict
import brainpy as bp

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


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

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

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

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

    return ds1dt, ds2dt

  def __init__(self, size, coh, JAext=.00117, J_rec=.3725,
               I_0=.3297, a=270., b=108., d=0.154, tau_s=.0
               **kwargs):
    # parameters
    self.coh = coh
    self.JAext = JAext
    self.J_rec = J_rec
    self.J_inh = J_inh
    self.I0 = I_0
    self.a = a
    self.b = b
    self.d = d
    self.tau_s = tau_s
    self.gamma = gamma

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

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

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

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

```
def phase_analyze(I, coh):
  decision = Decision(1, coh=coh)

  phase = bp.analysis.PhasePlane(decision.integral,
                                 target_vars=OrderedDict(s2
                                                         s1
                                 fixed_vars=None,
                                 pars_update=dict(I=I, coh=
                                                  JAext=.00
                                                  J_inh=.11
                                                  a=270., b
                                                  tau_s=.06
                                 numerical_resolution=.001,
                                 options={'escape_sympy_sol

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

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

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

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

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

CANN

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

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

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

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

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

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

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

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

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

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

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

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

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


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

                          frame_step=frame_step, frame_dela
                          show=True)
```

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

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

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


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

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

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

plot_animate()
```

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

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

plot_animate()
```