BrainPy

Release 0.3.1

Chaoming Wang

Jan 05, 2021

TUTORIALS

`BrainPy` is a lightweight framework based on the latest Just-In-Time (JIT) compilers. The goal of `BrainPy` is to provide a unified simulation and analysis framework for neuronal dynamics with the feature of high flexibility and efficiency.

Comprehensive examples of BrainPy please see BrainPy-Models.

Note: BrainPy is a project under development. More features are coming soon. Contributions are welcome. https://github.com/PKU-NIP-Lab/BrainPy

---

CHAPTER

ONE

INSTALLATION

`BrainPy` is designed to run on across-platforms, including Windows, GNU/Linux and OSX. It only relies on Python libraries.

1.1 Installation with Anaconda

You can install `BrainPy` from the `anaconda cloud`. To do so, use:

```
conda install -c brainpy brainpy
```

1.2 Installation with pip

If you decide not to use `conda`, you can install `BrainPy` from the GitHub, or OpenI.

To do so, use:

```
pip install git+https://github.com/PKU-NIP-Lab/BrainPy
```

Or

```
pip install git+https://git.openi.org.cn/OpenI/BrainPy
```

To install the specific version of `BrainPy`, your can use

```
pip install -e git://github.com/PKU-NIP-Lab/BrainPy.git@V0.2.5
```

CHAPTER

TWO

BRAINPY QUICKSTART

Two main functions are provided in BrainPy: **neurodynamics simulation** and **neurodynamics analysis**. In this part, I will focus on neuronal dynamics simulation, and tell you how to code a dynamical network in BrainPy bu using the example of *(Wang & Buzsáki, 1996)*.

- Wang, Xiao-Jing, and György Buzsáki. "Gamma oscillation by synaptic inhibition in a hippocampal interneuronal network model." Journal of neuroscience 16.20 (1996): 6402-6413.

(Wang & Buzsáki, 1996) demonstrates how a group of neuron with mutual inhibition produce the gamma oscillation (20–80 Hz) observed in the neocortex and hippocampus. In this network model, the neurons are modeled as a variant of Hodgkin–Huxley (HH) neuron model, and the inhibition connections between neurons are modeled as the GABAA synapses.

Here, we will first build a HH neuron model. Then, construct a GABAA synapse model. Finally, combining the HH model and GABAA moldel together, we will build a network model. We expect at the suitable parameter regions, the network will produce gamma oscillation.

First of all, import your favorite `brainpy` and `numpy` package.

[1]:
```
import brainpy as bp
import numpy as np
```

In BrainPy, all the system-level settings are implmented in `bp.profile`. By using `bp.profile`, you can set the backend or the device of the models going to run on, the method and the precision of the numerical integrator, etc. Before diving into this tutorial, let's set the necessary profiles:

[2]:
```
bp.profile.set(jit=True,
               device='cpu',
               dt=0.04,
               numerical_method='exponential')
```

This setting means we will JIT compile our model on `cpu` device, the default numerical method is set to exponential euler method (`exponential`), and the numerical step is set to `0.04`.

2.1 How to build a neuron model?

In BrainPy, the solving of differential equations is based on numerical methods, such as Euler method, Runge–Kutta methods. Therefore, the definition of a neuron/synapse model is the definition of the *step functions* which explicitly point out how variable state at the current time point $x(t)$ is transited to the next time point $x(t+1)$.

Let's take the neuron model as an example.

To build a neuron model in BrainPy is to create an instance of `NeuType`. The instantiation of `NeuType` requires three items:

- *ST*: The neuron model state.
- *steps*: The step function to update at each cycle of run.
- *name* : The name of the neuron model (will be useful in error reporting and debugging).

Here, we are going to create a HH neuron model. The parameters of HH model are defined in the follows:

[3]:
```
V_th = 0.  # the spike threshold
C = 1.0  # the membrane capacitance
gLeak = 0.1  # the conductance of leaky channel
ELeak = -65  # the reversal potential of the leaky channel
gNa = 35.  # the conductance of sodium channel
ENa = 55.  # the reversal potential of sodium
gK = 9.  # the conductance of potassium channel
EK = -90.  # the reversal potential of potassium
phi = 5.0  # the temperature depdendent scaling
```

In this variant of HH model, three dynamical variables (V, h and n) exist.

Coding the differential equations

For any ordinary differential equation

$$\frac{dx}{dt} = f(x, t)$$

you only need write down the right-hand part of the differential equations $f(x, t)$. By adding a powerfull decorator porovided by BrainPy, `@bp.integrate`, the framework will automatically numerically integrate the defined equations. Generally, an ordinary differential equation in BrainPy can be coded as:

[4]:
```
@bp.integrate
def func(x, t, other_arguments):
    # ... some computation ...
    dxdt = ...
    return dxdt
```

`@bp.integrate` receives `method` keyword to specify the numerical method you want to choose. For example, adding `@bp.integrate(method='rk4')` means you integrate the decorated function by using Fouth-order Runge–Kutta method (The full list of supportted numerical integrators please see the document of Numerical integrators). Otherwise, the differential function will be integrated by the system default method.

Specifically, for **h channel**, the differential equation is mathematically formed as:

$$\frac{dh}{dt} = \alpha_h(V)(1-h) - \beta_h(V)h \tag{2.1}$$

$$\alpha_h(V) = 0.07 \cdot \exp\left(-\frac{V+58}{20}\right) \tag{2.2}$$

$$\beta_h(V) = \frac{1}{1+\exp\left(-\frac{V+28}{10}\right)} \tag{2.3}$$

In BrainPy, you can code h channel equation like this:

[4]:
```
@bp.integrate
def int_h(h, t, V):
    alpha = 0.07 * np.exp(-(V + 58) / 20)
    beta = 1 / (np.exp(-0.1 * (V + 28)) + 1)
    dhdt = alpha * (1 - h) - beta * h
    return phi * dhdt
```

The differential equation of **n channel**

$$\frac{dn}{dt} = \alpha_n(V)(1-n) - \beta_n(V)n \tag{2.4}$$

$$\alpha_n(V) = \frac{0.01 \cdot (V+34)}{1-\exp\left(-\frac{V+34}{10}\right)} \tag{2.5}$$

$$\beta_n(V) = 0.125 \cdot \exp\left(-\frac{V+44}{80}\right) \tag{2.6}$$

can be coded as:

[5]:
```
@bp.integrate
def int_n(n, t, V):
    alpha = -0.01 * (V + 34) / (np.exp(-0.1 * (V + 34)) - 1)
    beta = 0.125 * np.exp(-(V + 44) / 80)
    dndt = alpha * (1 - n) - beta * n
    return phi * dndt
```

Finally, the differential equation of **membrane potential V** is expressed as:

$$C_m \frac{dV}{dt} = -\bar{g}_{\mathrm{K}} n^4 (V - V_K) - \bar{g}_{\mathrm{Na}} m^3 h (V - V_{Na}) - \bar{g}_l (V - V_l) + I_{syn} \tag{2.7}$$

where m is modeled as an instaneous channel (fast enough and substituted by its steady-state function)

$$\alpha_m(V) = \frac{0.1(V+40)}{1-\exp\left(-\frac{V+40}{10}\right)} \tag{2.8}$$

$$\beta_m(V) = 4.0 \cdot \exp\left(-\frac{V+65}{18}\right) \tag{2.9}$$

$$m = \frac{\alpha_m(V)}{\alpha_m(V) + \beta_m(V)} \tag{2.10}$$

Therefore, the differential equations of V is coded as:

[6]:
```
@bp.integrate
def int_V(V, t, h, n, Isyn):
    m_alpha = -0.1 * (V + 35) / (np.exp(-0.1 * (V + 35)) - 1)
    m_beta = 4 * np.exp(-(V + 60) / 18)
    m = m_alpha / (m_alpha + m_beta)
    INa = gNa * m ** 3 * h * (V - ENa)
    IK = gK * n ** 4 * (V - EK)
    IL = gLeak * (V - ELeak)
    dvdt = (- INa - IK - IL + Isyn) / C
    return dvdt
```

Neuron state

In order to support the convenient state management, BrainPy provides `NeuState` to help you manage your model state.

In HH neuron, there are V, h and n dynamical variables. Moreover, the *input* of the neuron is time-varying. We can add a `input` item in HH neuron state to receive the varying input. Further, the neuron *spike* is also we take care of. So, the neuron state of HH model can be specified as

[7]:
```
HH_ST = bp.types.NeuState({
    'V': -55.,  # membrane potential, default initial value is -55.
    'h': 0.,  # h channel, default initial value is 0.
    'n': 0.,  # n channel, default initial value is 0.
```

(continues on next page)

(continued from previous page)

```
    'spike': 0.,  # neuron spike state, default initial value is 0.,
                  # if neuron emits a spike, it will be 1.
    'input': 0.  # neuron synaptic input, default initial value is 0.
})
```

The instantiation of `bp.types.NeuState` can receive a dict (which means the fields and their default initial values), or a list/tuple of fields (in this case the default initial value will be set to 0.).

Step functions

For each model, the most important thing is to define the step functions. After the definition of differential equations, the step function of the HH model can be defined as:

[8]:
```
def update(ST, _t):
    h = int_h(ST['h'], _t, ST['V'])
    n = int_n(ST['n'], _t, ST['V'])
    V = int_V(ST['V'], _t, ST['h'], ST['n'], ST['input'])
    sp = np.logical_and(ST['V'] < V_th, V >= V_th)
    ST['spike'] = sp
    ST['V'] = V
    ST['h'] = h
    ST['n'] = n
    ST['input'] = 0.
```

To define a step function, you can pass any data you need into the function as the functional arguments. The order of the arguments in each step function can be arbitrary.

In HH model step function, as you can see, two arguments are required:

- `ST`: the neuron state.
- `_t`: the current time.

`_t` is a system keyword, which denotes the current time point. In BrainPy, there are three system keywords: `_t`, `_i` (the current running step number), and `_dt` (the numerical integration step).

Define a NeuType

Finally, putting the above together, we get our HH neuron model as:

[9]:
```
HH = bp.NeuType(ST=HH_ST, name='HH_neuron', steps=update)
```

More advanced usage of `NeuType` definition please see Build Neurons.

2.2 How to build a synapse model?

Like the `NeuType` definition, let's announce the parameters all we need in the following:

[10]:
```
g_max = 0.1  # the maximal synaptic conductance
E = -75.  # the reversal potential
alpha = 12.  # the channel opening rate
beta = 0.1  # the channel closing rate
```

The GABAA synapse defined in *(Wang & Buzsáki, 1996)* is mathematically expressed as

$$\frac{ds}{dt} = \alpha F(V_{pre})(1-s) - \beta s \quad (1) \tag{2.11}$$

$$F(V_{pre}) = \frac{1}{1+\exp\left(-\frac{V_{pre}-V_{th}}{2}\right)} \quad (2)$$

The synaptic current output onto the post-synaptic neuron is expressed as

$$I_{syn} = g_{max} s \, (V - E) \quad (3)$$

Obviously, GABAA synapse model has one dynamical variable `s`. Thus, we can create the synapse state by using

[11]:
```
ST = bp.types.SynState(['s'])
```

Based on the equation (1) and (2), the state updating of GABAA synapse is coded as:

[12]:
```
@bp.integrate
def int_s(s, t, TT):
    return alpha * TT * (1 - s) - beta * s

def update(ST, _t, pre):
    T = 1 / (1 + np.exp(-(pre['V'] - V_th) / 2))
    s = int_s(ST['s'], _t, T)
    ST['s'] = s
```

Moreover, based on the equation (3), the delayed synaptic value output onto the post-synaptic neurons of GABAA synpase can be coded as:

[13]:
```
@bp.delayed
def output(ST, post):
    post['input'] -= g_max * ST['s'] * (post['V'] - E)
```

The decorator `@bp.delayed` can be added on the function which need the delayed `ST`. BrainPy will automatically recognize the delayed fileds. For example, in this `output()` function, the field `s` will be automatically delayed. When calling `output()`, the `ST['s']` will be the delayed one.

Moreover, as you can see, the definition of the GABAA synapse requires the following data:

- `pre`: The pre-synaptic neuron state, in which the field `V` is needed to compute the synapatic state.
- `post`: The post-synaptic neuron state, in which the fields `V` and `input` are needed.

If you use this model defined by yourself, you clearly know what data you need to run the model. However, when somebody use your defined model, they will be confused by what `pre` and `post` mean. So, here we can make declarations (optional):

[14]:
```
requires = dict(
    pre=bp.types.NeuState(['V']),
    post=bp.types.NeuState(['V', 'input']),
)
```

Finally, let's put the above difinitions together, and we get our wantted synapse model:

[15]:
```
GABAa = bp.SynType(ST=ST,
                   name='GABAa',
                   steps=(update, output),
                   requires=requires,
                   mode='scalar')
```

2.3 How to construct a network?

It is worthy to note that the above defined `HH` NeuType and `GABAa` SynType are abstract models. They can not be used for concrete computation. Instead, we should define the `NeuGroup` and `SynConn`.

Here, by using `bp.NeuGroup`, let's define a neuron group which contains 100 neurons. At the same time, we monitor the history trajectory of membrane potential `V` and spikes `spike`.

[16]:
```
num = 100
neu = bp.NeuGroup(HH, geometry=num, monitors=['spike', 'V'])
```

Similarly, the concrete synaptic connection can be constructed by using `bp.SynConn`. It receives an instance of SynType (argument `model`), the pre-synaptic neuron group (argument `pre_group`), the post-synaptic neuron group (argument `post_group`), the connection methods between the two groups (argument `conn`), and the delay length (argument `delay`).

[18]:
```
syn = bp.SynConn(model=GABAa,
                 pre_group=neu,
                 post_group=neu,
                 conn=bp.connect.All2All(include_self=False),
                 delay=0.5,
                 monitors=['s'])
```

The initial state value of a neuron group or an ensemble of synaptical connections can be updated by set `neu_group.ST[key] = value`, or `syn_conn.ST[key] = value`. In this example, we can update the initial value of `neu` as:

[17]:
```
v_init = -70. + np.random.random(num) * 20
h_alpha = 0.07 * np.exp(-(v_init + 58) / 20)
h_beta = 1 / (np.exp(-0.1 * (v_init + 28)) + 1)
h_init = h_alpha / (h_alpha + h_beta)
n_alpha = -0.01 * (v_init + 34) / (np.exp(-0.1 * (v_init + 34)) - 1)
n_beta = 0.125 * np.exp(-(v_init + 44) / 80)
n_init = n_alpha / (n_alpha + n_beta)

neu.ST['V'] = v_init
neu.ST['h'] = h_init
neu.ST['n'] = n_init
```

Moreover, the parameters of the created neuron groups or synaptical connections can be updated by using `neu_group.pars[key] = value`, or `syn_conn.pars[key] = value`. In this example, we can update the parameter of `syn` as:

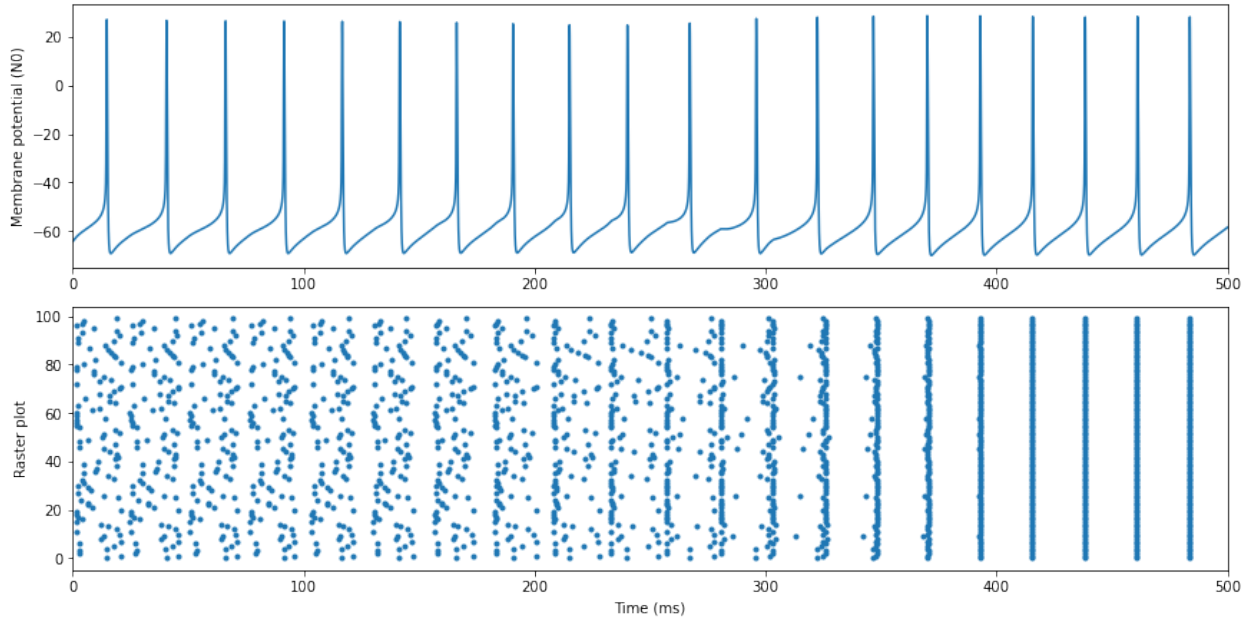
[19]:
```
syn.pars['g_max'] = 0.1 / num
```

Finally, by adding the created neuron groups and synapse connection into the `bp.Network`, we get an instance of the network. Each `bp.Network` has a powerful function `.run()`. You can specify the total duration to run (argument `duration`), the inputs to various components (argument `inputs`), and the option for progress reporting (arguments `report` and `report_percent`).

[20]:
```
net = bp.Network(neu, syn)
net.run(duration=500., inputs=[neu, 'ST.input', 1.2], report=False)
```

Let's visualize the network running results.

[21]:
```
import matplotlib.pyplot as plt

ts = net.ts
fig, gs = bp.visualize.get_figure(2, 1, 3, 12)

fig.add_subplot(gs[0, 0])
plt.plot(ts, neu.mon.V[:, 0])
plt.ylabel('Membrane potential (N0)')
plt.xlim(net.t_start - 0.1, net.t_end + 0.1)

fig.add_subplot(gs[1, 0])
index, time = bp.measure.raster_plot(neu.mon.spike, net.ts)
plt.plot(time, index, '.')
plt.xlim(net.t_start - 0.1, net.t_end + 0.1)
plt.xlabel('Time (ms)')
plt.ylabel('Raster plot')

plt.show()
```



The full file of this example model can be obtained in gamma_oscillation.

CHAPTER

THREE

BUILD NEURONS

Contents

In BrainPy, the *definition* and *usage* of the neuron model is separated from each other. In such a way, users can recycle the defined models to generate different neuron groups, or can use models defined by other people. Specifically, two class should be used:

- `brainpy.NeuType`: Define the abstract neuron model.
- `brainpy.NeuGroup`: Use the abstract neuron model to generate a concrete neuron group.

[4]:
```
import sys
sys.path.append('../../')

import brainpy as bp
import numpy as np

bp.profile.set(dt=0.01)
```

3.1 brainpy.NeuType

Three items should be specified to initialize a `NeuType`:

- `ST`: The neuronal state.
- `name`: The neuron model name.
- `steps`: The step functions to update at each time step.
- `requires`: The data requires to run the defined model (optional).

Two kinds of definition provided in BrainPy to define a `NeuType`:

- `scalar-based`: Each item in `ST` is a scalar, which represents the state of a single neuron.
- `vector-based`: Each item in `ST` is a vector, which represents the state of a group of neurons.

The definition logic of `scalar-based` models may be more straightforward than `vector-based` models. We will see this in the example of LIF model.

3.1.1 Hodgkin-Huxley model

Let's first take the Hodgkin-Huxley (HH) neuron model as an example to see how to define a `NeuType` in BrainPy.

[2]:
```
# parameters we need #
# ------------------ #

C = 1.0  # Membrane capacity per unit area (assumed constant).
g_Na = 120.  # Voltage-controlled conductance per unit area
             # associated with the Sodium (Na) ion-channel.
E_Na = 50.   # The equilibrium potentials for the sodium ions.
E_K = -77.   # The equilibrium potentials for the potassium ions.
g_K = 36.  # Voltage-controlled conductance per unit area
           # associated with the Potassium (K) ion-channel.
E_Leak = -54.402  # The equilibrium potentials for the potassium ions.
g_Leak = 0.003 # Conductance per unit area associated with the leak channels.
Vth = 20.  # membrane potential threshold for spike
```

Four differential equations exist in HH neuron model. Please check Differential equations to see how BrainPy supports differential equations.

For m channel, the difinition of the corresponding equations can be:

$$\frac{dm}{dt} = \alpha_m(V)(1-m) - \beta_m(V)m \tag{3.1}$$

$$\alpha_m(V) = \frac{0.1(V+40)}{1-\exp\left(-\frac{V+40}{10}\right)} \tag{3.2}$$

$$\beta_m(V) = 4.0 \cdot \exp\left(-\frac{V+65}{18}\right) \tag{3.3}$$

[3]:
```
@bp.integrate
def int_m(m, t, V):
    alpha = 0.1 * (V + 40) / (1 - np.exp(-(V + 40) / 10))
    beta = 4.0 * np.exp(-(V + 65) / 18)
    dmdt = alpha * (1 - m) - beta * m
    return dmdt
```

The h channel is defined as:

$$\frac{dm}{dt} = \alpha_m(V)(1-m) - \beta_m(V)m \tag{3.4}$$

$$\alpha_m(V) = \frac{0.1(V+40)}{1-\exp\left(-\frac{V+40}{10}\right)} \tag{3.5}$$

$$\beta_m(V) = 4.0 \cdot \exp\left(-\frac{V+65}{18}\right) \tag{3.6}$$

[4]:
```
@bp.integrate
def int_h(h, t, V):
    alpha = 0.07 * np.exp(-(V + 65) / 20.)
    beta = 1 / (1 + np.exp(-(V + 35) / 10))
    dhdt = alpha * (1 - h) - beta * h
    return dhdt
```

The n channel is defined as:

$$\frac{dn}{dt} = \alpha_n(V)(1-n) - \beta_n(V)n \tag{3.7}$$

$$\alpha_n(V) = \frac{0.1 \cdot (V+55)}{1 - \exp\left(-\frac{V+55}{10}\right)} \tag{3.8}$$

$$\beta_n(V) = 0.125 \cdot \exp\left(-\frac{V+65}{80}\right) \tag{3.9}$$

[5]:
```
@bp.integrate
def int_n(n, t, V):
    alpha = 0.01 * (V + 55) / (1 - np.exp(-(V + 55) / 10))
    beta = 0.125 * np.exp(-(V + 65) / 80)
    dndt = alpha * (1 - n) - beta * n
    return dndt
```

The membrane potential V is defined as:

$$C_m \frac{dV}{dt} = -\bar{g}_{\mathrm{K}} n^4 (V - V_K) - \bar{g}_{\mathrm{Na}} m^3 h (V - V_{Na}) - \bar{g}_l (V - V_l) + I_{syn} \tag{3.10}$$

[6]:
```
@bp.integrate
def int_V(V, t, m, h, n, Isyn):
    INa = g_Na * m ** 3 * h * (V - E_Na)
    IK = g_K * n ** 4 * (V - E_K)
    IL = g_Leak * (V - E_Leak)
    dvdt = (- INa - IK - IL + Isyn) / C
    return dvdt
```

In BrainPy, most of the integration of differential equations are implemented by the numerical methods, such as Euler, Exponential Euler, RK2, RK4 (please see Numerical integrators). Therefore, after defining the differential equations, the next important thing is to define the update logic for each variable from the current time point to next.

Here, let's first define the state of a HH model. We provide a data structure `brainpy.types.NeuState` to support the neuron state management.

[7]:
```
ST = bp.types.NeuState(
     'm',  # denotes potassium channel activation probability.
     'h',  # denotes sodium channel activation probability.
     'n',  # denotes sodium channel inactivation probability.
     'spike',  # denotes spiking state.
     'input',  # denotes synaptic input.
     V=-65.,  # denotes membrane potential.
)
```

In `ST`, the dynamical variable V, m, h, and n are inluded (without the value specification, the default value of m and n will be 0.). We also take care about whether the neuron provide a $spike$ at current time. Moreover, we define a $input$ item to receive the synaptic inputs and the external inputs.

Based on the neuron state `ST`, the update logic of the HH model from the current time point (t) to the next time point $(t + dt)$ can be defined as:

[8]:
```
def update(ST, _t):
    m = np.clip(int_m(ST['m'], _t, ST['V']), 0., 1.)
    h = np.clip(int_h(ST['h'], _t, ST['V']), 0., 1.)
```

(continues on next page)

(continued from previous page)

```
    n = np.clip(int_n(ST['n'], _t, ST['V']), 0., 1.)
    V = int_V(ST['V'], _t, ST['m'], ST['h'], ST['n'], ST['input'])

    ST['spike'] = np.logical_and(ST['V'] < Vth, V >= Vth)
    ST['V'] = V
    ST['m'] = m
    ST['h'] = h
    ST['n'] = n
    ST['input'] = 0.
```

In this example, the `update()` function of HH model needs two data:

- `ST`: The neuron state.
- `_t`: The system time at current point.

Putting together, a HH neuron model is defined as:

[9]:
```
HH = bp.NeuType(name='HH_neuron',
                ST=ST,
                steps=update,
                mode='vector')
```

Here, we should note that we just define an abstract HH neuron model. This model can run with any number of neurons, and with any geometry (one dimension, or two dimension). Only after define a concrete *neuron group*, can we run it or use it to construct a network.

3.1.2 LIF model (vector-based)

Here, same with *HH model* defined above, let's define a vector-based LIF model. The formal equations of a LIF model is given by:

$$\tau_m \frac{dV}{dt} = -(V(t) - V_{rest}) + I(t)$$
$$\text{after } V(t) V_{th}, V(t) = V_{rest} \text{ last } \tau_{ref} \text{ ms}$$

where V is the membrane potential, V_{rest} is the rest membrane potential, V_{th} is the spike threshold, τ_m is the time constant, τ_{ref} is the refractory time period, and I is the time-variant synaptic inputs.

Let's define the following item in neuron state:

- `V`: The membrane potential.
- `input`: The synaptic input.
- `spike`: Whether produce a spike.
- `refractory`: Whether the neuron is in refractory state.
- `t_last_spike`: The last spike time for calculating refractory state.

[10]:
```
ST = bp.types.NeuState(
        'V',     # membrane potential
         'input',  # synaptic input
         'spike',  # spike state
         'refractory',  # refractory state
         t_last_spike=-1e7  # last spike time
    )
```

Assume the items in the neuron state `ST` of a LIF model are vectors, the update logic of vector-based LIF neuron model is:

[11]:
```
tau_m=10.; Vr=0.; Vth=10.; tau_ref=0.

@bp.integrate
def int_f(V, t, Isyn):
    return (-V + Vr + Isyn) / tau_m

def update(ST, _t):
    V = int_f(ST['V'], _t, ST['input'])
    is_ref = _t - ST['t_last_spike'] < tau_ref
    V = np.where(is_ref, ST['V'], V)
    is_spike = V > Vth
    spike_idx = np.where(is_spike)[0]
    if len(spike_idx):
        V[spike_idx] = Vr
        is_ref[spike_idx] = 1.
        ST['t_last_spike'][spike_idx] = _t
    ST['V'] = V
    ST['spike'] = is_spike
    ST['refractory'] = is_ref
    ST['input'] = 0.

lif = bp.NeuType(name='LIF',
                 ST=ST,
                 steps=update,
                 mode='vector')
```

Here, for vector-based LIF model, we must differentiate the states for each neuron at every time point. For neurons in refractory period (`is_ref`), we must keep its V unchange. For neurons in spiking state (`is_spike`), we must reset its membrane potential. So, it looks like the definition of vector-based LIF mode is somewhat complex. However, the good news is that BrainPy support the difinition of neuron models in scalar mode, which means at each time point, your model difinition can only consider the behavior of one single neuron. Let's take a look.

3.1.3 LIF model (scalar-based)

[12]:
```
def update(ST, _t):
    if _t - ST['t_last_spike'] > tau_ref:
        V = int_f(ST['V'], _t, ST['input'])
        if V >= Vth:
            V = Vr
            ST['t_last_spike'] = _t
            ST['spike'] = True
        ST['V'] = V
    else:
        ST['spike'] = False
    ST['input'] = 0.

lif = bp.NeuType(name='LIF',
                 ST=ST,
                 steps=update,
                 mode='scalar')
```

As you can see, the scalar-based LIF model is intuitive and straightforward in BrainPy. If the neuron is not in refractory period (`_t - ST['t_last_spike'] > tau_ref`), integrate the membrane potential by calling `int_f()`. If

the neuron reaches the spike threshold (`V >= Vth`), then reset the membrane potential (`V = Vr`) and set the spike state to bs `True`.

However, it's worthy to note that the scalar-based and the vector-based model have different flexibility ratio. The scalar-based model is convinient and easy to define, but is not flexible enough like the vector-based model. For the vector-based model, you can control everything, and define any data you want. For example, you can operate on the whole group level to count the total spikes by defining a variable `total_spike`, or get the instantaneous population firing rate, etc. Later, we will come back to this topic.

3.2 brainpy.NeuGroup

After we talk about `brainpy.NeuType`, the uasge of `brainpy.NeuGroup` is a piece of cake. This is because in a real project the most efforts we pay is the difinition of the models, and BrainPy provide a very convenient way to use your defined models. Specifically, a `brainpy.NeuGroup` receives the following specifications:

- `model`: The neuron type will be used to generate a neuron group.
- `geometry`: The geometry of the neuron group. Can be a int, or a tuple/list of int.
- `monitors`: The items to monitor (record the history values.)
- `name`: The neuron group name.

Let's take our defined HH model as an example.

[14]:
```
group = bp.NeuGroup(HH, geometry=10, monitors=['V', 'm', 'n', 'h'])
```

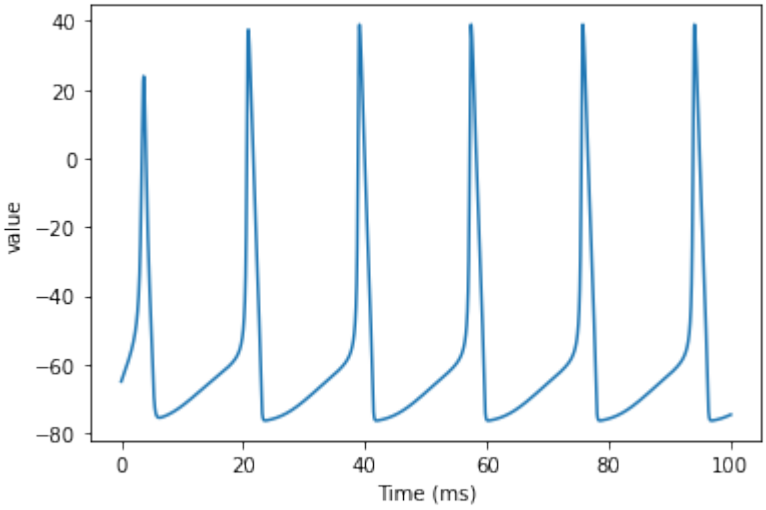
Each group has a powerful function: `.run()`. In this function, it receives the following arguments:

- `duration`: Specify the simulation duration. Can be a tuple with `(start time, end time)`. Or it can be a int to specify the duration `length` (then the default start time is `0`).
- `inputs`: Specify the inputs for each model component. With the format of `(target, value, [operation])`. The default operation is `+`, which means the input `value` will be added to the `target`. Or, the operation can be $-$, $*$, $/$, or $=$.

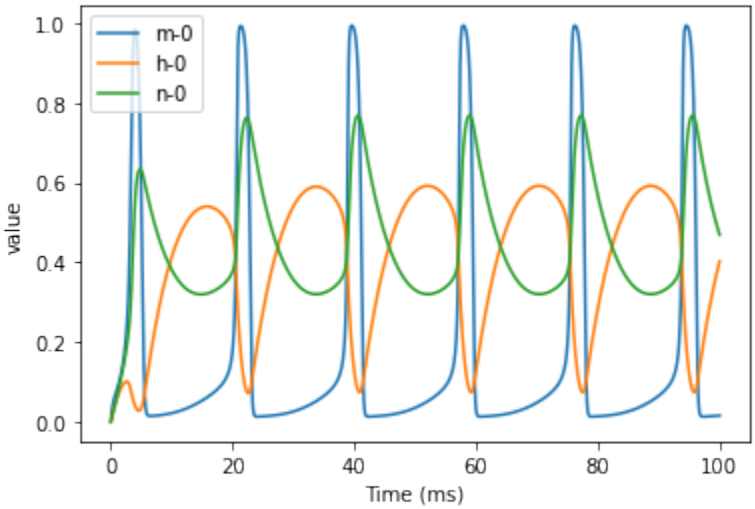
[15]:
```
group.run(100., inputs=('ST.input', 5.), report=True)
```

```
Compilation used 0.0000 s.
Start running ...
Run 10.0% used 0.104 s.
Run 20.0% used 0.208 s.
Run 30.0% used 0.317 s.
Run 40.0% used 0.420 s.
Run 50.0% used 0.522 s.
Run 60.0% used 0.632 s.
Run 70.0% used 0.738 s.
Run 80.0% used 0.844 s.
Run 90.0% used 0.952 s.
Run 100.0% used 1.059 s.
Simulation is done in 1.059 s.
```

[16]:
```
bp.visualize.line_plot(group.mon.ts, group.mon.V, show=True)
```



[17]:
```
bp.visualize.line_plot(group.mon.ts, group.mon.m, legend='m')
bp.visualize.line_plot(group.mon.ts, group.mon.h, legend='h')
bp.visualize.line_plot(group.mon.ts, group.mon.n, legend='n', show=True)
```



3.3 Reconcile the scalar- and vector-based model

The scalar-bsed and the vector-based model have different performance under different settings. When the users run the model without JIT acceleration (or `profile.set(jit=False)`), the vector-based model is much more efficient than the scalar-based model due to the powerful array-oriented programming support in NumPy. On the contrary, the scalar-based *LIF* model sometimes is efficient than the vector-based ones in `JIT` mode (However, for the neuron model without too many `if .. else ...` conditions, the vector-based nueron is much more efficient, for example, the HH model). This is thanks to the JIT acceleration of for loop provided in Numba. Therefore, we recommend you to choose different difinition mode for different running backend. For example, we can define a unified LIF model with the explicit backend judgement:

[13]:
```
def get_lif(tau_m=10., Vr=0., Vth=10., tau_ref=0.):
    ST = bp.types.NeuState({'V': 0, 'input':0, 'spike':0,
                            'refractory': 0, 't_last_spike': -1e7})

    @bp.integrate
    def int_f(V, t, Isyn):
        return (-V + Vr + Isyn) / tau_m

    if bp.profile.is_jit():

        def update(ST, _t):
            if _t - ST['t_last_spike'] > tau_ref:
                V = int_f(ST['V'], _t, ST['input'])
                if V >= Vth:
                    V = Vr
                    ST['t_last_spike'] = _t
                    ST['spike'] = True
                ST['V'] = V
            else:
                ST['spike'] = False
            ST['input'] = 0.

        lif = bp.NeuType(name='LIF', ST=ST, steps=update, mode='scalar')


    else:

        def update(ST, _t):
            V = int_f(ST['V'], _t, ST['input'])
            is_ref = _t - ST['t_last_spike'] < tau_ref
            V = np.where(is_ref, ST['V'], V)
            is_spike = V > Vth
            spike_idx = np.where(is_spike)[0]
            if len(spike_idx):
                V[spike_idx] = Vr
                is_ref[spike_idx] = 1.
                ST['t_last_spike'][spike_idx] = _t
            ST['V'] = V
            ST['spike'] = is_spike
            ST['refractory'] = is_ref
            ST['input'] = 0.

        lif = bp.NeuType(name='LIF', ST=ST, steps=update, mode='vector')

    return lif
```

3.4 NeuType `requires` and NeuGroup `satisfies`

The design of BrainPy is constrained by the two goals:

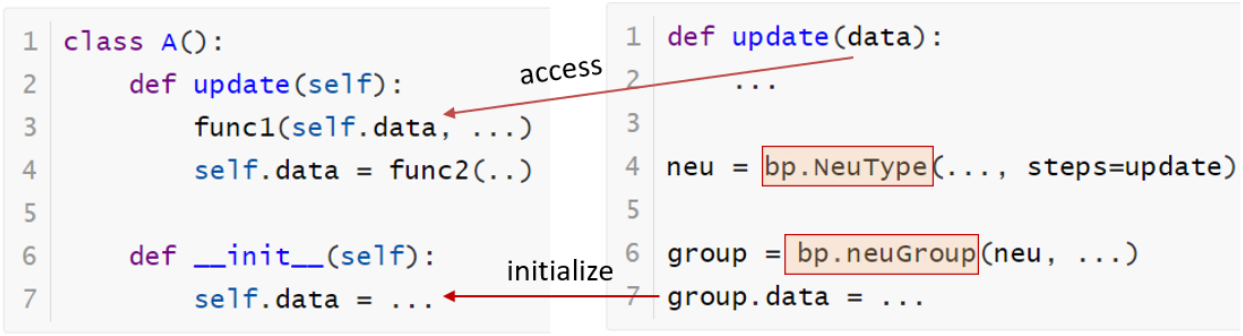
- JIT support.
- The model definition and usage separation.

BrainPy heavily relies on Numba. Unfortunately, Numba has poor support for `Python class`. And the performance of class computation is greatly reduced. In order to get the best JIT performace, BrainPy only allow users to define

models by using the `Python function`.

On the other hand, in order to recycle the defined models and hold model reproducibility, BranPy provides `NeuType` for model definition and `NeuGroup` for model usage.

Therefore, one core feature of BrainPy is to bring the computation of a class onto the function. In a class, any data you want can be accessed by `self.xxx`. In the function, such data `xxx` you **require** can be defined as an argument in `NeuType` step function. When using `NeuGroup`, user must initialize the data `xxx` to **satisfy** the step function requires.



Let's take the following illustrating model as an example:

[6]:
```
def update(ST, _t, data1, data2):
    ...

neu = bp.NeuType('test', ST=bp.NeuState(), steps=update,
                 requires={'data1': bp.types.Array(dim=1),
                           'data2': bp.types.Array(dim=2)})
```

The neuron type `neu` *requires* `data1` (a one-dimensional array) and `data2` (a two-dimensional array) to compute its update function.

[7]:
```
group = bp.NeuGroup(neu, geometry=1,
                    satisfies={'data1': np.zeros(10),
                               'data2':  np.ones((10, 10))})
```

When someone uses this defined model, he/she must provide the corresponding data (`data1` and `data2`) in the `NeuGroup` to *satisfy* the `NeuType` requirements.

However, one can also provide the required data in the following ways:

[8]:
```
group = bp.NeuGroup(neu, geometry=1)
group.data1 = np.zeros(10)
group.data2 = np.ones((10, 10))
```

3.5 NeuType `hand_overs` data/func to NeuGroup

Another useful keyword provided in `NueType` is `hand_overs`, which means you can hand over the data or the functions defined in the `NeuType` to the `NeuGroup` when someone want to use. For example, you can define a `init_state()` function to initialize the HH neuron model state:

[12]:
```
ST = bp.types.NeuState('V', 'm', 'h', 'n', 'spike', 'input')

@bp.integrate
```

(continues on next page)

(continued from previous page)

```
def int_m(m, _t, V):
    alpha = 0.1 * (V + 40) / (1 - np.exp(-(V + 40) / 10))
    beta = 4.0 * np.exp(-(V + 65) / 18)
    return alpha * (1 - m) - beta * m

@bp.integrate
def int_h(h, _t, V):
    alpha = 0.07 * np.exp(-(V + 65) / 20.)
    beta = 1 / (1 + np.exp(-(V + 35) / 10))
    return alpha * (1 - h) - beta * h

@bp.integrate
def int_n(n, _t, V):
    alpha = 0.01 * (V + 55) / (1 - np.exp(-(V + 55) / 10))
    beta = 0.125 * np.exp(-(V + 65) / 80)
    return alpha * (1 - n) - beta * n

@bp.integrate
def int_V(V, _t, m, h, n, I_ext):
    I_Na = (g_Na * np.power(m, 3.0) * h) * (V - E_Na)
    I_K = (g_K * np.power(n, 4.0))* (V - E_K)
    I_leak = g_leak * (V - E_leak)
    dVdt = (- I_Na - I_K - I_leak + I_ext)/C
    return dVdt, noise / C

# update the variables change over time (for each step)
def update(ST, _t):
    m = np.clip(int_m(ST['m'], _t, ST['V']), 0., 1.)
    h = np.clip(int_h(ST['h'], _t, ST['V']), 0., 1.)
    n = np.clip(int_n(ST['n'], _t, ST['V']), 0., 1.)
    V = int_V(ST['V'], _t, m, h, n, ST['input'])
    spike = np.logical_and(ST['V'] < V_th, V >= V_th)
    ST['spike'] = spike
    ST['V'] = V
    ST['m'] = m
    ST['h'] = h
    ST['n'] = n
    ST['input'] = 0.

def init_state(ST, Vr):
    ST['V'] = Vr
    V = ST['V']

    alpha = 0.1 * (V + 40) / (1 - np.exp(-(V + 40) / 10))
    beta = 4.0 * np.exp(-(V + 65) / 18)
    ST['m'] = alpha / (alpha + beta)

    alpha = 0.07 * np.exp(-(V + 65) / 20.)
    beta = 1 / (1 + np.exp(-(V + 35) / 10))
    ST['h'] = alpha / (alpha + beta)

    alpha = 0.01 * (V + 55) / (1 - np.exp(-(V + 55) / 10))
    beta = 0.125 * np.exp(-(V + 65) / 80)
    ST['n'] = alpha / (alpha + beta)

HH_neu = bp.NeuType(name='HH_neuron',
                    ST=ST,
```

(continues on next page)

(continued from previous page)

```
                    steps=update,
                    mode='vector',
                    hand_overs={'init_state': init_state})
```

Once you use this model, you can easily initialize the model state by using:

[15]:
```
HH_group = bp.NeuGroup(HH_neu, geometry=10)
HH_group.init_state(HH_group.ST, np.random.random(10) * 10 - 75.)
```

[16]:
```
HH_group.ST['V']
```

[16]:
```
array([-70.9077187 , -66.55979129, -65.14340083, -69.51226795,
       -73.38228859, -68.06085232, -72.08114256, -71.82573823,
       -71.86175207, -72.1604964 ])
```

[17]:
```
HH_group.ST['m']
```

[17]:
```
array([0.02582321, 0.04396935, 0.05204429, 0.03070063, 0.01891655,
       0.03667513, 0.02229492, 0.02302182, 0.02291798, 0.02207352])
```

3.6 The advantages of the vector-based model

The advantage of the vector-based model is that you can control all the things that will happend in a neuron group.

For example, you can define a variable `spike_num` to count the spike number in a neuron group.

[18]:
```
tau_m=10.; Vr=0.; Vth=10.; tau_ref=0.

ST = bp.types.NeuState('V', 'input', 'spike', t_last_spike=-1e7 )

@bp.integrate
def int_f(V, t, Isyn):
    return (-V + Vr + Isyn) / tau_m

def update(ST, _t, spike_num):
    V = int_f(ST['V'], _t, ST['input'])
    is_spike = V > Vth
    spike_idx = np.where(is_spike)[0]
    num_sp = len(spike_idx)
    spike_num[0] += num_sp
    if num_sp > 0:
        V[spike_idx] = Vr
        ST['t_last_spike'][spike_idx] = _t
    ST['V'] = V
    ST['spike'] = is_spike
    ST['input'] = 0.

lif2 = bp.NeuType(name='LIF',ST=ST, steps=update, mode='vector',
                  hand_overs={'spike_num': np.array([0])})
```

Here, we define `spike_num` in `update()` function, and initialize `spike_num` as `np.array([0])` in the `hand_overs`. In such a way, any instance of `NeuGroup` for such neuron type will automatically have the required data `spike_num`. Note here we use a array to get the spike number, this is because BrainPy do not support step function return (by using arrays, this problem can be easily solved).

[19]:
```
group = bp.NeuGroup(lif2, geometry=10, monitors=['spike'])
group.run(100., inputs=('input', np.random.random(10) * 5 + 10.))
```

[21]:
```
group.spike_num[0]
```

[21]:
```
50
```

[22]:
```
group.mon.spike.sum()
```

[22]:
```
50.0
```

CHAPTER

FOUR

BUILD SYNAPSES

When you have some neurons, you need to build synapses to connect them. This tutorial will show you how to use synapse models to connect neurons to networks.

As neuron models, the *definition* and *usage* of the synapse model are separated from each other. Specifically, two classes should be used:

- `brainpy.SynType`: Define the abstract synapse model.
- `brainpy.SynConn`: Use the abstract synapse model to generate a concrete synapse connection.

We will first take a look at the definition with `brainpy.SynType`, then in the second part, we will show the usage with `brainpy.SynConn`.

Before we start, let's import the BrainPy and Numpy packages.

[1]:
```
import brainpy as bp
import numpy as np
```

4.1 brainpy.SynType

You can define any abstract synapse type with `SynType`, which is very flexible.

As neuron models, four parameters should be specified to initialize a `SynType`:

- `name`: The synapse model name.
- `steps`: The step functions to update at each time step. You can define your own update logic functions.
- `requires`: The data required to run this synapse model, such as synaptic states and neuronal states of the connected neurons.
- `mode`: Whether define the model based on scalar, vector, or matrix.

We provide a data structure `brainpy.types.SynState` to support the synapse state management.

Three kinds of definition provided in BrainPy to define a `SynType`:

- `mode = 'scalar'`: Synapse state `ST` represents the state of a single synapse connection. And, each item in `ST` is a scalar.
- `mode = 'vector'`: Synapse state `ST` represents the state of a group of synapse connections. And each item in `ST` is a vector,
- `mode = 'matrix'`: Synapse state `ST` represents the state of a group of synapse connections. And each item in `ST` is a matrix with the shape of `(num_pre, num_post)`.

The definition logic of scalar-based models may be more straightforward than vector- and matrix- based models. We will first introduce the definition of a simple synapse model in scalar-based mode.

4.1.1 Example: AMPA synapse model (scalar mode)

Let's first take the AMPA synapse model as an example to see how to define a `SynType` in BrainPy.

The formal equations of an AMPA synapse is given by:

$$I_{syn} = \bar{g}_{syn} s (V - E_{syn})$$

$$\frac{ds}{dt} = -\frac{s}{\tau_{decay}} + \sum_k \delta(t - t_j^k)$$

where $\bar{g}_{syn}$ is the maximum synaptic conductance, s is the gating variable, and V is the membrane potential of the postsynaptic neuron. The time constant τ_{decay} is about 2ms and the equilibrium potential E_{syn} for AMPA synapse is 0.

[2]:
```
# parameters we need #
# ------------------ #

tau_decay = 2.   # time constant of the dacay after synapse respond to a␣
↪neurontransmitter.
g_max = .10      # Voltage-controlled conductance per unit area
                 # associated with the Sodium (Na) and Potassium (K) ion-channels on␣
↪the synapse (postsynaptic membrane).
E = 0.           # The equilibrium potentials for the synapse.
```

Please check Differential equations to see how BrainPy supports differential equations.

[3]:
```
# dynamics of gating variable
@bp.integrate
def ints(s, t):
    return - s / tau_decay
```

Here, let's first define the state of a synapse model.

[4]:
```
ST=bp.types.SynState(['s'], help='AMPA synapse state.')
```

In `ST`, the dynamical variable s is included.

Since a synapse connects a presynaptic neuron and a postsynaptic neuron, we need to know the state of the two neurons.

[5]:
```
pre=bp.types.NeuState(['spike'], help='Presynaptic neuron state must have "sp" item.')
post=bp.types.NeuState(['V', 'input'], help='Postsynaptic neuron state must have "V"␣
↪and "inp" item.')
```

From the equations of the AMPA synapse, we need to know whether the presynaptic neuron (`pre`) provides a $spike$ at current time. We also need to know the current membrane potential V of the postsynaptic neuron, and add synaptic current to the $input$ item of the `post`.

Based on the synapse state `ST` and neuron states, the update logic of the synapse model from the current time point (t) to the next time point ($t + dt$) can be defined as:

[6]:
```
def update(ST, _t, pre):
    s = ints(ST['s'], _t)
    if pre['spike'] == True:
        s += 1
    ST['s'] = s
```

In this example, the `update()` function of AMPA model needs three data:

- `ST`: The synapse state.
- `_t`: The system time at current point.
- `pre`: The neuron state of the presynaptic neuron.

We also need to define an output logic to compute the synaptic current and add it to the postsynaptic inputs.

The synaptic delay between a presynaptic spike and the postsynaptic change can be implemented with a `@bp.delayed` decorator.

[7]:
```
@bp.delayed
def output(ST, post):
    I_syn = - g_max * ST['s'] * (post['V'] - E)
    post['input'] += I_syn
```

Putting together, an AMPA synapse model is defined as:

[8]:
```
AMPA = bp.SynType(name='AMPA_synapse',
                  ST=ST,
                  requires=dict(pre=pre, post=post),
                  steps=(update, output),
                  mode='scalar')
```

Here, we should note that we just define an abstract AMPA synapse model. This model can run with any number of synapse connections. Only after defining a concrete *synapse connection*, can we use it to construct a network.

4.1.2 Example: AMPA synapse model (matrix mode)

In matrix mode, each item in the synapse state `ST` is a matrix.

The differential equation part is the same as the scalar mode, and we also need a `SynState` and the `NeuState` of presynaptic and postsynaptic neurons.

[9]:
```
tau_decay = 2.
g_max = .10
E = 0.

@bp.integrate
def ints(s, t):
    return - s / tau_decay

ST=bp.types.SynState(['s', 'g'], help='AMPA synapse state.')
pre=bp.types.NeuState(['spike'], help='Presynaptic neuron state must have "sp" item.')
post=bp.types.NeuState(['V', 'input'], help='Presynaptic neuron state must have "V"␣
↪and "inp" item.')
```

We also need to define a connectivity matrix to specify the connectivity patterns between the presynaptic neurons and postsynaptic neurons, which can be defined with `brainpy.types.MatConn()`.

[10]:
```
conn_mat=bp.types.MatConn()
```

Here is an example of the connectivity matrix:

		post id								
		0	1	2	3	4	5	6	7	···
pre id	0	0	0	0	1	0	1	0	1	···
	1	1	0	1	0	1	0	1	0	···
	2	0	1	0	0	0	0	0	0	···
	···	···	···	···	···	···	···	···	···	···

The update and output are also similar to the scalar mode, but notice that the `pre` and `post` here are vectors, so all the operations are vectors.

[11]:
```
def update(ST, _t, pre, conn_mat):
    s = ints(ST['s'], _t)
    s += pre['spike'].reshape((-1, 1)) * conn_mat
    ST['s'] = s
    ST['g'] = g_max * s

@bp.delayed
def output(ST, post):
    g = np.sum(ST['g'], axis=0)
    post['input'] -= g * (post['V'] - E)

AMPA = bp.SynType(name='AMPA_synapse',
                  ST=ST,
                  requires=dict(pre=pre, post=post, conn_mat=conn_mat),
                  steps=(update, output),
                  mode='matrix')
```

4.1.3 Vector mode

In vector mode, each item in the synapse state `ST` is a vector.

Let's look at the synaptic connections in vector form.

Synaptic connectivity

Suppose we have two vectors of neurons and a vector of synapses connecting the neurons within the two neuron vectors. Many different connectivities are possible, and we use $index$ to recognize different synapses.

Each synapse receives information from one presynaptic neuron, and, commonly, different synapses get inconsistent signals. Therefore, it is helpful to specify a map from the presynaptic neuron vector to the synapses vector.

For example, we have a connectivity as below:

pre id \ post id	0	1	2	3	4	5	6	7	...
0	0	0	0	1	0	1	0	1	...
1	1	0	1	0	1	0	1	0	...
2	0	1	0	0	0	0	0	0	...
...	...	...	...	...	...	...	...	...	...

Where 1 and 0 indicate the presence and absence of synaptic connections, respectively. We can then arrange the synapses in the following manner:

pre ids	0	0	0	1	1	1	1	2	...
post ids	3	5	7	0	2	4	6	1	...
syn ids	0	1	2	3	4	5	6	7	...

We can create a `pre2syn` list, the indexes of this list correspond to the indexes of the presynaptic neurons vector, and the elements indicate the indexes of synapses that having connections to the neuron. Here, the first neuron connects the 3rd, 5th, and 7th neurons with synapses 0, 1, 2, so we store [0, 1, 2] as the first element of the `pre2syn` list. Thus, if the first neuron fire, then we can get the indexes of synapses by `syn_ids = pre2syn[0]` and changes the states of those synapses.

pre2syn

pre id	syn ids
0	[0, 1, 2]
1	[3, 4, 5, 6]
2	[7]
...	...

Similarly, we can use a `post2syn` list to indicate the connections between synapses and postsynaptic neurons. The indexes of this list correspond to the indexes of the presynaptic neurons vector, and the elements indicate the indexes of synapses that having connections to the neuron.

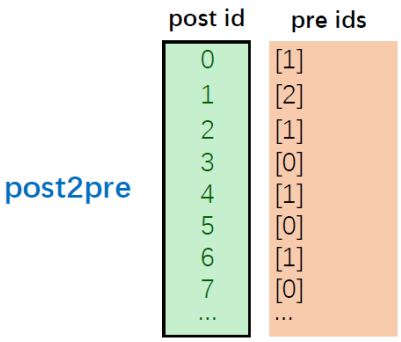
post2syn

post id	syn ids
0	[3]
1	[7]
2	[4]
3	[0]
4	[5]
5	[1]
6	[6]
7	[2]
...	...

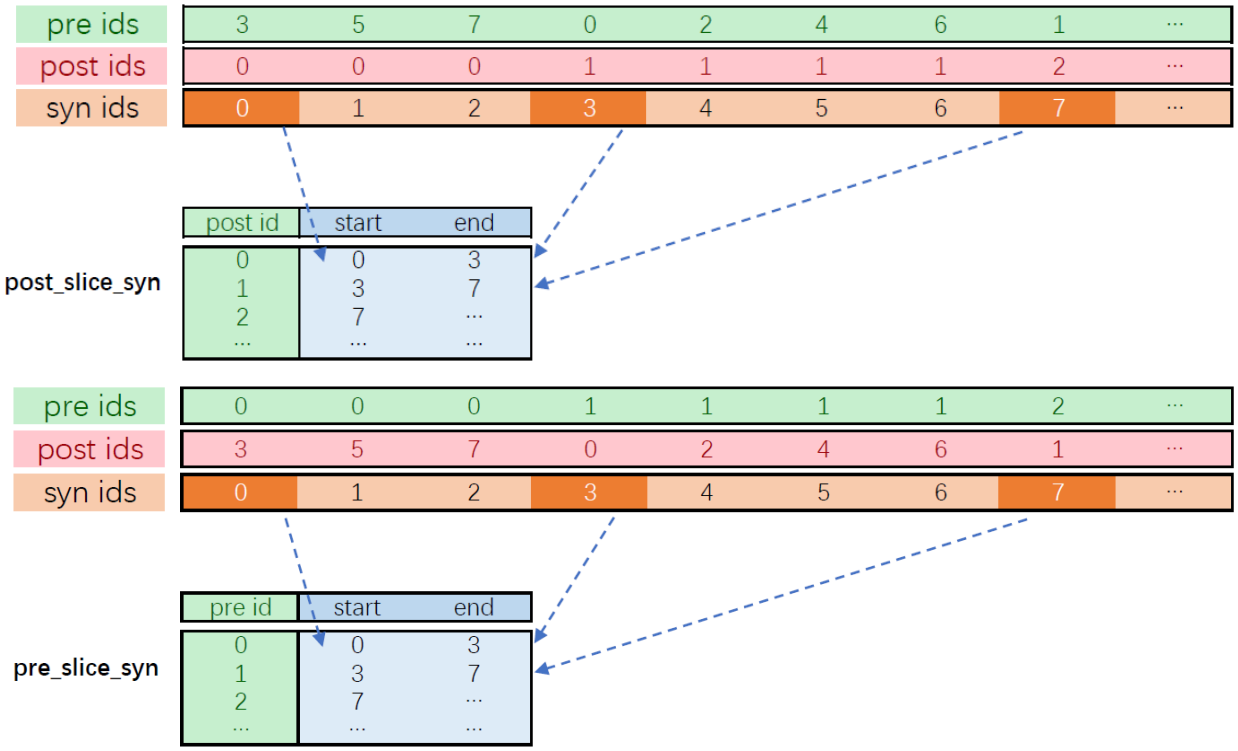
We can also create a map between two neurons vectors using a `pre2post` list and a `post2pre` list.

pre2post

pre id	post ids
0	[3, 5, 7]
1	[0, 2, 4, 6]
2	[1]
...	...



Other mapping ways are also possible.



4.1.4 Example: AMPA synapse model (vector mode)

Now let's see how to implement a vector-based synapses model by taking AMPA model as example. The formal equations of an AMPA synapse is the same as the scalar-based one:

$$I_{syn} = \bar{g}_{syn} s (V - E_{syn})$$

$$\frac{ds}{dt} = -\frac{s}{\tau_{decay}} + \sum_k \delta(t - t_j^k)$$

where $\bar{g}_{syn}$ is the maximum synaptic conductance, s is the gating variable, and V is the membrane potential of the postsynaptic neuron. The time constant τ_{decay} is about 2ms and the equilibrium potential E_{syn} for AMPA synapse is 0.

The differential equation part is the same as the scalar and matrix mode, and we also need a `SynState` and the `NeuState` of presynaptic and postsynaptic neurons.

[12]:
```
tau_decay = 2.
g_max = .10
E = 0.

@bp.integrate
def ints(s, t):
    return - s / tau_decay

ST=bp.types.SynState(['s'], help='AMPA synapse state.')
pre=bp.types.NeuState(['spike'], help='Presynaptic neuron state must have "sp" item.')
post=bp.types.NeuState(['V', 'input'], help='Presynaptic neuron state must have "V"␣
↪and "inp" item.')
```

For the mapping between synapse and neurons, BrainPy provides `brainpy.types.ListConn`.

[13]:
```
pre2syn = bp.types.ListConn()
post2syn = bp.types.ListConn()
```

Assume the items in the synapse state `ST` and neuron states `pre` and `post` are vectors, and we have the mapping lists `pre2syn` and `post2syn`, the update logic of vector-based AMPA synapse model is:

[14]:
```
def update(ST, _t, pre, pre2syn):
    s = ints(ST['s'], _t)

    spikeike_idx = np.where(pre['spike'] > 0.)[0]
    for i in spikeike_idx:
        syn_idx = pre2syn[i]
        s[syn_idx] += 1.

    # update values
    ST['s'] = s


@bp.delayed
def output(ST, post, post2syn):
    post_cond = np.zeros(len(post2syn), dtype=np.float_)
    for post_id, syn_ids in enumerate(post2syn):
        post_cond[post_id] = np.sum(g_max * ST['s'][syn_ids])
    post['input'] -= post_cond * (post['V'] - E)

AMPA_vector = bp.SynType(name='AMPA_synapse',
                         ST=ST,
                         requires=dict(pre=pre, post=post,
                                       pre2syn=pre2syn, post2syn=post2syn),
                         steps=(update, output),
                         mode='vector')
```

4.2 brainpy.SynConn

Synapse connections determine the architecture of a network. A `brainpy.SynConn` receives the following parameters:

- `model`: The synapse type will be used to generate a synapse connection.
- `pre_group`: The presynaptic neuron group.
- `post_group`: The postsynaptic neuron group.
- `conn`: The connection method to create synaptic connectivity between the neuron groups.
- `monitors`: The items to monitor (record the history values.)
- `delay`: The time of the synapse delay (in milliseconds).

BrainPy pre-defines several commonly used connection methods in `brainpy.connect`, read Usage of connect module for more details.

Let's take our defined AMPA model as an exmaple.

We can get pre-defined neuron models from the `bpmodels` package. Here we use the leaky intergrate-and-fire (LIF) model to create neuron groups

[15]:
```
from bpmodels.neurons import get_LIF

LIF = get_LIF(V_rest=-65., V_reset=-65., V_th=-55.)
pre = bp.NeuGroup(LIF, 1, monitors=['spike', 'V'])
pre.ST['V'] = -65.
post = bp.NeuGroup(LIF, 1, monitors=['V'])
post.ST['V'] = -65.
```

[16]:
```
syn = bp.SynConn(model=AMPA, pre_group=pre, post_group=post,
                 conn=bp.connect.All2All(),
                 monitors=['s'], delay=1.5)
```

You can specify the synapse behavior by using `syn.runner.set_schedule`.

[17]:
```
syn.runner.set_schedule(['input', 'update', 'output', 'monitor'])
```

Note that you cannot run the synapse connection (unlike neuron groups). You have to run them in a network.

[18]:
```
net = bp.Network(pre, syn, post)

net.run(duration=100., inputs=(pre, "ST.input", 20.))
```

CHAPTER

FIVE

BUILD NETWORK

To run a simulation with `BrainPy`, users will mainly use three classes: two fundamental classes `brainpy.NeuGroup` and `brainpy.SynConn`, which refers to groups of neurons and synapses, respectively; and class `brainpy.Network`, with which users can build a neural network and simulate the whole network as an ensemble.

In build neurons and build synapses, we have already introduced how to define `brainpy.NeuType` and `brainpy.SynType`. Base on former introductions, in this section, we will first introduce `brainpy.NeuGroup` and `brainpy.SynConn`, then transition to the contruction and simulation of neural networks with `brainpy.Network`.

[9]:
```
import brainpy as bp
import bpmodels
import numpy as np
import matplotlib.pyplot as plt
```

5.1 brainpy.NeuGroup

Users can implement a group of neurons as an object of `NeuGroup` class. Five input parameters are provided to the class contructor to describe the neuron group:

- `model`: The neuron models used to generate the neuron group.
- `geometry`: Geometry of the neuron group. Can be a int or a two-tuple of int.
- `pars_update`: Parameters to update. (??? is this clear?)
- `monitors`: List of ST members to monitor.
- `name`: Name of neuron group. Will be automatically assigned if not given.

Example code:

```
neu_group_1 = bp.NeuGroup(model=LIF, geometry=(10, ), monitors=['V'])
```

Parameter `geometry` represents the geometry of the neuron group users build. As `BrainPy` supports one-dimensional and two-dimensional neuron structure, `geometry` should be a int or a (int, int) tuple.

Parameter `monitors` includes the ST members that will be recorded during simulation. If a ST member is listed in `monitors`, its time series will be saved in variable `<NEU_GROUP_NAME>.mon.<MEM_NAME>` for later use.

Example code:

```
V_time_series = neu.mon.V[START_TIME:END_TIME, NEURON_NO]
```

`NeuGroup` supports index and slice operation, both for one-dimensional and two-dimensional geometry. Users can include all the neurons of the same model in one neuron group, then operate different subgroups of it.

Example code:

```
neu_group_2 = bp.NeuGroup(model = LIF, geometry = (100, 100))
sub_group_1 = neu_group_2[:50]
sub_group_2 = neu_group_2[:5, :5]
syn_conn_1 = bp.SynConn(model = AMPA,
                        pre_group = sub_group_2,
                        post_group = neu_group_2[:10, :10],
                        conn = bp.connect.All2All())
```

Users can schedule the order to run subfunctions in the neuron groups with member function `runner.set_schedule`. The input parameter of function `set_schedule` is a list of the functions users and `BrainPy` utilized in the model.

Example code:

```
neu.runner.set_schedule(['input', 'update', 'monitor', 'reset'])
```

Users can set parameters and initialize ST members after building neuron group. BrainPy also support heterogeneous parameter assignment: users can assign different values to the same ST member or parameter of different neurons in a neuron group.

Example code:

```
neu_group_1.pars['V_rest'] = np.random.randint(0, 2, size=(10,))

neu_group_1.ST['V'] = -65.
```

5.2 brainpy.SynConn

Users implement a group of synapses as an object of `SynConn` class. Nine input parameters are provided to the class contructor to describe the synapse connections:

- `model`: The neuron models used to generate the neuron group.
- `pars_update`: Parameters to update. (??? is this clear?)
- `pre_group`: Pre-synaptic neuron group.
- `post_group`: Post-synaptic neuron group.
- `conn`: Connection method to create synaptic connectivity.
- `num`: Number of synapses.
- `delay`: Time of synaptic delay.
- `monitors`: List of ST members to monitor.
- `name`: Name of neuron group. Will be automatically assigned if not given.

BrainPy implements two ways to define a set of synapse connection:

– 1. Assign values to the parameters pre_group, post_group and conn, and leave the parameter num blank. In this case, pre_group and post_group are two objects of NeuGroup class, and parameter conn specifies the connection strategy BrainPy will use to build synapses.

Eample code:

```
gabaa = bp.SynConn(model=GABAa_syn, pre_group=pre, post_group=post,
                   conn=bp.connect.All2All(), monitors=['s'], delay=10.)
```

– 2. Assign value to the parameter num, leave the parameters pre_group, post_group and conn blank. After SynConn object has been defined, users should assign value to four members of SynConn: pre, post, pre2syn and post2syn. In this case, num refers to the number of synapses, pre and post choose which neurons are to be connected, pre2syn and post2syn map the synapse and the pre-synaptic, post-synaptic neurons.

Example code:

```
syn = bp.SynConn(model = syn_model, num = delta_t_num,
                 monitors = ['w'], delay = 10.)
syn.pre = bp.types.NeuState(['spike'])(2)
syn.post = bp.types.NeuState(['V', 'input', 'spike'])(2)
pre2syn_list = [[1, 1], [2, 2]]
post2syn_list = [[1, 1], [2, 2]]
syn.pre2syn = syn.requires['pre2syn'].make_copy(pre2syn_list)
syn.post2syn = syn.requires['post2syn'].make_copy(post2syn_list)
```

Parameter delay refers to the synapse delay time. According to the biological nature of synapses, there is a time delay between the moment pre-synaptic spike affects the synapse state and the moment synapse state affects the post-synaptic spike. BrainPy realizes this delay with decorator @brainpy.delayed, and it will be automatically computed with proper defination of NeuType (See build synapses for detail).

Parameter monitors includes the ST members that will be recorded during simulation. If a ST member is listed in monitors, its time series will be saved in variable <NEU_GROUP_NAME>.mon.<MEM_NAME> for later use.

Example code:

```
w_time_series = syn.mon.w[START_TIME:END_TIME, NEURON_NO]
```

Users can schedule the order to run subfunctions in the synapse connections with member function runner.set_schedule. The input parameter of function set_schedule is a list of the functions users and BrainPy utilized in the model.

Example code:

```
syn.runner.set_schedule(['input', 'update', 'output', 'monitor'])
```

Users can set parameters and initialize ST members after building synapse connections. BrainPy also support heterogeneous parameter assignment: users can assign different values to the same ST member or parameter of different synapses in a synapse connection.

Example code:

```
syn_conn_1.pars['w'] = np.random.rand(syn_num)
```

5.3 brainpy.Network

To initialize a neural network, i.e. an object of `brainpy.Network` class, a list of `NeuGroup` and `SynConn` objects should be provided to the `Network` constructor as input parameters. These objects will compose the network.

- `*args`: List of objects.
- `mode`: If the model will be simulated repeatly.
- `**kwargs`: List of object names. If this parameter is not provided, the object will be automatically named.

These objects can be accessed via their names, e.g. `NET_NAME.OBJ_NAME`.

Example code:

```
net = bp.Network(neu_group_1, neu_group_2, syn_conn_1, syn_conn_2)
```

5.3.1 add

Users can add objects to the network using member function `add` of class `Network` after initialization is done. Input parameters of function `add` is similar to the ones of constructors function.

- `*args`: List of objects.
- `**kwargs`: List of object names. If this parameter is not provided, the object will be automatically named.

5.3.2 run

`BrainPy` run simulation with member function `run` of class `Network`.

- `duration` : The amount of simulation time (ms) to run for.
- `inputs` : List of external inputs, each element is a quadruple of (receiver, item, external input, [operation]), operation is set to '+' by default if not given. Default = ().
- `report` : Report the progress of the simulation. Default = false.
- `report_percent` : The interval of simulation progress reporting. Default = 0.1.

The items of `inputs` quadruple:

- `receiver`: The `NeuGroup` or `SynConn` object that receives the input.
- `item`: Specific item of the receiver that receives the input.
- `external input`: Description of the input time series.
- `operation`: Which operation should the item take when it receives the input.

`BrainPy` supports diverse input methods. The `operation` item can be set as one of { + , - , * , / , = }, and if users do not provide this item explictly, it will be set to '+' by default. The value of `operation` defines in which way the input will be given to the receive item. Generally, users can see it as a binary operator. For example, if `operation == '+'`, then in a single update, the value of receive item `val = val + input`.

Example code:

```
net.run(duration=100., inputs=[(neu_group_1, 'ST.input', 1.)], report=True)
```

After calling `run` function, `NeuGroup` and `SynConn` objects in the network will be updated in the way they were defined in their description, and in the order they were provided to the `Network` constructor as input parameters. We should mention that this order barely affects the simulation result, because the update interval `dt` are relatively small, therefore single update will not cause a sharp change in variable values.

Note:

i. Users can directly run `NeuGroup` with member function `NeuGroup.run`. However, `SynConn` can not be run in this way. See more details in build neurons and build synapses.

ii. Although the calculation order barely affects the simulation progress, users should be aware of the order and deal with relative assignments carefully, otherwise this feature may cause a bug.

5.4 E/I balance network

We take E/I balance network as an example. E/I balance network is a canonical network model, in which the populations of excitatory and inhibitory neurons achieve an approximate balance.

[10]:
```
# set global parameters
bp.profile.set(jit=True, device='cpu',
               numerical_method='exponential')

num_exc = 500
num_inh = 500
prob = 0.1
```

Neuron model is defined as:

$$\tau \frac{dV}{dt} = -(V - V_{rest} + I^{ext} + I^{net}(t))$$

With a threshold of $-50mV$ and a refractory of $5ms$.

[11]:
```
# define neuron model as bp.NeuType
V_rest = -52.
V_reset = -60.
V_th = -50.
R=1.
tau=10.
t_refractory=5.
noise=0.

ST_neu = bp.types.NeuState(
             {'V': 0, 'input': 0, 'spike': 0, 'refractory': 0, 't_last_spike': -1e7}
         )

@bp.integrate
def int_V(V, _t, I_ext):  # integrate u(t)
    return (- (V - V_rest) + R * I_ext) / tau, noise / tau

def update(ST, _t):
    # update variables
    ST['spike'] = 0
    if _t - ST['t_last_spike'] <= t_refractory:
        ST['refractory'] = 1.
```

(continues on next page)

(continued from previous page)

```
    else:
        ST['refractory'] = 0.
        V = int_V(ST['V'], _t, ST['input'])
        if V >= V_th:
            V = V_reset
            ST['spike'] = 1
            ST['t_last_spike'] = _t
        ST['V'] = V

def reset(ST):
    ST['input'] = 0.


neu = bp.NeuType(name='LIF',
                 ST=ST_neu,
                 steps=(update, reset),
                 mode='scalar')
```

Synapse model is defined as:

$$f(t) = \begin{cases} exp(-\frac{t}{\tau_s}), & t \geq 0, \\ 0, & t < 0. \end{cases} \tag{5.1}$$

$$I^{net}(t) = J_E \sum_{j=1}^{pN_E} \sum_{t_j^\alpha < t} f(t - t_j^\alpha) - J_I \sum_{j=1}^{pN_I} \sum_{t_j^\alpha < t} f(t - t_j^\alpha)$$

[12]:
```
# define synapse model as bp.SynType
tau_decay = 2.
JE = 1 / np.sqrt(prob * num_exc)
JI = 1 / np.sqrt(prob * num_inh)

ST_syn = bp.types.SynState({'s':0., 'w': .1, 'g':0.}, help='synapse state.')

@bp.integrate
def ints(s, t):
    return - s / tau_decay


def update(ST, _t, pre):
    s = ints(ST['s'], _t)
    s += pre['spike']
    ST['s'] = s
    ST['g'] = ST['w'] * s


def output(ST, post):
    post['input'] += ST['g']


syn = bp.SynType(name='alpha_synapse',
                 ST=ST_syn,
                 steps=(update, output),
                 mode='scalar')
```

After neuron model and synapse model are built, we generate the concrete neuron group and synapse connections.

Note that we assign heterogeneous initial values to ST members, and build synapse connections between two subgroups of one neuron group using the slice operation.

[13]:
```
# generate bp.NeuGroup & bp.SynConn
group = bp.NeuGroup(neu, geometry=num_exc + num_inh, monitors=['spike'])

group.ST['V'] = np.random.random(num_exc + num_inh) * (V_th - V_rest) + V_rest

exc_conn = bp.SynConn(syn,
                      pre_group=group[:num_exc],
                      post_group=group,
                      conn=bp.connect.FixedProb(prob=prob))
exc_conn.ST['w'] = JE

inh_conn = bp.SynConn(syn,
                      pre_group=group[num_exc:],
                      post_group=group,
                      conn=bp.connect.FixedProb(prob=prob))
exc_conn.ST['w'] = -JI
```

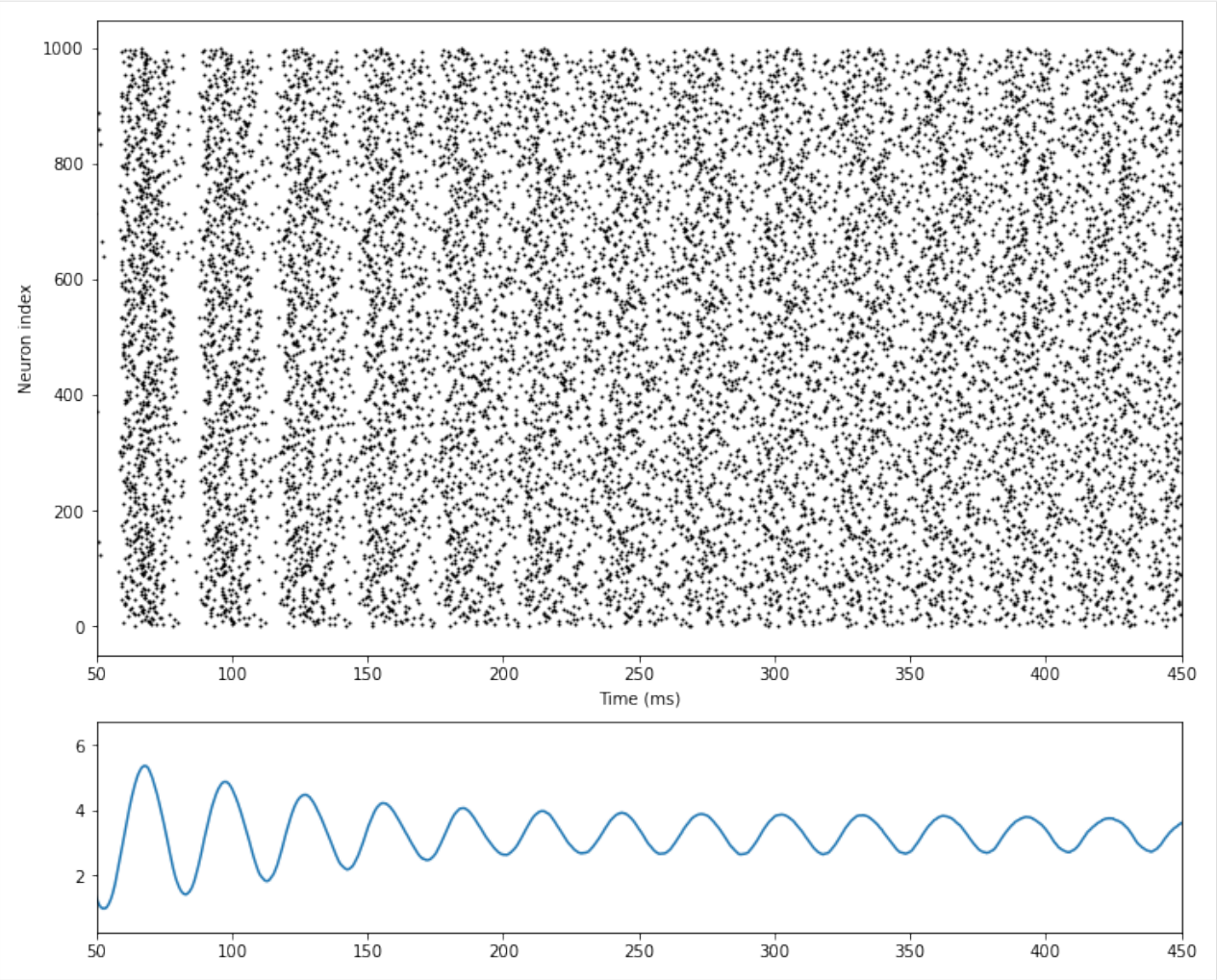
To integrate the neuron groups and synapse connection into a network, we initialize an object of `brainpy.Network` class with these previous defined objects, then run the network with member function `run`. In this progress, we simulate the network for 500 ms, give a constant input of amplitude `3.` to all the neurons in neuron group, and report the simulation progress.

[14]:
```
net = bp.Network(group, exc_conn, inh_conn)
net.run(duration=500., inputs=(group, 'ST.input', 3.), report=True)
```

```
Compilation used 0.5022 s.
Start running ...
Run 10.0% used 0.088 s.
Run 20.0% used 0.174 s.
Run 30.0% used 0.275 s.
Run 40.0% used 0.393 s.
Run 50.0% used 0.483 s.
Run 60.0% used 0.577 s.
Run 70.0% used 0.665 s.
Run 80.0% used 0.770 s.
Run 90.0% used 0.857 s.
Run 100.0% used 0.942 s.
Simulation is done in 0.942 s.
```

After visualization, users can see the typical oscillation pattern of E/I balance network.

[16]:
```
fig, gs = bp.visualize.get_figure(4, 1, 2, 10)

fig.add_subplot(gs[:3, 0])
bp.visualize.raster_plot(net.ts, group.mon.spike, xlim=(50, 450))

fig.add_subplot(gs[3, 0])
rates = bp.measure.firing_rate(group.mon.spike, 5.)
plt.plot(net.ts, rates)
plt.xlim(50, 450)
plt.show()
```


Neuron index
1000
800
600
400
200
0
50
100
150
200
250
300
350
400
450
Time (ms)
6
4
2
50
100
150
200
250
300
350
400
450

CHAPTER

SIX

DYNAMICS ANALYSIS

BrainPy provides fundamental methods for dynamics analysis of neuron models, including:

- (1) phase plane analysis for 1-dimensional and 2-dimensional systems;
- (2) codimension 1 and codimension 2 bifurcation analysis.

We take FitzHugh-Nagumo model as example to demonstrate how to perform dynamics analysis with BrainPy API.

The model is given by:

$$\frac{dV}{dt} = V(1 - \frac{V^2}{3}) - w + I_{ext}$$

$$\tau \frac{dw}{dt} = V + a - bw$$

There are two variables V and w, so this is a two-dimensional system with three parameters a, b and τ.

Let's start by defining the model.

[1]:
```
import brainpy as bp
import numpy as np

bp.profile.set(dt=0.02, numerical_method='rk4')


def get_FNmodel(a=0.7, b=0.8, tau=12.5, Vth=1.9):
    state = bp.types.NeuState({'v': 0., 'w': 1., 'spike': 0., 'input': 0.})

    @bp.integrate
    def int_w(w, t, v):
        return (v + a - b * w) / tau

    @bp.integrate
    def int_v(v, t, w, Iext):
        return v - v * v * v / 3 - w + Iext

    def update(ST, _t):
        ST['w'] = int_w(ST['w'], _t, ST['v'])
        v = int_v(ST['v'], _t, ST['w'], ST['input'])
        ST['spike'] = np.logical_and(v >= Vth, ST['v'] < Vth)
        ST['v'] = v
        ST['input'] = 0.

    return bp.NeuType(name='FitzHugh_Nagumo',
                      ST=state,
                      steps=update)
```

6.1 Phase Plane Analysis

We provide `brainpy.PhasePortraitAnalyzer` to support phase plane analysis for 1-dimensional and 2-dimensional dynamical systems.

Two parameters should be specified to initialize a `PhasePortraitAnalyzer`:

- `model`: The neuron model to be analysis.
- `target_vars`: The variables to be analysis and its value range.

And two parameters are optional:

- `fixed_vars`: The slow variables to be fixed as stational variables (Optional for higher order system).
- `pars_update`: Parameters to update.

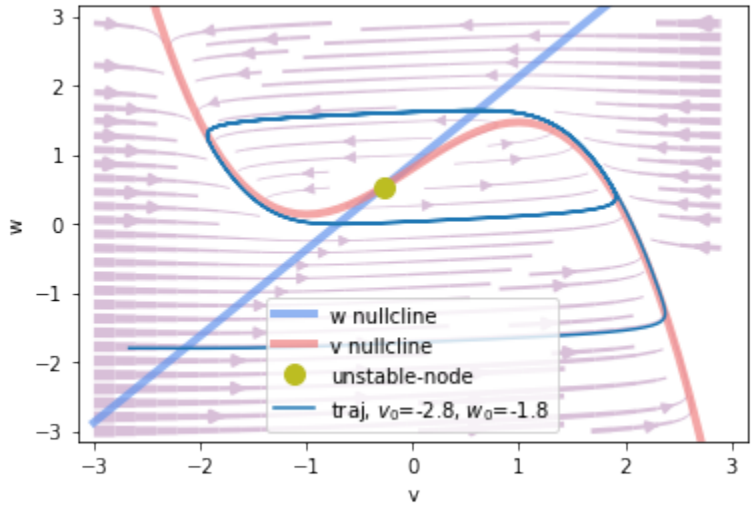
After defining a `PhasePortraitAnalyzer`, you can call the following functions:

- `plot_nullcline()`: Plot the nullclines.
- `plot_vector_field()`: Plot the vector field.
- `plot_fixed_point()`: Find and plot the fixed points, and perform stability analysis (print to the terminal).
- `plot_trajectory()`: Plot trajectories according to the settings (initial var1, initial var2, duration).

Here we perform a phase plane analysis with parameters $a = 0.7, b = 0.8, \tau = 12.5$, and input $I_{ext} = 0.8$.

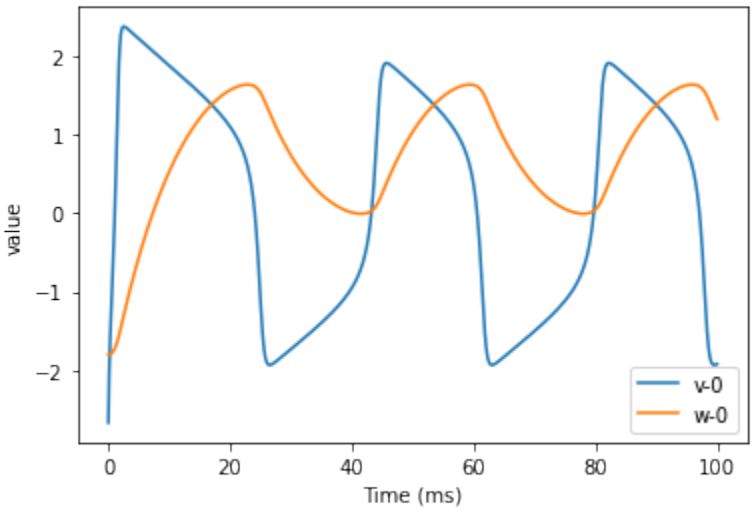
[2]:
```
neuron = get_FNmodel(a=0.7, b=0.8, tau=12.5)

analyzer = bp.PhasePortraitAnalyzer(
    model=neuron,
    target_vars={'v': [-3, 3], 'w': [-3., 3.]},
    fixed_vars={'Iext': 0.8})
analyzer.plot_nullcline()
analyzer.plot_vector_field()
analyzer.plot_fixed_point()
analyzer.plot_trajectory([(-2.8, -1.8, 100.)],
                         inputs=('ST.input', 0.8),
                         show=True)
```

```
Fixed point #1 at v=-0.2729009589972752, w=0.5338738012534059 is a unstable-node.
```



We can see an unstable-node at the point (v=-0.27, w=0.53) inside a limit cycle. Then we can run a simulation with the same parameters and initial values to see the periodic activity that correspond to the limit cycle.

[3]:
```
group = bp.NeuGroup(neuron, 1, monitors=['v', 'w'])
group.ST['v'] = -2.8
group.ST['w'] = -1.8
group.run(100., inputs=('ST.input', 0.8))
bp.visualize.line_plot(group.mon.ts, group.mon.v, legend='v', )
bp.visualize.line_plot(group.mon.ts, group.mon.w, legend='w', show=True)
```



6.2 Bifurcation Analysis

We provide `brainpy.BifurcationAnalyzer` for users to perform bifucation analysis.

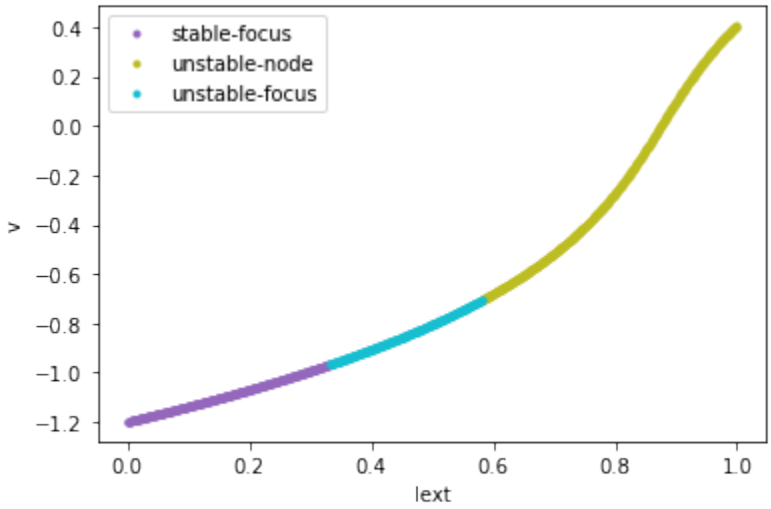
The `PhasePortraitAnalyzer` receives the following parameters:

- `model`: The neuron model to be analysis.
- `target_pars`: The parameters to be change and the ranges.
- `dynamical_vars`: The variables of the system and the change ranges.
- `par_resolution`: The numerical resolution of the bifurcation analysis.

6.2.1 Codimension 1 bifurcation analysis

We will first see the codimension 1 bifurcation anlysis of the model. For example, we vary the input I_{ext} between 0 to 1 and see how the system change it's stability.

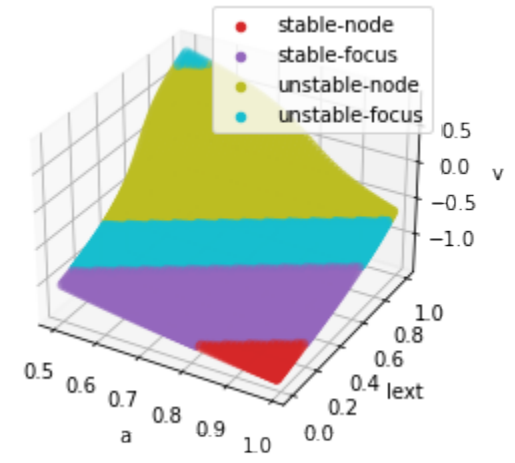
[4]:
```
analyzer = bp.BifurcationAnalyzer(
    model=neuron,
    target_pars={'Iext': [0., 1.]},
    dynamical_vars={'v': [-3, 3], 'w': [-3., 3.]},
    par_resolution=0.001,
)
analyzer.plot_bifurcation(plot_vars=['v'], show=True)
```



6.2.2 Codimension 2 bifurcation analysis

We simulaneously change I_{ext} and parameter a.

[5]:
```
analyzer = bp.BifurcationAnalyzer(
    model=neuron,
    target_pars={'a': [0.5, 1.], 'Iext': [0., 1.]},
    dynamical_vars={'v': [-3, 3], 'w': [-3., 3.]},
    par_resolution=0.01,
)
analyzer.plot_bifurcation(plot_vars=['v'], show=True)
```



CHAPTER

SEVEN

DIFFERENTIAL EQUATIONS

[1]:
```
import brainpy as bp
```

In BrainPy, the difinition of differential equations is supportted by a powerfull decorator `@bp.integrate`. Users should only explicitly write out the right hand of the differential equations, and BrainPy will automatically integerates your defined differential equations.

BrainPy supports the numerical integration of ordinary differential equations (ODEs) and stochastic differential equations (SDEs).

7.1 ODEs

For an ordinary differential equation

$$\frac{dx}{dt} = f(x, t)$$

the coding in BrainPy has a general form of:

```
@bp.integrate
def func(x, t, other_arguments):
    # ... do some computation
    f = ...
    return f

x_t_plus = func(x_t, t, other_arguments)
```

7.2 SDEs

For the stochastic differential equation:

$$\frac{dx}{dt} = f(x, t) + g(x, t)dW$$

the coding in BrainPy can be conducted as:

```
@bp.integrate
def func(x, t, other_arguments):
    # ... do some computation
    f = ...
    g = ...
```

(continues on next page)

(continued from previous page)

```
    return f, g

x_t_plus = func(x_t, t, other_arguments)
```

7.3 Return intermediate values

BrainPy also supports the user return the intermediate computed results. Let's take the differential equation of V in Hodgkin–Huxley (HH) neuron model as an example. In HH model, the stochastic differential equation V is expressed as:

$$C_m \frac{dV}{dt} = -\bar{g}_K n^4 (V - V_K) - \bar{g}_{Na} m^3 h (V - V_{Na}) - \bar{g}_l (V - V_l) + I^{ext} + I^{noise} * dW, \tag{7.1}$$

where

- the potassium channel current is $I_K = \bar{g}_K n^4 (V - V_K)$,
- the sodium channel current is $I_{Na} = \bar{g}_{Na} m^3 h (V - V_{Na})$, and
- the leaky current is $I_L = \bar{g}_l (V - V_l)$.

The user may not only has the interest of the final value V, but also take care of the intermediate value I_{Na}, I_K and I_L. In BrainPy, this kind of requirement can be coded as:

```
@bp.integrate
def func(V, t, m, h, n, Iext):
    INa = gNa * m ** 3 * h * (V - ENa)
    IK = gK * n ** 4 * (V - EK)
    IL = gLeak * (V - ELeak)
    f = (- INa - IK - IL + Isyn) / C
    g = noise / C
    return (f, g), INa, IK, IL

V_t_plus, INa, IK, IL = func(V_t, t, m, h, n, Iext)
```

Generally, return intermediate values in ODE function can be coded as:

```
@bp.integrate
def func(x, t, other_arguments):
    # ... do some computation
    f = ...
    return (f, ), some_values
```

Return intermediate values in SDE function can be coded as:

```
@bp.integrate
def func(x, t, other_arguments):
    # ... do some computation
    f = ...
    g = ...
    return (f, g), some_values
```

CHAPTER

EIGHT

NUMERICAL INTEGRATORS

`BrainPy` provides several methods for the numerical integration of Ordinary Differential Equations (ODEs) and Stochastic Differential Equations (SDEs).

Method	keyword for use	ODE	SDE
Euler	euler	Y	Y
ExponentialEuler	exponential	Y	Y
MidPoint	midpoint	Y	N
Heun	heun	Y	Y
RK2	rk2	Y	coming soon
RK3	rk3	Y	coming soon
RK4	rk4	Y	coming soon
RK4Alternative	rk4_alternative	Y	N
MilsteinIto	milstein_ito	Y	Y
MilsteinStra	milstein_stra	Y	Y

CHAPTER

NINE

DEBUGGING

Even if you write clear and readable code, even if you fully understand your codes, env if you are very familiar with your model, weird bugs will inevitably appear and you will need to debug them in some way.

Fortunately, `BrainPy` supports debugging with pdb module or breakpoint (The latest version of BrainPy removes the support of debugging in IDEs). That is to say, you do not need to resort to using bunch of `print` statements to see what's happening in their code. On the contrary, you can work with Python's interactive source code debugger to see the state of any variable in your model.

For the variables you are interested in, you just need to add the `pdb.set_trace()` or `breakpoint()` after the code line.

In this section, let's take the HH neuron model as an example to illustrate how to debug your model within BrainPy.

[1]:
```
import brainpy as bp
import numpy as np
import pdb
```

If you want to debug your model, we would like to recommond you to open the `show_code=True`.

[2]:
```
bp.profile.set(show_code=True, jit=False)
```

Here, the HH neuron model is defined as:

[3]:
```
E_Na = 50.
E_K = -77.
E_leak = -54.387
C = 1.0
g_Na = 120.
g_K = 36.
g_leak = 0.03
V_th = 20.
noise = 1.

ST = bp.types.NeuState(
    {'V': -65., 'm': 0.05, 'h': 0.60,
     'n': 0.32, 'spike': 0., 'input': 0.}
)

@bp.integrate
def int_m(m, _t, V):
    alpha = 0.1 * (V + 40) / (1 - np.exp(-(V + 40) / 10))
    beta = 4.0 * np.exp(-(V + 65) / 18)
    return alpha * (1 - m) - beta * m
```

(continues on next page)

(continued from previous page)

```
@bp.integrate
def int_h(h, _t, V):
    alpha = 0.07 * np.exp(-(V + 65) / 20.)
    beta = 1 / (1 + np.exp(-(V + 35) / 10))
    return alpha * (1 - h) - beta * h

@bp.integrate
def int_n(n, _t, V):
    alpha = 0.01 * (V + 55) / (1 - np.exp(-(V + 55) / 10))
    beta = 0.125 * np.exp(-(V + 65) / 80)
    return alpha * (1 - n) - beta * n

@bp.integrate
def int_V(V, _t, m, h, n, I_ext):
    I_Na = (g_Na * np.power(m, 3.0) * h) * (V - E_Na)
    I_K = (g_K * np.power(n, 4.0))* (V - E_K)
    I_leak = g_leak * (V - E_leak)
    dVdt = (- I_Na - I_K - I_leak + I_ext)/C
    return dVdt, noise / C

def update(ST, _t):
    m = np.clip(int_m(ST['m'], _t, ST['V']), 0., 1.)
    h = np.clip(int_h(ST['h'], _t, ST['V']), 0., 1.)
    n = np.clip(int_n(ST['n'], _t, ST['V']), 0., 1.)
    V = int_V(ST['V'], _t, m, h, n, ST['input'])

    pdb.set_trace()

    spike = np.logical_and(ST['V'] < V_th, V >= V_th)
    ST['spike'] = spike
    ST['V'] = V
    ST['m'] = m
    ST['h'] = h
    ST['n'] = n
    ST['input'] = 0.

HH = bp.NeuType(ST=ST,
                name='HH_neuron',
                steps=update,
                mode='vector')
```

In this example, we add `pdb.set_trace()` after the variables m, h, n and V.

Then we can create a neuron group, and try to run this neuron model:

[4]:
```
group = bp.NeuGroup(HH, geometry=1, monitors=['spike'])
group.run(1000., inputs=('input', 10.))
```

```
def NeuGroup0_input_step(ST, input_inp,):
  # "input" step function of NeuGroup0
  ST[5] += input_inp



def NeuGroup0_monitor_step(ST, _i, mon_ST_spike,):
  # "monitor" step function of NeuGroup0
  mon_ST_spike[_i] = ST[4]
```

(continues on next page)

(continued from previous page)

```
def NeuGroup0_update(ST, _t,):
 # "update" step function of NeuGroup0
 _int_m_m = ST[1]
 _int_m__t = _t
 _int_m_V = ST[0]
 _int_m_alpha = 0.1 * (_int_m_V + 40) / (1 - np.exp(-(_int_m_V + 40) / 10))
 _int_m_beta = 4.0 * np.exp(-(_int_m_V + 65) / 18)
 _dfm_dt = _int_m_alpha * (1 - _int_m_m) - _int_m_beta * _int_m_m
 _int_m_m = 0.1*_dfm_dt + _int_m_m
 _int_m_res = _int_m_m
 m = np.clip(_int_m_res, 0.0, 1.0)

 _int_h_h = ST[2]
 _int_h__t = _t
 _int_h_V = ST[0]
 _int_h_alpha = 0.07 * np.exp(-(_int_h_V + 65) / 20.0)
 _int_h_beta = 1 / (1 + np.exp(-(_int_h_V + 35) / 10))
 _dfh_dt = _int_h_alpha * (1 - _int_h_h) - _int_h_beta * _int_h_h
 _int_h_h = 0.1*_dfh_dt + _int_h_h
 _int_h_res = _int_h_h
 h = np.clip(_int_h_res, 0.0, 1.0)

 _int_n_n = ST[3]
 _int_n__t = _t
 _int_n_V = ST[0]
 _int_n_alpha = 0.01 * (_int_n_V + 55) / (1 - np.exp(-(_int_n_V + 55) / 10))
 _int_n_beta = 0.125 * np.exp(-(_int_n_V + 65) / 80)
 _dfn_dt = _int_n_alpha * (1 - _int_n_n) - _int_n_beta * _int_n_n
 _int_n_n = 0.1*_dfn_dt + _int_n_n
 _int_n_res = _int_n_n
 n = np.clip(_int_n_res, 0.0, 1.0)

 _int_V_V = ST[0]
 _int_V__t = _t
 _int_V_m = m
 _int_V_h = h
 _int_V_n = n
 _int_V_I_ext = ST[5]
 _int_V_I_Na = g_Na * np.power(_int_V_m, 3.0) * _int_V_h * (_int_V_V - E_Na)
 _int_V_I_K = g_K * np.power(_int_V_n, 4.0) * (_int_V_V - E_K)
 _int_V_I_leak = g_leak * (_int_V_V - E_leak)
 _int_V_dVdt = (-_int_V_I_Na - _int_V_I_K - _int_V_I_leak + _int_V_I_ext) / C
 _dfV_dt = _int_V_dVdt
 _V_dW = _normal_like(_int_V_V)
 _dgV_dt = noise / C
 _int_V_V = _int_V_V + 0.316227766016838*_V_dW*_dgV_dt + 0.1*_dfV_dt
 _int_V_res = _int_V_V
 V = _int_V_res

 pdb.set_trace()

 spike = np.logical_and(ST[0] < V_th, V >= V_th)
 ST[4] = spike
 ST[0] = V
```

(continues on next page)

(continued from previous page)

```
 ST[1] = m
 ST[2] = h
 ST[3] = n
 ST[5] = 0.0



def step_func(_t, _i, _dt):
  NeuGroup0_runner.input_step(NeuGroup0.ST["_data"], NeuGroup0_runner.input_inp,)
  NeuGroup0_runner.update(NeuGroup0.ST["_data"], _t,)
  NeuGroup0_runner.monitor_step(NeuGroup0.ST["_data"], _i, NeuGroup0.mon["spike"],)

> c:\users\oujag\codes\projects\brainpy\docs\advanced(52)update()

ipdb> p m
array([0.05123855])
ipdb> p n
array([0.31995744])
ipdb> p h
array([0.59995445])
ipdb> n
> c:\users\oujag\codes\projects\brainpy\docs\advanced(53)update()

ipdb> n
> c:\users\oujag\codes\projects\brainpy\docs\advanced(54)update()

ipdb> n
> c:\users\oujag\codes\projects\brainpy\docs\advanced(55)update()

ipdb> n
> c:\users\oujag\codes\projects\brainpy\docs\advanced(56)update()

ipdb> n
> c:\users\oujag\codes\projects\brainpy\docs\advanced(57)update()

ipdb> n
> c:\users\oujag\codes\projects\brainpy\docs\advanced(58)update()

ipdb> p ST
array([[-6.41827214e+01],
       [ 5.12385538e-02],
       [ 5.99954448e-01],
       [ 3.19957442e-01],
       [ 0.00000000e+00],
       [ 1.00000000e+01]])
ipdb> q
```

```
---------------------------------------------------------------------------
BdbQuit                                   Traceback (most recent call last)
<ipython-input-4-703915fa7809> in <module>
      1 group = bp.NeuGroup(HH, geometry=1, monitors=['spike'])
----> 2 group.run(1000., inputs=('input', 10.))

~\codes\projects\BrainPy\brainpy\core\base.py in run(self, duration, inputs, report,␣
↪report_percent)
    584         else:
    585             for run_idx in range(run_length):
```

(continues on next page)

(continued from previous page)

```
--> 586                 step_func(_t=times[run_idx], _i=run_idx, _dt=dt)
    587
    588         if profile.run_on_gpu():

? in step_func(_t, _i, _dt)

? in update(ST, _t)

? in update(ST, _t)

~\Miniconda3\envs\py38\lib\bdb.py in trace_dispatch(self, frame, event, arg)
     86             return # None
     87         if event == 'line':
---> 88             return self.dispatch_line(frame)
     89         if event == 'call':
     90             return self.dispatch_call(frame, arg)

~\Miniconda3\envs\py38\lib\bdb.py in dispatch_line(self, frame)
    111         if self.stop_here(frame) or self.break_here(frame):
    112             self.user_line(frame)
--> 113             if self.quitting: raise BdbQuit
    114         return self.trace_dispatch
    115

BdbQuit:
```

CHAPTER

TEN

REPEAT MODE OF `NETWORK`

Another simple but powerful function provided in `BrainPy` is the repeat running mode of `brainpy.Network`. This function allows users to run a compiled network model repeatly. Specifically, the repeat mode can be done in a `brainpy.Network` instantiation with the keyword `mode='repeat'`, i.e., `net = brainpy.Network(mode='repeat')`.

Before each function call of `net.run()`, users can arbitrarily set the values in `ST` state, or the input values in `inputs`. However, user's parameter updating `model.pars['xx'] = xx` can not be done. This is because BrainPy compiles the `ST` and `inputs` as dynamical variables and treat them as the functional arguments, while for the `pars`, BrainPy recognizes them as the static variables and can be changed after the model compilation.

Repeat mode of `brainpy.Network` is very useful at least in the following two situations: *parameter searching* and *RAM memory saving*.

10.1 Parameter Searching

`Parameter searching` is one of the most common things in computational modeling. When creating a model, we'll be presented with many parameters to control how our defined model evolves. Often times, we don't immediately know what the optimal parameter set should be for a given model, and thus we'd like to be able to explore a range of possibilities.

Fortunately, with the `repeat` mode provided in `brainpy.Network`, parameter searching is a very easy thing.

Here, we illustrate this with the example of gamma oscillation, and to see how different value of `g_max` (the maximal synaptic conductance) affect the network coherence.

First, let's import the necessary packages and define the models first.

[1]:
```
import numpy as np
import brainpy as bp
import matplotlib.pyplot as plt

bp.profile.set(jit=True, dt=0.04, numerical_method='exponential')
```

Same with gamma oscillation, the HH neuron model is defined as:

[2]:
```
# HH neuron model #
# --------------- #

V_th = 0.
C = 1.0
gLeak = 0.1
ELeak = -65
```

(continues on next page)

(continued from previous page)

```
gNa = 35.
ENa = 55.
gK = 9.
EK = -90.
phi = 5.0

HH_ST = bp.types.NeuState({'V': -55., 'h': 0., 'n': 0., 'spike': 0., 'inp': 0.})

@bp.integrate
def int_h(h, t, V):
    alpha = 0.07 * np.exp(-(V + 58) / 20)
    beta = 1 / (np.exp(-0.1 * (V + 28)) + 1)
    dhdt = alpha * (1 - h) - beta * h
    return phi * dhdt

@bp.integrate
def int_n(n, t, V):
    alpha = -0.01 * (V + 34) / (np.exp(-0.1 * (V + 34)) - 1)
    beta = 0.125 * np.exp(-(V + 44) / 80)
    dndt = alpha * (1 - n) - beta * n
    return phi * dndt

@bp.integrate
def int_V(V, t, h, n, Isyn):
    m_alpha = -0.1 * (V + 35) / (np.exp(-0.1 * (V + 35)) - 1)
    m_beta = 4 * np.exp(-(V + 60) / 18)
    m = m_alpha / (m_alpha + m_beta)
    INa = gNa * m ** 3 * h * (V - ENa)
    IK = gK * n ** 4 * (V - EK)
    IL = gLeak * (V - ELeak)
    dvdt = (- INa - IK - IL + Isyn) / C
    return dvdt

def neu_update(ST, _t):
    h = int_h(ST['h'], _t, ST['V'])
    n = int_n(ST['n'], _t, ST['V'])
    V = int_V(ST['V'], _t, ST['h'], ST['n'], ST['inp'])
    sp = np.logical_and(ST['V'] < V_th, V >= V_th)
    ST['spike'] = sp
    ST['V'] = V
    ST['h'] = h
    ST['n'] = n
    ST['inp'] = 0.

HH = bp.NeuType(name='HH_neuron', ST=HH_ST, steps=neu_update)
```

Different from the orignal definition, here we add the parameter `g_max` into the `ST`, because later we will change the value of `g_max` and rerun the model.

[3]:
```
# GABAa #
# ----- #

g_max = 0.1
E = -75.
alpha = 12.
beta = 0.1
```

(continues on next page)

(continued from previous page)

```
@bp.integrate
def int_s(s, t, TT):
    return alpha * TT * (1 - s) - beta * s

def syn_update(ST, _t, pre):
    T = 1 / (1 + np.exp(-(pre['V'] - V_th) / 2))
    s = int_s(ST['s'], _t, T)
    ST['s'] = s

def syn_output(ST, post):
    post['inp'] -= ST['g_max'] * ST['s'] * (post['V'] - E)

GABAa = bp.SynType(name='GABAa',
                   ST=bp.types.SynState(['g_max', 's']),
                   steps=(syn_update, syn_output),
                   requires=dict(pre=bp.types.NeuState(['V']),
                                 post=bp.types.NeuState(['V', 'inp'])),
                   mode='scalar')
```

Putting the HH neuron and the GABAA syanpse together, let's define the network in which HH neurons are interconnected with the GABAA syanpses. Here, what's different is we add a term `mode='repeat'` into the `bp.Network` definition.

[4]:
```
# Network #
# ------- #

num = 100
v_init = -70. + np.random.random(num) * 20
h_alpha = 0.07 * np.exp(-(v_init + 58) / 20)
h_beta = 1 / (np.exp(-0.1 * (v_init + 28)) + 1)
h_init = h_alpha / (h_alpha + h_beta)
n_alpha = -0.01 * (v_init + 34) / (np.exp(-0.1 * (v_init + 34)) - 1)
n_beta = 0.125 * np.exp(-(v_init + 44) / 80)
n_init = n_alpha / (n_alpha + n_beta)

group = bp.NeuGroup(HH, geometry=num, monitors=['spike'])
conn = bp.SynConn(GABAa, pre_group=group, post_group=group,
                  conn=bp.connect.All2All(include_self=False))
net = bp.Network(group, conn, mode='repeat')
```

Now, by using the cross correlation measurement, we can evaluate the network coherence under the different parameter setting of `g_max`.

[5]:
```
# Parameter Searching #
# ------------------- #

all_g_max = np.arange(0.05, 0.151, 0.01) / num
all_cc = []

for i, g_max in enumerate(all_g_max):
    print('When g_max = {:.5f} ...'.format(g_max))

    group.ST['V'] = v_init
    group.ST['h'] = h_init
    group.ST['n'] = n_init
```

(continues on next page)

(continued from previous page)

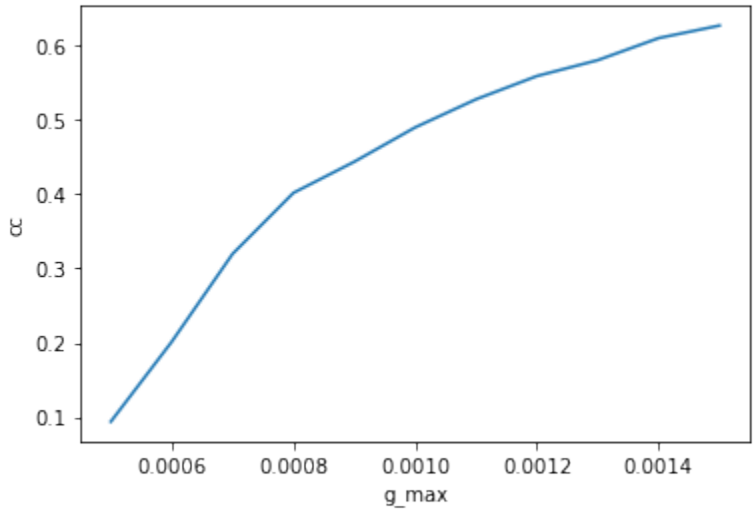
```
    group.ST['spike'] = 0.
    group.ST['inp'] = 0.

    conn.ST['s'] = 0.
    conn.ST['g_max'] = g_max

    report = i < 2
    net.run(duration=500., inputs=[group, 'ST.inp', 1.2], report=report, report_
↪percent=0.2)

    cc = bp.measure.cross_correlation(group.mon.spike, bin_size=int(0.5 / bp.profile.
↪get_dt()))
    all_cc.append(cc)

    if report: print('\n')
```

```
When g_max = 0.00050 ...
Compilation used 3.1810 s.
Start running ...
Run 20.0% used 0.669 s.
Run 40.0% used 1.344 s.
Run 60.0% used 2.011 s.
Run 80.0% used 2.681 s.
Run 100.0% used 3.352 s.
Simulation is done in 3.352 s.


When g_max = 0.00060 ...
Compilation used 0.0000 s.
Start running ...
Run 20.0% used 0.670 s.
Run 40.0% used 1.345 s.
Run 60.0% used 2.021 s.
Run 80.0% used 2.702 s.
Run 100.0% used 3.377 s.
Simulation is done in 3.377 s.


When g_max = 0.00070 ...
When g_max = 0.00080 ...
When g_max = 0.00090 ...
When g_max = 0.00100 ...
When g_max = 0.00110 ...
When g_max = 0.00120 ...
When g_max = 0.00130 ...
When g_max = 0.00140 ...
When g_max = 0.00150 ...
```

As you can see, the network was only compiled at the fist time run. And the overall speed of the later running does not change.

Finally, we can plot the relationship between the `g_max` and network coherence `cc`.

[6]:
```
plt.plot(all_g_max, all_cc)
plt.xlabel('g_max')
plt.ylabel('cc')
plt.show()
```



It is worthy to note that in this example, before each `net.run()`, we reset the `ST` state of the neuron group and the synaptical connection. This is because each repeat run is independent with each other in the case of the parameter tuning. However, in the following example, the current `net.run()` relies on the previous network running, the `ST` state should not be reset.

10.2 Memory Saving

Another annoyance often occurs is that our computers have limited RAM memory. Once the model size is big, or the running duration is long, `MemoryError` usually occurs.

Here, with `brainpy.Network` repeat running mode, BrainPy can partially solve this problem by allowing uers to split a long duration into multiple short durations. BrainPy allows user to repeatedly call `run()`, but with the same `inputs` structure and the same runing duration length. In this section, we illustrate this function by using the above defined gamma oscillation network.

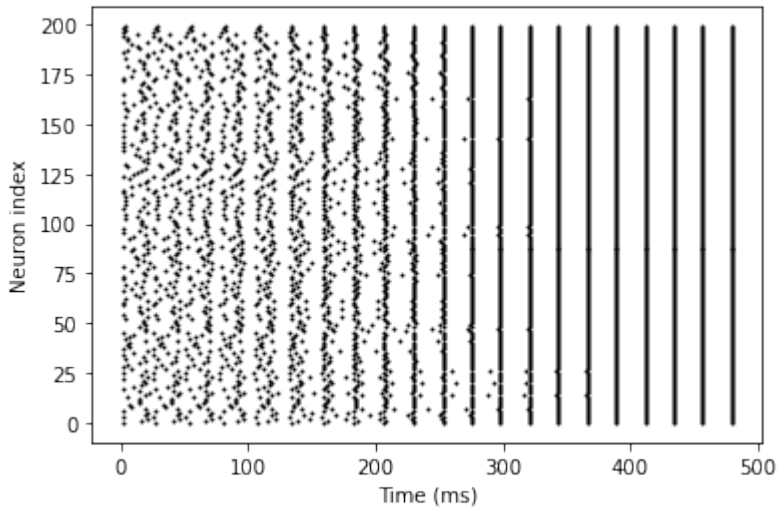
We define a network with the size of 200 HH neurons, and try to run this network in 2 seconds.

[8]:
```
num = 200
v_init = -70. + np.random.random(num) * 20
h_alpha = 0.07 * np.exp(-(v_init + 58) / 20)
h_beta = 1 / (np.exp(-0.1 * (v_init + 28)) + 1)
h_init = h_alpha / (h_alpha + h_beta)
n_alpha = -0.01 * (v_init + 34) / (np.exp(-0.1 * (v_init + 34)) - 1)
n_beta = 0.125 * np.exp(-(v_init + 44) / 80)
n_init = n_alpha / (n_alpha + n_beta)

group = bp.NeuGroup(HH, geometry=num, monitors=['spike'])
conn = bp.SynConn(GABAa, pre_group=group, post_group=group,
                  conn=bp.connect.All2All(include_self=False))
net = bp.Network(group, conn, mode='repeat')

group.ST['V'] = v_init
group.ST['h'] = h_init
group.ST['n'] = n_init

conn.ST['g_max'] = 0.1 / num
```

Here, we do not run the total 2 second at one time. On the contrary, we run the model with four steps, with each step of 0.5 second running duration.

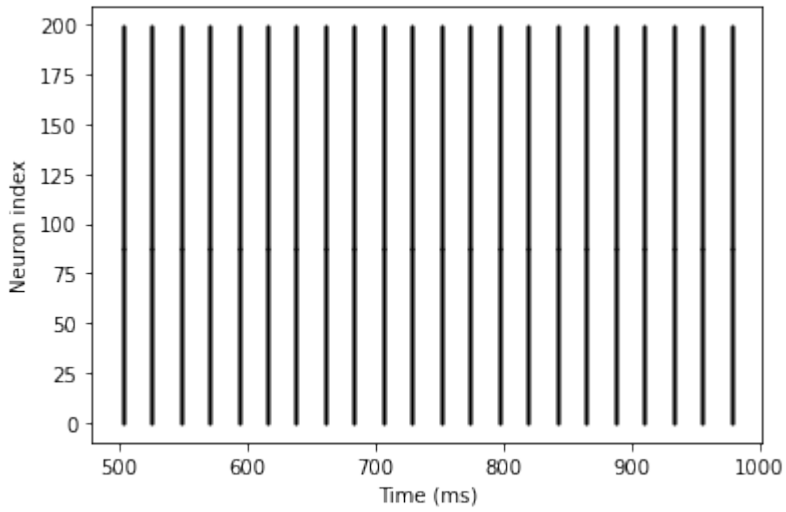
[10]:
```
# run 1: 0 - 500 ms

net.run(duration=(0., 500.), inputs=[group, 'ST.inp', 1.2], report=True, report_
↪percent=0.2)
bp.visualize.raster_plot(net.ts, group.mon.spike, show=True)
```

```
Compilation used 0.0010 s.
Start running ...
Run 20.0% used 2.525 s.
Run 40.0% used 5.052 s.
Run 60.0% used 7.562 s.
Run 80.0% used 10.073 s.
Run 100.0% used 12.573 s.
Simulation is done in 12.573 s.
```



[11]:
```
# run 2: 500 - 1000 ms

net.run(duration=(500., 1000.), inputs=[group, 'ST.inp', 1.2], report=True, report_
↪percent=0.2)
bp.visualize.raster_plot(net.ts, group.mon.spike, show=True)
```

```
Compilation used 0.0010 s.
Start running ...
Run 20.0% used 2.540 s.
Run 40.0% used 5.042 s.
Run 60.0% used 7.550 s.
Run 80.0% used 10.072 s.
Run 100.0% used 12.587 s.
Simulation is done in 12.587 s.
```



Set a different running duration, the `ModelUseError` will occur.

[13]:
```
# run 1000 - 2000 ms

net.run(duration=(1000., 2000.), inputs=[group, 'ST.inp', 1.2], report=True, report_
↪percent=0.2)
bp.visualize.raster_plot(net.ts, group.mon.spike, show=True)
```

```
---------------------------------------------------------------------------
ModelUseError                             Traceback (most recent call last)
<ipython-input-13-bb2a6051d2e9> in <module>
      1 # run 1000 - 2000 ms
      2
----> 3 net.run(duration=(1000., 2000.), inputs=[group, 'ST.inp', 1.2], report=True,␣
↪report_percent=0.2)
      4 bp.visualize.raster_plot(net.ts, group.mon.spike, show=True)

D:\codes\Projects\BrainPy\brainpy\core\network.py in run(self, duration, inputs,␣
↪report, report_percent)
    227             # ----------------------
    228             if self.t_duration != (end - start):
--> 229                 raise ModelUseError(f'Each run in "repeat" mode must be done '
    230                                     f'with the same duration, but got '
    231                                     f'{self.t_duration} != {end - start}.')

ModelUseError: Each run in "repeat" mode must be done with the same duration, but got␣
↪500.0 != 1000.0.
```

Set a different inputs structure, the `ModelUseError` will also occur.

[14]:
```
# run 1000 - 1500 ms

Iext = np.ones(num) * 1.2
net.run(duration=(1000., 1500.), inputs=[group, 'ST.inp', Iext],
        report=True, report_percent=0.2)
bp.visualize.raster_plot(net.ts, group.mon.spike, show=True)
```

```
---------------------------------------------------------------------------
ModelUseError                             Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-14-051b5873270f> in <module>
      3 Iext = np.ones(num) * 1.2
      4 net.run(duration=(1000., 1500.), inputs=[group, 'ST.inp', Iext],
----> 5         report=True, report_percent=0.2)
      6 bp.visualize.raster_plot(net.ts, group.mon.spike, show=True)

D:\codes\Projects\BrainPy\brainpy\core\network.py in run(self, duration, inputs,␣
↪report, report_percent)
    240                 for key, val, ops, data_type in inps:
    241                     if np.shape(obj_inputs[key][0]) != np.shape(val):
--> 242                         raise ModelUseError(f'The input shape for "{key}"␣
↪should keep the same. '
    243                                             f'However, we got the last input␣
↪shape '
    244                                             f'= {np.shape(obj_inputs[key][0])}
↪, '

ModelUseError: The input shape for "ST.inp" should keep the same. However, we got the␣
↪last input shape = (), and the current input shape = (200,)
```

Another thing worthy noting is that if the model `ST` state relies on the time (for example, the LIF neuron model `ST['t_last_spike']`). Setting the continuous time duration between each repeat run is necessary, because the model's logic is dependent on the current time `_t`. However, in this gamma oscillation network, although we set the duration as `duration=500.` (not `duration=(1000., 1500.)`) in the third repeat run, the model will also produce the correct results. This is because the network running logic is independent with the current time `_t`.

CHAPTER

ELEVEN

TIPS ON JIT PROGRAMMING

BrainPy heavily relies on JIT compilation. Minimal knowledge about JIT programming will make you code much faster codes on BrainPy.

11.1 Global variable cannot be updated

JIT compilers treat global variables as compile-time constants, which means during the updating of step functions, global variables will not be updated. Anything you want to modify must be passed as an argument in the step functions. For example, if you define a 1D array, and at every call of the step function, some value want to be stored in the array:

```
import brainpy as bp
import numpy as np

array_to_store = np.zeros(100)

def update(_i_):
    array_to_store[_i_] = np.random.random()

neu = bp.NeuType('test', steps=update, ...)
```

The update function in defined `neu` will not work actually. `array_to_store` will not be modified. Instead, you can update `array_to_store` by passing it into the function as the argument:

```
def update(_i_, array_to_store):
    array_to_store[_i_] = np.random.random()

neu = bp.NeuType('test', steps=update, ...)
```

This will work.

11.2 Avoid containers

JIT compilers (like Numba, JAX) support containers, such as `dict`, `namedtuple`. However, the computation based in containers will greatly reduces the performance.

Continue . . . .

CHAPTER

TWELVE

HOW BRAINPY WORKS

12.1 Desgin philosophy

The goal of `BrainPy` is to provide a highly flexible and efficient neural simulator for Python users. Specifically, several principles are kept in mind during the development of BrainPy.

- **Easy to learn and use**. The aim of BrainPy is to accelerate your reaches on neuronal dynamics modeling. We don't want BrainPy make you pay much time on the learning of how to code. On the contrary, all you need is to focus on the implementation logic of the network model by using your familiar NumPy APIs. Although you've never used NumPy, even you are unfamiliar with Python, using BrainPy is also a easy thing. This is because the Python and NumPy syntax is simple, elegant and human-like.
- **Flexible and Transparent**. Another consideration of BrainPy is the flexibility. Traditional simulators with code generation approach (such as Brain2 and ANNarchy) have intrinsic limitations. In order to generate efficient low-level (such as c++) codes, these simulators make assumptions for models to simulate, and require users to provide string descriptions to define models. Such string descriptions greatly reduce the programming capacity. Moreover, there will always be exceptions beyond the framework assumptions, such as the data or logical flows that the framework do not consider before. Once such frameworks are not tailored to the user needs, extensions becomes difficult and even impossible. Furthermore, no framework is immune to errors when dealing with user's incredible models (even the well-tested framework TensorFlow). Therefore, making the framework transparent to users becomes indispensable. Considering this, BrainPy enables the users to directly modify the final formatted code once some errors are found (see examples comming soon). Actually, BrainPy endows the users with the fully data/logic flow control.

  It is concise and powerful, and there is no secrets for users.
- **Efficient**. The final consideration of BrainPy is to accelerate the running speed of of your coded models. In order to achieve high efficiency, we incorporate several Just-In-Time compilers (now support Numba, future will support JAX and others) into BrainPy. Moreover, an unified NumPy-like API are provided for these compilers. The aim of the API design is to let the user *code once and run everywhere* (the same code runs on CPU, multi-core, GPU, OpenCL, etc.).

12.2 BrainPy is more flexbile than what you think

Actually, the above illustration is just a tip of the iceberg. BrainPy is much more flebible than what you think. We will come back this section lator.

CHAPTER

THIRTEEN

USAGE OF CONNECT MODULE

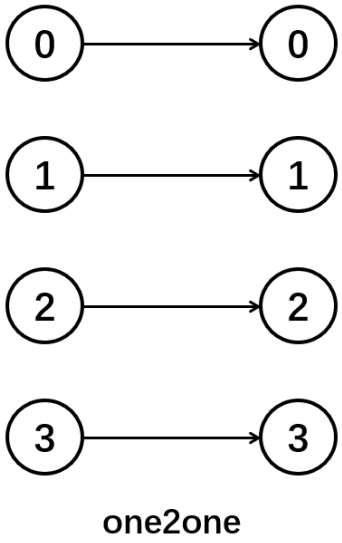
Here, BrainPy pre-defines several commonly used connection methods.

13.1 one-to-one connection

The neurons in the pre-synaptic neuron group only connect to the neurons in the same position of the post-synaptic group. Thus, this connection requires the indicesetry of two neuron groups same. Otherwise, an error will occurs.

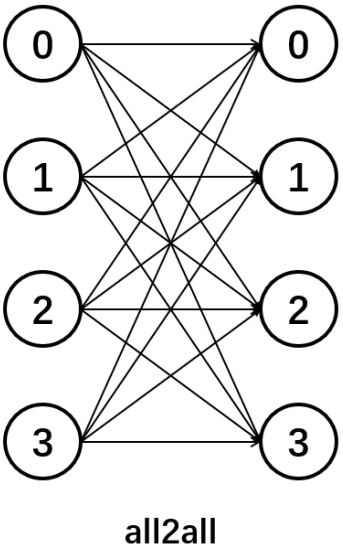
Usage of the method:

```
import brainpy as bp

pre_ids, post_ids = bp.connect.one2one(num_pre, num_post)
```



13.2 all-to-all connection

All neurons of the post-synaptic population form connections with all neurons of the pre-synaptic population (dense connectivity). Users can choose whether connect the neurons at the same position (*include_self=True or False*).

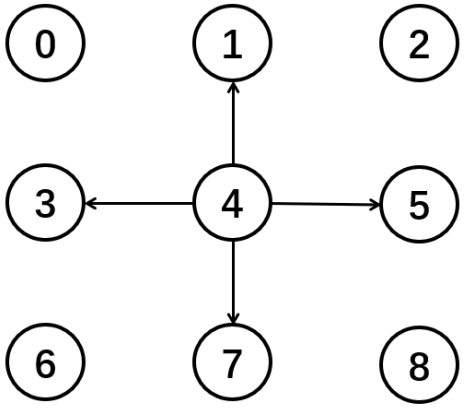


Usage of the method:

```
all2all = bp.connect.All2All(include_self=True)
pre_ids, post_ids = all2all(pre_indices, post_indices)
```

13.3 grid-four connection

grid-four connection is the four nearest neighbors connection. Each neuron connect to its nearest four neurons.

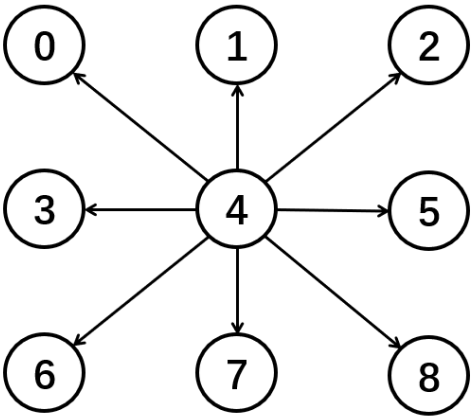


grid_four

```
grid_four = bp.connect.GridFour(include_self=True)
pre_ids, post_ids = grid_four(height_and_width)
```

13.4 grid-eight connection

grid-eight connection is eight nearest neighbors connection. Each neuron connect to its nearest eight neurons.

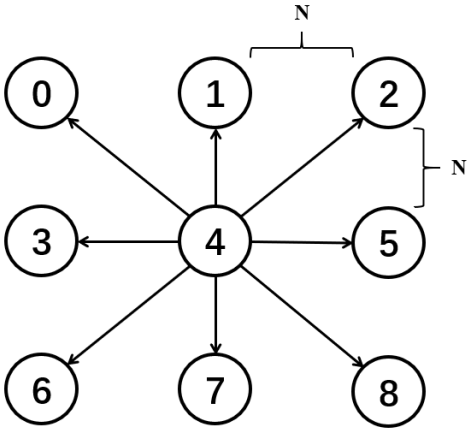


grid_eight

```
grid_eight = bp.connect.GridEight(include_self=False)
pre_ids, post_ids = grid_eight(height_and_width)
```

13.5 grid-N connection

grid-N connection is also a nearest neighbors connection. Each neuron connect to its nearest $2N \cdot 2N$ neurons.

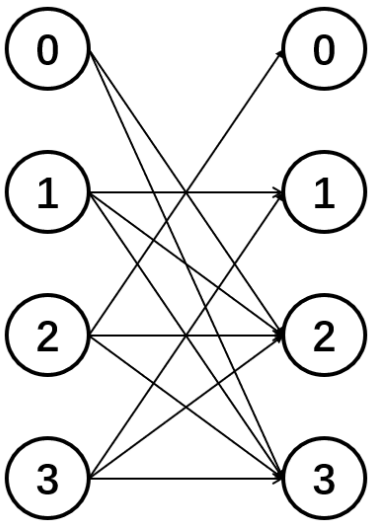


grid_N

```
grid_n = bp.connect.GridN(n=2, include_self=True)
pre_ids, post_ids = grid_n(height_and_width)
```

13.6 fixed_probability connection

For each post-synaptic neuron, there is a fixed probability that it forms a connection with a neuron of the pre-synaptic population. It is basically a all_to_all projection, except some synapses are not created, making the projection sparser.

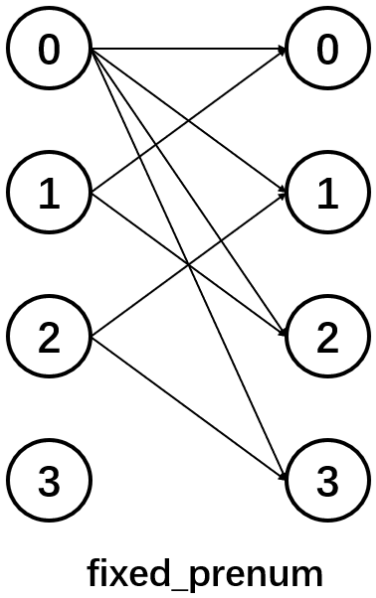


fixed_prob

```
fixed_prob = bp.connect.FixedProb(prob=0.1, include_self=True, seed=123)
pre_ids, post_ids = fixed_prob(pre_indices, post_indices)
```

13.7 fixed pre-synaptic number connection

Each neuron in the post-synaptic population receives connections from a fixed number of neurons of the pre-synaptic population chosen randomly. It may happen that two post-synaptic neurons are connected to the same pre-synaptic neuron and that some pre-synaptic neurons are connected to nothing.



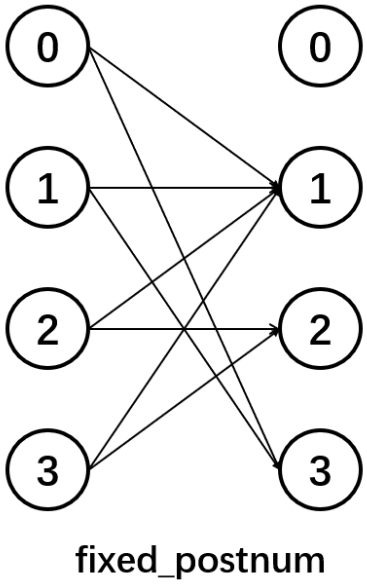
fixed_prenum

```
fixed_num = bp.connect.FixedPreNum(num=10, include_self=True, seed=123)
pre_ids, post_ids = fixed_num(pre_indices, post_indices)
```

13.8 fixed post-synaptic number connection

Each neuron in the pre-synaptic population sends a connection to a fixed number of neurons of the post-synaptic population chosen randomly. It may happen that two pre-synaptic neurons are connected to the same post-synaptic neuron and that some post-synaptic neurons receive no connection at all.

```
fixed_num = bp.connect.FixedPostNum(num=10, include_self=True, seed=123)
pre_ids, post_ids = fixed_num(pre_indices, post_indices)
```



13.9 gaussian probability connection

Builds a Gaussian connection pattern between the two populations, where the connection probability decay according to the gaussian function.

Specifically,

$$p = \exp(-\frac{(x-x_c)^2 + (y-y_c)^2}{2\sigma^2})$$

where (x, y) is the position of the pre-synaptic neuron and (x_c, y_c) is the position of the post-synaptic neuron.

For example, in a 30x30 two-dimensional networks, when $\beta = \frac{1}{2\sigma^2} = 0.1$, the connection pattern is shown as the follows:

```
gaussian_prob = bp.connect.GaussianProb(sigma=2.236, normalize=False,
                                        include_self=True, seed=123)
pre_ids, post_ids = gaussian_prob(pre_indices, post_indices)
```

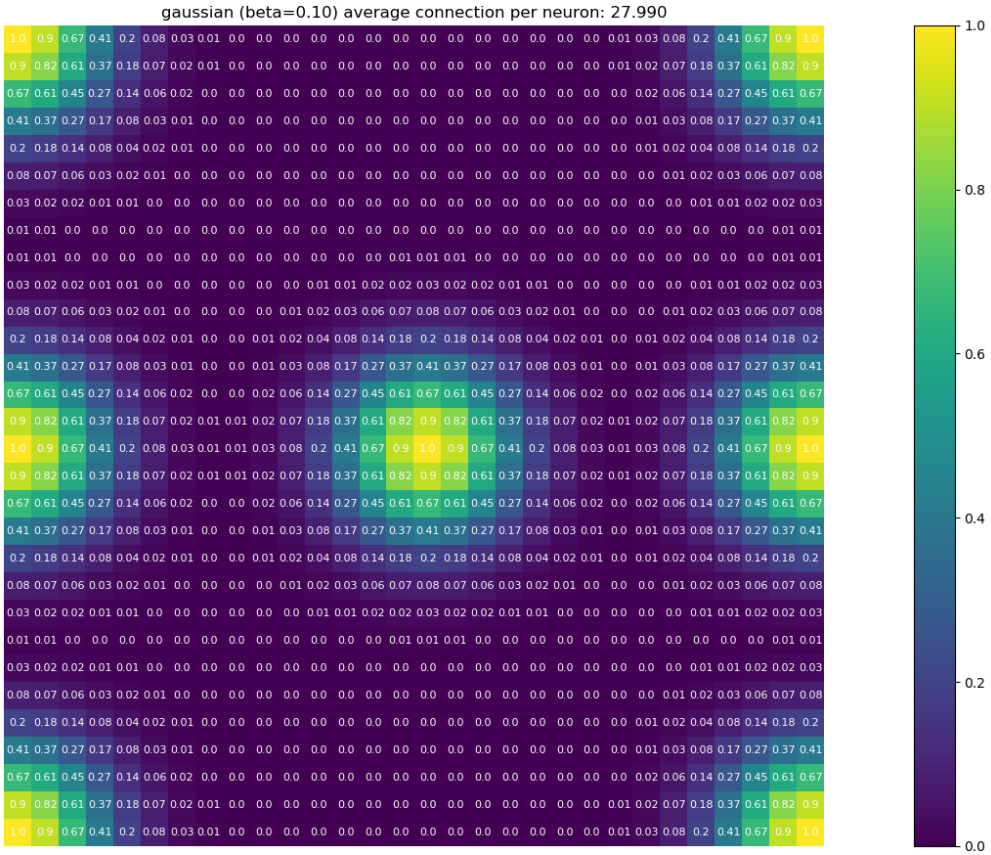
13.10 gaussian weight connection

Builds a Gaussian connection pattern between the two populations, where the weights decay with gaussian function.

Specifically,

$$w(x, y) = w_{max} \cdot \exp(-\frac{(x-x_c)^2 + (y-y_c)^2}{2\sigma^2})$$

where (x, y) is the position of the pre-synaptic neuron (normalized to [0,1]) and (x_c, y_c) is the position of the post-synaptic neuron (normalized to [0,1]), w_{max} is the maximum weight. In order to void creating useless synapses, w_{min} can be set to restrict the creation of synapses to the cases where the value of the weight would be superior to w_{min}. Default is $0.01 w_{max}$.



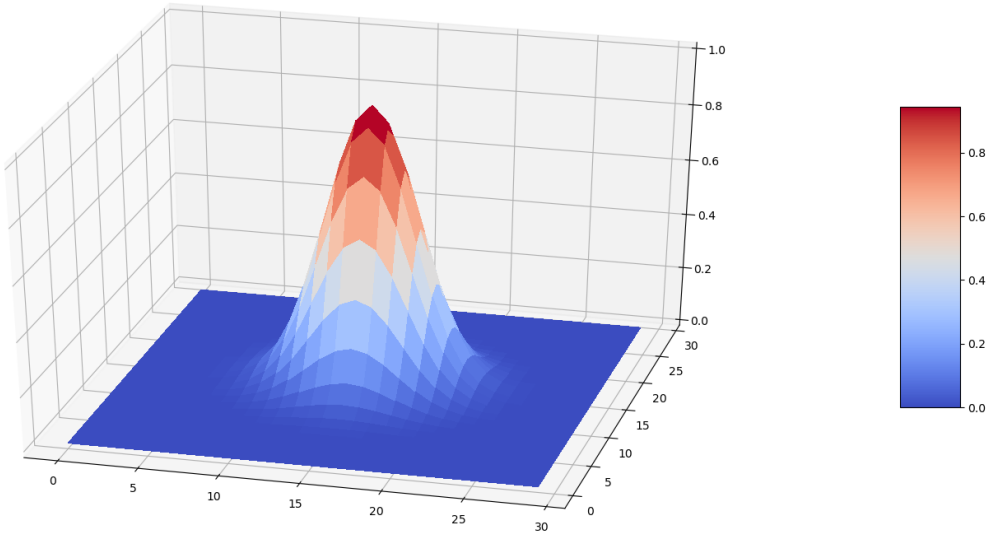

```
import numpy as np
import matplotlib.pyplot as plt

def show_weight(pre_ids, post_ids, weights, indicesetry, neu_id):
    height, width = indicesetry
    ids = np.where(pre_ids == neu_id)[0]
    post_ids = post_ids[ids]
    weights = weights[ids]

    X, Y = np.arange(height), np.arange(width)
    X, Y = np.meshgrid(X, Y)
    Z = np.zeros(indicesetry)
    for id_, weight in zip(post_ids, weights):
        h, w = id_ // width, id_ % width
        Z[h, w] = weight

    fig = plt.figure()
    ax = fig.gca(projection='3d')
    surf = ax.plot_surface(X, Y, Z, cmap=plt.cm.coolwarm, linewidth=0,␣
↪antialiased=False)
    fig.colorbar(surf, shrink=0.5, aspect=5)
    plt.show()
```

```
gaussian_weight = bp.connect.GaussianWeight(
                  sigma=0.1, w_max=1., w_min=0.,
                  normalize=True, include_self=True)
```

(continues on next page)

(continued from previous page)

```
pre_indices = post_indices = np.arange(30*30).reshape((30, 30))
pre_ids, post_ids, weights = gaussian_weight(pre_indices, post_indices)

show_weight(pre_ids, post_ids, weights, pre_indices, 465)
```



13.11 difference-of-gaussian (dog) connection

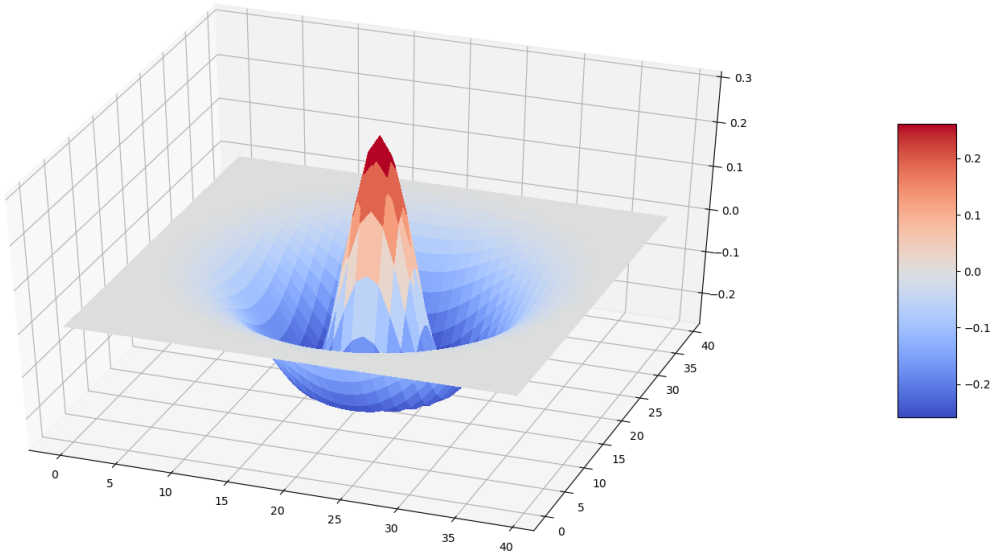
Builds a Difference-Of-Gaussian (dog) connection pattern between the two populations.

Mathematically,

$$w(x,y) = w^{+}_{max} \cdot \exp(-\frac{(x-x_c)^2+(y-y_c)^2}{2\sigma^2_{+}}) - w^{-}_{max} \cdot \exp(-\frac{(x-x_c)^2+(y-y_c)^2}{2\sigma^2_{-}})$$

where weights smaller than $0.01 * abs(w_{max} - w_{min})$ are not created and self-connections are avoided by default (parameter allow_self_connections).

```
dog = bp.connect.DOG(
         sigmas=[0.08, 0.15], ws_max=[1.0, 0.7], w_min=0.01,
         normalize=True, include_self=True)
pre_indices = post_indices = np.arange(40*40).reshape((40, 40))
pre_ids, post_ids, weights = dog(pre_indices, post_indices)

show_weight(pre_ids, post_ids, weights, pre_indices, 820)
```


0.3
0.2
0.1
0.0
−0.1
−0.2
0
5
10
15
20
25
30
35
40
0
5
10
15
20
25
30
35
40
0.2
0.1
0.0
−0.1
−0.2

CHAPTER

FOURTEEN

USAGE OF `inputs` MODULE

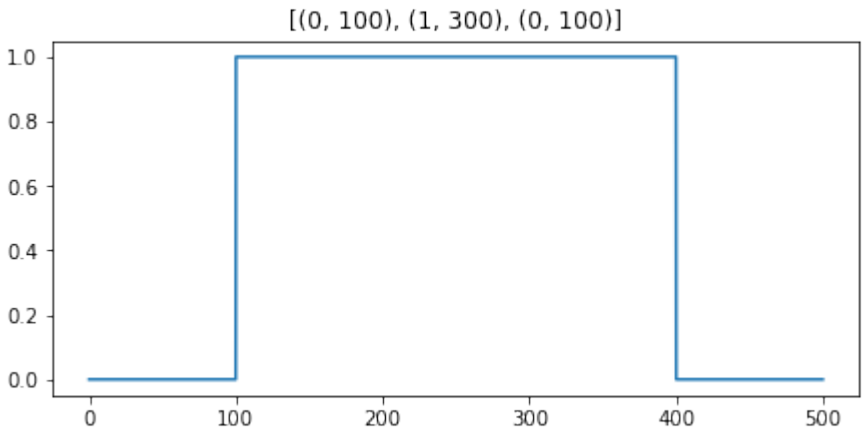
[1]:
```
import numpy as np
import matplotlib.pyplot as plt

import brainpy as bp
```

14.1 constant_current

`constant_current()` function helps you to format constant current in several periods.

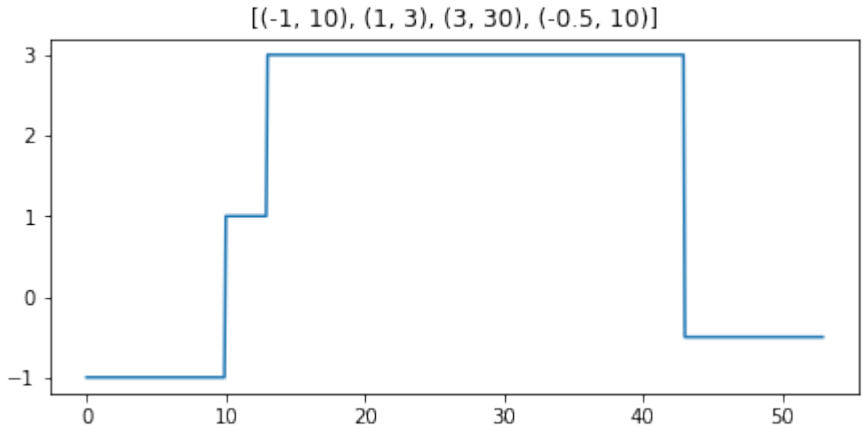
For example, if you want to get an input in which 0-100 ms is zero, 100-400 ms is value `1.`, and 400-500 ms is zero, then, you can define:

[2]:
```
current, duration = bp.inputs.constant_current([(0, 100), (1, 300), (0, 100)], 0.1)

fig, gs = bp.visualize.get_figure(1, 1)
fig.add_subplot(gs[0, 0])
ts = np.arange(0, duration, 0.1)
plt.plot(ts, current)
plt.title('[(0, 100), (1, 300), (0, 100)]')
plt.show()
```



Another example is this:

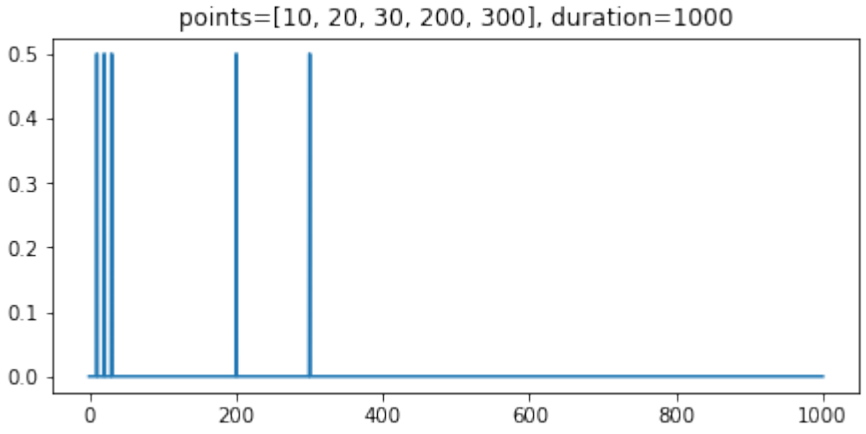
[3]:
```
current, duration = bp.inputs.constant_current([(-1, 10), (1, 3), (3, 30), (-0.5,␣
↪10)], 0.1)

fig, gs = bp.visualize.get_figure(1, 1)
fig.add_subplot(gs[0, 0])
ts = np.arange(0, duration, 0.1)
plt.plot(ts, current)
plt.title('[(-1, 10), (1, 3), (3, 30), (-0.5, 10)]')
plt.show()
```



14.2 spike_current

`spike_current()` function helps you to construct an input like a series of short-time spikes.

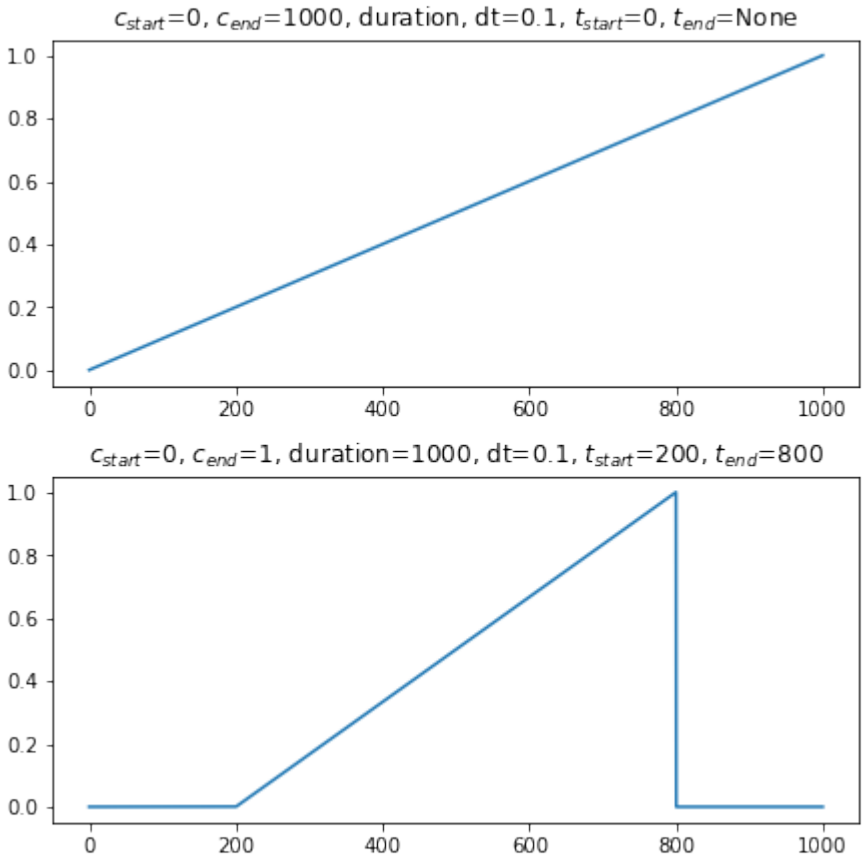
[4]:
```
points, length, size, duration, _dt = [10, 20, 30, 200, 300], 1., 0.5, 1000, 0.1
current = bp.inputs.spike_current(points, length, size, duration, _dt)

fig, gs = bp.visualize.get_figure(1, 1)
fig.add_subplot(gs[0, 0])
ts = np.arange(0, duration, _dt)
plt.plot(ts, current)
plt.title(r'points=%s, duration=%d' % (points, duration))
plt.show()
```



In the above example, at 10 ms, 20 ms, 30 ms, 200 ms, 300 ms, the assumed neuron produces spikes. Each spike lasts 1 ms, and the spike current is 0.5.

14.3 ramp_current

[5]:
```
fig, gs = bp.visualize.get_figure(2, 1)

duration, _dt = 1000, 0.1
current = bp.inputs.ramp_current(0, 1, duration)

ts = np.arange(0, duration, _dt)
fig.add_subplot(gs[0, 0])
plt.plot(ts, current)
plt.title(r'$c_{start}$=0, $c_{end}$=%d, duration, dt=%.1f, '
          r'$t_{start}$=0, $t_{end}$=None' % (duration, _dt,))

duration, _dt, t_start, t_end = 1000, 0.1, 200, 800
current = bp.inputs.ramp_current(0, 1, duration, t_start, t_end)

ts = np.arange(0, duration, _dt)
fig.add_subplot(gs[1, 0])
plt.plot(ts, current)
plt.title(r'$c_{start}$=0, $c_{end}$=1, duration=%d, dt=%.1f, '
          r'$t_{start}$=%d, $t_{end}$=%d' % (duration, _dt, t_start, t_end))

plt.show()
```


c_{start}=0, c_{end}=1000, duration, dt=0.1, t_{start}=0, t_{end}=None
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
c_{start}=0, c_{end}=1, duration=1000, dt=0.1, t_{start}=200, t_{end}=800
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000

CHAPTER

FIFTEEN

BRAINPY.PROFILE PACKAGE

The setting of the overall framework by `profile.py` API.

set([jit, device, numerical_method, dt, ...])	
run_on_cpu()	Check whether the device is "CPU".
run_on_gpu()	Check whether the device is "GPU".
set_device(jit[, device])	Set the backend and the device to deploy the models.
get_device()	Get the device name.
set_dt(dt)	Set the numerical integrator precision.
get_dt()	Get the numerical integrator precision.
set_numerical_method(method)	Set the default numerical integrator method for differential equations.
get_numerical_method()	Get the default numerical integrator method.
set_numba_profile(**kwargs)	Set the compilation options of Numba JIT function.
get_numba_profile()	Get the compilation setting of numba JIT function.
set_backend(backend)	Set the running backend.
get_backend()	Get the used backend of BrainPy.
get_num_thread_gpu()	
is_jit()	Check whether the backend is `numba`.
is_merge_integrators()	
is_merge_steps()	
is_substitute_equation()	
show_code_scope()	
show_format_code()	

15.1 brainpy.profile.set

`brainpy.profile.`**`set`**(*jit=None*, *device=None*, *numerical_method=None*, *dt=None*, *float_type=None*, *int_type=None*, *merge_integrators=None*, *merge_steps=None*, *substitute=None*, *show_code=None*, *show_code_scope=None*)

15.2 brainpy.profile.run_on_cpu

brainpy.profile.**run_on_cpu**()
Check whether the device is "CPU".

Returns **device** – True or False.

Return type bool

15.3 brainpy.profile.run_on_gpu

brainpy.profile.**run_on_gpu**()
Check whether the device is "GPU".

Returns **device** – True or False.

Return type bool

15.4 brainpy.profile.set_device

brainpy.profile.**set_device**(*jit*, *device=None*)
Set the backend and the device to deploy the models.

Parameters

- **jit** (*bool*) – Whether use the jit acceleration.
- **device** (*str, optional*) – The device name.

15.5 brainpy.profile.get_device

brainpy.profile.**get_device**()
Get the device name.

Returns **device** – Device name.

Return type str

15.6 brainpy.profile.set_dt

brainpy.profile.**set_dt**(*dt*)
Set the numerical integrator precision.

Parameters **dt** (*float*) – Numerical integration precision.

15.7 brainpy.profile.get_dt

brainpy.profile.**get_dt**()
Get the numerical integrator precision.

Returns **dt** – Numerical integration precision.

Return type float

15.8 brainpy.profile.set_numerical_method

brainpy.profile.**set_numerical_method**(*method*)
Set the default numerical integrator method for differential equations.

Parameters **method** (*str, callable*) – Numerical integrator method.

15.9 brainpy.profile.get_numerical_method

brainpy.profile.**get_numerical_method**()
Get the default numerical integrator method.

Returns **method** – The default numerical integrator method.

Return type str

15.10 brainpy.profile.set_numba_profile

brainpy.profile.**set_numba_profile**(***kwargs*)
Set the compilation options of Numba JIT function.

Parameters **kwargs** (*Any*) – The arguments, including cache, fastmath, parallel, nopython.

15.11 brainpy.profile.get_numba_profile

brainpy.profile.**get_numba_profile**()
Get the compilation setting of numba JIT function.

Returns **numba_setting** – Numba setting.

Return type dict

15.12 brainpy.profile.set_backend

brainpy.profile.**set_backend**(*backend*)
Set the running backend.

Parameters **backend** (*str*) – The backend name.

15.13 brainpy.profile.get_backend

brainpy.profile.**get_backend**()
Get the used backend of BrainPy.

Returns **backend** – The backend name.

Return type str

15.14 brainpy.profile.get_num_thread_gpu

brainpy.profile.**get_num_thread_gpu**()

15.15 brainpy.profile.is_jit

brainpy.profile.**is_jit**()
Check whether the backend is numba.

Returns **jit** – True or False.

Return type bool

15.16 brainpy.profile.is_merge_integrators

brainpy.profile.**is_merge_integrators**()

15.17 brainpy.profile.is_merge_steps

brainpy.profile.**is_merge_steps**()

15.18 brainpy.profile.is_substitute_equation

brainpy.profile.**is_substitute_equation**()

15.19 brainpy.profile.show_code_scope

brainpy.profile.**show_code_scope**()

15.20 brainpy.profile.show_format_code

brainpy.profile.**show_format_code**()

CHAPTER

SIXTEEN

BRAINPY.CORE PACKAGE

ObjType(ST, name, steps, List, Tuple], ...)	The base type of neuron and synapse.
NeuType(name, ST, steps, List, Tuple], mode, ...)	Abstract Neuron Type.
SynType(name, ST, steps, list, tuple], mode, ...)	Abstract Synapse Type.
Ensemble(name, num, model, monitors, ...)	Base Ensemble class.
NeuGroup(model, geometry, List, int], ...)	Neuron Group.
SynConn(model, pre_group, ...)	Synaptic connections.
Network(*args[, mode])	The main simulation controller in BrainPy.
ParsUpdate(all_pars, num, model)	Class for parameter updating.
delayed(func)	Decorator for synapse delay.

16.1 brainpy.core.ObjType

class brainpy.core.**ObjType**(*ST: brainpy.core.types.ObjState*, *name: str*, *steps: Union[Callable, List, Tuple]*, *requires: Optional[Dict] = None*, *mode: str = 'vector'*, *hand_overs: Optional[Dict] = None*, *heter_params_replace: Optional[Dict] = None*)

The base type of neuron and synapse.

Parameters **name** (*str, optional*) – Model name.

__init__(*ST: brainpy.core.types.ObjState*, *name: str*, *steps: Union[Callable, List, Tuple]*, *requires: Optional[Dict] = None*, *mode: str = 'vector'*, *hand_overs: Optional[Dict] = None*, *heter_params_replace: Optional[Dict] = None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(ST, name, steps[, requires, mode, ...])	Initialize self.

16.2 brainpy.core.NeuType

class brainpy.core.**NeuType**(*name: str*, *ST: brainpy.core.types.NeuState*, *steps: Union[Callable, List, Tuple]*, *mode: str = 'vector'*, *requires: Optional[dict] = None*, *hand_overs: Optional[Dict] = None*, *heter_params_replace: Optional[Dict] = None*)

Abstract Neuron Type.

It can be defined based on a group of neurons or a single neuron.

__init__(*name: str*, *ST: brainpy.core.types.NeuState*, *steps: Union[Callable, List, Tuple]*, *mode: str = 'vector'*, *requires: Optional[dict] = None*, *hand_overs: Optional[Dict] = None*, *heter_params_replace: Optional[Dict] = None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(name, ST, steps[, mode, requires, ...])	Initialize self.

16.3 brainpy.core.SynType

class brainpy.core.**SynType**(*name: str*, *ST: brainpy.core.types.SynState*, *steps: Union[callable, list, tuple]*, *mode: str = 'vector'*, *requires: Optional[dict] = None*, *hand_overs: Optional[Dict] = None*, *heter_params_replace: Optional[dict] = None*)

Abstract Synapse Type.

It can be defined based on a collection of synapses or a single synapse model.

__init__(*name: str*, *ST: brainpy.core.types.SynState*, *steps: Union[callable, list, tuple]*, *mode: str = 'vector'*, *requires: Optional[dict] = None*, *hand_overs: Optional[Dict] = None*, *heter_params_replace: Optional[dict] = None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(name, ST, steps[, mode, requires, ...])	Initialize self.

16.4 brainpy.core.Ensemble

class brainpy.core.**Ensemble**(*name: str*, *num: int*, *model: brainpy.core.base.ObjType*, *monitors: Tuple*, *pars_update: Dict*, *cls_type: str*, *satisfies: Optional[dict] = None*)

Base Ensemble class.

Parameters

- **name** (*str*) – Name of the (neurons/synapses) ensemble.
- **num** (*int*) – The number of the neurons/synapses.

- **model** (ObjType) – The (neuron/synapse) model.
- **monitors** (*list, tuple, None*) – Variables to monitor.
- **pars_update** (*dict, None*) – Parameters to update.
- **cls_type** (*str*) – Class type.

__init__(*name: str*, *num: int*, *model: brainpy.core.base.ObjType*, *monitors: Tuple*, *pars_update: Dict*, *cls_type: str*, *satisfies: Optional[dict] = None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(name, num, model, monitors, ... [, ...])	Initialize self.
get_schedule()	
run(duration[, inputs, report, report_percent])	
set_schedule(schedule)	
type_checking()	

Attributes

requires

16.5 brainpy.core.NeuGroup

class brainpy.core.**NeuGroup**(*model: brainpy.core.neurons.NeuType*, *geometry: Union[Tuple, List, int]*, *monitors: Optional[Union[List, Tuple]] = None*, *name: Optional[str] = None*, *satisfies: Optional[Dict] = None*, *pars_update: Optional[Dict] = None*)
Neuron Group.

Parameters

- **model** (NeuType) – The instantiated neuron type model.
- **geometry** (*int, tuple*) – The neuron group geometry.
- **pars_update** (*dict, None*) – Parameters to update.
- **monitors** (*list, tuple, None*) – Variables to monitor.
- **name** (*str, None*) – The name of the neuron group.

__init__(*model: brainpy.core.neurons.NeuType*, *geometry: Union[Tuple, List, int]*, *monitors: Optional[Union[List, Tuple]] = None*, *name: Optional[str] = None*, *satisfies: Optional[Dict] = None*, *pars_update: Optional[Dict] = None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(model, geometry[, monitors, name, ...])	Initialize self.
`get_schedule`()	
`run`(duration[, inputs, report, report_percent])	
`set_schedule`(schedule)	
`type_checking`()	

Attributes

`requires`

16.6 brainpy.core.SynConn

class brainpy.core.**SynConn**(*model: brainpy.core.synapses.SynType*, *pre_group: Optional[Union[brainpy.core.neurons.NeuGroup, brainpy.core.neurons.NeuSubGroup]] = None*, *post_group: Optional[Union[brainpy.core.neurons.NeuGroup, brainpy.core.neurons.NeuSubGroup]] = None*, *conn: Optional[Union[brainpy.connectivity.base.Connector, numpy.ndarray, Dict]] = None*, *delay: float = 0.0*, *name: Optional[str] = None*, *monitors: Optional[Union[List, Tuple]] = None*, *satisfies: Optional[Dict] = None*, *pars_update: Optional[Dict] = None*)

Synaptic connections.

Parameters

- **model** (SynType) – The instantiated neuron type model.
- **pars_update** (*dict, None*) – Parameters to update.
- **pre_group** (NeuGroup, *None*) – Pre-synaptic neuron group.
- **post_group** (NeuGroup, *None*) – Post-synaptic neuron group.
- **conn** (Connector, *None*) – Connection method to create synaptic connectivity.
- **num** (*int*) – The number of the synapses.
- **delay** (*float*) – The time of the synaptic delay.
- **monitors** (*list, tuple, None*) – Variables to monitor.
- **name** (*str, None*) – The name of the neuron group.

__init__(*model: brainpy.core.synapses.SynType*, *pre_group: Optional[Union[brainpy.core.neurons.NeuGroup, brainpy.core.neurons.NeuSubGroup]] = None*, *post_group: Optional[Union[brainpy.core.neurons.NeuGroup, brainpy.core.neurons.NeuSubGroup]] = None*, *conn: Optional[Union[brainpy.connectivity.base.Connector, numpy.ndarray, Dict]] = None*, *delay: float = 0.0*, *name: Optional[str] = None*, *monitors: Optional[Union[List, Tuple]] = None*, *satisfies: Optional[Dict] = None*, *pars_update: Optional[Dict] = None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(model[, pre_group, post_group, ...])	Initialize self.
get_schedule()	
run(duration[, inputs, report, report_percent])	
set_schedule(schedule)	
type_checking()	

Attributes

requires

16.7 brainpy.core.Network

class brainpy.core.**Network**(**args*, *mode='once'*, ***kwargs*)

The main simulation controller in BrainPy.

Network handles the running of a simulation. It contains a set of objects that are added with *add()*. The *run()* method actually runs the simulation. The main loop runs according to user add orders. The objects in the *Network* are accessible via their names, e.g. *net.name* would return the *object* (including neurons and synapses).

__init__(**args*, *mode='once'*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(*args[, mode])	Initialize self.
add(*args, **kwargs)	Add object (neurons or synapses or monitor) to the network.
build(run_length[, inputs])	
run(duration[, inputs, report, report_percent])	Run the simulation for the given duration.

Attributes

dt	
ts	Get the time points of the network.

16.8 brainpy.core.ParsUpdate

class brainpy.core.**ParsUpdate**(*all_pars*, *num*, *model*)

Class for parameter updating.

Structure of `ParsUpdate`

- origins : original parameters
- num : number of the neurons
- updates : parameters to update
- heters : parameters to update, and they are heterogeneous
- model : the model which this ParsUpdate belongs to

__init__(*all_pars*, *num*, *model*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(all_pars, num, model)	Initialize self.
clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(item)	Get the parameter value by its key.
items()	All parameters, including keys and values.
keys()	All parameters can be updated.
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	2-tuple; but raise KeyError if D is empty.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

all
heters
model
num
origins
updates

16.9 brainpy.core.delayed

brainpy.core.**delayed**(*func*)

Decorator for synapse delay.

Parameters **func** (*callable*) – The step function which use delayed synapse state.

Returns **func** – The modified step function.

Return type callable

class brainpy.core.**ObjType**(*ST: brainpy.core.types.ObjState*, *name: str*, *steps: Union[Callable, List, Tuple]*, *requires: Optional[Dict] = None*, *mode: str = 'vector'*, *hand_overs: Optional[Dict] = None*, *heter_params_replace: Optional[Dict] = None*)

The base type of neuron and synapse.

Parameters **name** (*str, optional*) – Model name.

class brainpy.core.**NeuType**(*name: str*, *ST: brainpy.core.types.NeuState*, *steps: Union[Callable, List, Tuple]*, *mode: str = 'vector'*, *requires: Optional[dict] = None*, *hand_overs: Optional[Dict] = None*, *heter_params_replace: Optional[Dict] = None*)

Abstract Neuron Type.

It can be defined based on a group of neurons or a single neuron.

class brainpy.core.**SynType**(*name: str*, *ST: brainpy.core.types.SynState*, *steps: Union[callable, list, tuple]*, *mode: str = 'vector'*, *requires: Optional[dict] = None*, *hand_overs: Optional[Dict] = None*, *heter_params_replace: Optional[dict] = None*)

Abstract Synapse Type.

It can be defined based on a collection of synapses or a single synapse model.

class brainpy.core.**Ensemble**(*name: str*, *num: int*, *model: brainpy.core.base.ObjType*, *monitors: Tuple*, *pars_update: Dict*, *cls_type: str*, *satisfies: Optional[dict] = None*)

Base Ensemble class.

Parameters

- **name** (*str*) – Name of the (neurons/synapses) ensemble.
- **num** (*int*) – The number of the neurons/synapses.
- **model** (ObjType) – The (neuron/synapse) model.
- **monitors** (*list, tuple, None*) – Variables to monitor.
- **pars_update** (*dict, None*) – Parameters to update.
- **cls_type** (*str*) – Class type.

class brainpy.core.**NeuGroup**(*model: brainpy.core.neurons.NeuType*, *geometry: Union[Tuple, List, int]*, *monitors: Optional[Union[List, Tuple]] = None*, *name: Optional[str] = None*, *satisfies: Optional[Dict] = None*, *pars_update: Optional[Dict] = None*)

Neuron Group.

Parameters

- **model** (NeuType) – The instantiated neuron type model.
- **geometry** (*int, tuple*) – The neuron group geometry.

- **pars_update** (*dict, None*) – Parameters to update.
- **monitors** (*list, tuple, None*) – Variables to monitor.
- **name** (*str, None*) – The name of the neuron group.

class brainpy.core.**SynConn**(*model: brainpy.core.synapses.SynType, pre_group: Optional[Union[brainpy.core.neurons.NeuGroup, brainpy.core.neurons.NeuSubGroup]] = None, post_group: Optional[Union[brainpy.core.neurons.NeuGroup, brainpy.core.neurons.NeuSubGroup]] = None, conn: Optional[Union[brainpy.connectivity.base.Connector, numpy.ndarray, Dict]] = None, delay: float = 0.0, name: Optional[str] = None, monitors: Optional[Union[List, Tuple]] = None, satisfies: Optional[Dict] = None, pars_update: Optional[Dict] = None*)

Synaptic connections.

Parameters

- **model** (SynType) – The instantiated neuron type model.
- **pars_update** (*dict, None*) – Parameters to update.
- **pre_group** (NeuGroup, *None*) – Pre-synaptic neuron group.
- **post_group** (NeuGroup, *None*) – Post-synaptic neuron group.
- **conn** (Connector, *None*) – Connection method to create synaptic connectivity.
- **num** (*int*) – The number of the synapses.
- **delay** (*float*) – The time of the synaptic delay.
- **monitors** (*list, tuple, None*) – Variables to monitor.
- **name** (*str, None*) – The name of the neuron group.

class brainpy.core.**Network**(**args*, *mode='once'*, ***kwargs*)

The main simulation controller in BrainPy.

Network handles the running of a simulation. It contains a set of objects that are added with *add()*. The *run()* method actually runs the simulation. The main loop runs according to user add orders. The objects in the *Network* are accessible via their names, e.g. *net.name* would return the *object* (including neurons and synapses).

add(**args*, ***kwargs*)

Add object (neurons or synapses or monitor) to the network.

Parameters

- **args** – The nameless objects.
- **kwargs** – The named objects, which can be accessed by *net.xxx* (xxx is the name of the object).

run(*duration*, *inputs=()*, *report=False*, *report_percent=0.1*)

Run the simulation for the given duration.

This function provides the most convenient way to run the network. For example:

Parameters

- **duration** (*int, float, tuple, list*) – The amount of simulation time to run for.
- **inputs** (*list, tuple*) – The receivers, external inputs and durations.

- **report** (*bool*) – Report the progress of the simulation.
- **report_percent** (*float*) – The speed to report simulation progress.

class brainpy.core.**ParsUpdate**(*all_pars*, *num*, *model*)

Class for parameter updating.

Structure of `ParsUpdate`

- origins : original parameters
- num : number of the neurons
- updates : parameters to update
- heters : parameters to update, and they are heterogeneous
- model : the model which this ParsUpdate belongs to

get(*item*)

Get the parameter value by its key.

Parameters **item** (*str*) – Parameter name.

Returns **value** – Parameter value.

Return type any

items()

All parameters, including keys and values.

Returns **items** – The iterable parameter items.

Return type iterable

keys()

All parameters can be updated.

Returns **keys** – List of parameter names.

Return type list

CHAPTER

SEVENTEEN

BRAINPY.INTEGRATION PACKAGE

17.1 diff_equation module

Expression(var, code)	
DiffEquation(func)	Differential Equation.

17.1.1 brainpy.integration.Expression

class brainpy.integration.**Expression**(*var*, *code*)

__init__(*var*, *code*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(var, code)	Initialize self.
get_code([subs])	

Attributes

identifiers

17.1.2 brainpy.integration.DiffEquation

class brainpy.integration.**DiffEquation**(*func*)
Differential Equation.

A differential equation is defined as the standard form:

dx/dt = f(x) + g(x) dW

Parameters **func** (*callable*) – The user defined differential equation.

__init__(*func*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(func)	Initialize self.
get_f_expressions([substitute_vars])	
get_g_expressions()	
replace_f_expressions(name, y_sub[, t_sub])	Replace expressions of df part.
replace_g_expressions(name, y_sub[, t_sub])	

Attributes

expr_names
is_functional_noise
is_multi_return
is_stochastic
stochastic_type

class brainpy.integration.**DiffEquation**(*func*)
Differential Equation.

A differential equation is defined as the standard form:

dx/dt = f(x) + g(x) dW

Parameters **func** (*callable*) – The user defined differential equation.

replace_f_expressions(*name*, *y_sub*, *t_sub=None*)
Replace expressions of df part.

Parameters

- **name** (*str*) – The name of the new expression.
- **y_sub** (*str*) – The new name of the variable “y”.
- **t_sub** (*str, optional*) – The new name of the variable “t”.

Returns **list_of_expr** – A list of expressions.

Return type list

17.2 integrator module

integrate([func, method])	Generate the one-step integrator function for differential equations.
get_integrator(method)	
Integrator(diff_eq)	
Euler(diff_eq)	Forward Euler method.
Heun(diff_eq)	Two-stage method for numerical integrator.
MidPoint(diff_eq)	Explicit midpoint Euler method.
RK2(diff_eq[, beta])	Parametric second-order Runge-Kutta (RK2).
RK3(diff_eq)	Kutta's third-order method (commonly known as RK3).
RK4(diff_eq)	Fourth-order Runge-Kutta (RK4)[9][10][11] .
RK4Alternative(diff_eq)	An alternative of fourth-order Runge-Kutta method.
ExponentialEuler(diff_eq)	First order, explicit exponential Euler method.
MilsteinIto(diff_eq)	Itô stochastic integral.
MilsteinStra(diff_eq)	Heun two-stage stochastic numerical method for Stratonovich integral.

17.2.1 brainpy.integration.integrate

brainpy.integration.**integrate**(*func=None*, *method=None*)

Generate the one-step integrator function for differential equations.

Using this method, the users only need to define the right side of the equation. For example, for the *m* channel in the Hodgkin–Huxley neuron model

$$\alpha = \frac{0.1 * (V + 40}{1 - \exp(-(V + 40)/10)}$$
$$\beta = 4.0 * \exp(-(V + 65)/18)$$
$$\frac{dm}{dt} = \alpha * (1 - m) - \beta * m$$

Using BrainPy, this ODE function can be written as

```
>>> import numpy as np
>>> from brainpy import integrate
>>>
>>> @integrate(method='rk4')
>>> def int_m(m, t, V):
>>>     alpha = 0.1 * (V + 40) / (1 - np.exp(-(V + 40) / 10))
>>>     beta = 4.0 * np.exp(-(V + 65) / 18)
>>>     return alpha * (1 - m) - beta * m
```

Parameters

- **func** (*callable*) – The function at the right hand of the differential equation. If a stochastic equation (SDE) is defined, then *func* is the drift coefficient (the deterministic part) of the SDE.
- **method** (*None, str, callable*) – The method of numerical integrator.

[9] http://mathworld.wolfram.com/Runge-KuttaMethod.html
[10] https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods
[11] https://zh.wikipedia.org/wiki/

Returns **integrator** – If f is provided, then the one-step numerical integrator will be returned. if not, the wrapper will be provided.

Return type *Integrator*

17.2.2 brainpy.integration.get_integrator

brainpy.integration.**get_integrator**(*method*)

17.2.3 brainpy.integration.Integrator

class brainpy.integration.**Integrator**(*diff_eq*)

__init__(*diff_eq*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(diff_eq)	Initialize self.
get_nb_step(diff_eq, *args)	

Attributes

code_scope
py_func_name
update_code
update_func

17.2.4 brainpy.integration.Euler

class brainpy.integration.**Euler**(*diff_eq*)
Forward Euler method. Also named as explicit_Euler.

The simplest way for solving ordinary differential equations is "the Euler method" by Press et al. (1992)[1] :

$$y_{n+1} = y_n + f(y_n, t_n)\Delta t$$

This formula advances a solution from y_n to $y_{n+1} = y_n + h$. Note that the method increments a solution through an interval h while using derivative information from only the beginning of the interval. As a result, the step's error is $O(h^2)$.

For SDE equations, this approximation is a continuous time stochastic process that satisfy the iterative scheme[1].

$$Y_{n+1} = Y_n + f(Y_n)h_n + g(Y_n)\Delta W_n$$

where $n = 0, 1, \cdots, N-1$, $Y_0 = x_0$, $Y_n = Y(t_n)$, $h_n = t_{n+1} - t_n$ is the step size, $\Delta W_n = [W(t_{n+1}) - W(t_n)] \sim N(0, h_n) = \sqrt{h}N(0, 1)$ with $W(t_0) = 0$.

[1] W. H.; Flannery, B. P.; Teukolsky, S. A.; and Vetterling, W. T. Numerical Recipes in FORTRAN: The Art of Scientific Computing, 2nd ed. Cambridge, England: Cambridge University Press, p. 710, 1992.

For simplicity, we rewrite the above equation into

$$Y_{n+1} = Y_n + f_n h + g_n \Delta W_n$$

As the order of convergence for the Euler-Maruyama method is low (strong order of convergence 0.5, weak order of convergence 1), the numerical results are inaccurate unless a small step size is used. By adding one more term from the stochastic Taylor expansion, one obtains a 1.0 strong order of convergence scheme known as *Milstein scheme*[2].

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

References

__init__(*diff_eq*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(diff_eq)	Initialize self.
get_nb_step(diff_eq, *args)	

Attributes

code_scope
py_func_name
update_code
update_func

17.2.5 brainpy.integration.Heun

class brainpy.integration.**Heun**(*diff_eq*)
Two-stage method for numerical integrator.

For ODE, please see "RK2".

For stochastic Stratonovich integral, the Heun algorithm is given by, according to paper[4][5].

$$Y_{n+1} = Y_n + f_n h + \frac{1}{2}[g_n + g(\overline{Y}_n)]\Delta W_n$$
$$\overline{Y}_n = Y_n + g_n \Delta W_n$$

[2] U. Picchini, Sde toolbox: Simulation and estimation of stochastic differential equations with matlab.
[4] H. Gilsing and T. Shardlow, SDELab: A package for solving stochastic differential equations in MATLAB, Journal of Computational and Applied Mathematics 205 (2007), no. 2, 1002-1018.
[5] P.reversal_potential. Kloeden, reversal_potential. Platen, and H. Schurz, Numerical solution of SDE through computer experiments, Springer, 1994.

Or, it is written as

$$Y_1 = y_n + f(y_n)h + g_n \Delta W_n$$
$$y_{n+1} = y_n + \frac{1}{2}[f(y_n) + f(Y_1)]h + \frac{1}{2}[g(y_n) + g(Y_1)]\Delta W_n$$

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

References

See also:

RK2, *MidPoint*, *MilsteinStra*

__init__ (*diff_eq*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(diff_eq)	Initialize self.
get_nb_step(diff_eq, *args)	

Attributes

code_scope
py_func_name
update_code
update_func

17.2.6 brainpy.integration.MidPoint

class brainpy.integration.**MidPoint** (*diff_eq*)
Explicit midpoint Euler method. Also named as modified_Euler.

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

See also:

RK2, *Heun*

__init__ (*diff_eq*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(diff_eq)	Initialize self.
`get_nb_step`(diff_eq, *args)	

Attributes

`code_scope`
`py_func_name`
`update_code`
`update_func`

17.2.7 brainpy.integration.RK2

class `brainpy.integration.`**RK2**(*diff_eq*, *beta=0.6666666666666666*)

Parametric second-order Runge-Kutta (RK2). Also named as `RK2`.

It is given in parametric form by[3] .

$$
\begin{aligned}
k_1 &= f(y_n, t_n) \\
k_2 &= f(y_n + \beta \Delta t k_1, t_n + \beta \Delta t) \\
y_{n+1} &= y_n + \Delta t[(1 - \frac{1}{2\beta})k_1 + \frac{1}{2\beta}k_2]
\end{aligned}
$$

Parameters

- **diff_eq** (`DiffEquation`) – The differential equation.
- **beta** (*float*) – Popular choices for 'beta': 1/2 : explicit midpoint method 2/3 : Ralston's method 1 : Heun's method, also known as the explicit trapezoid rule

Returns **func** – The one-step numerical integrator function.

Return type callable

References

See also:

Heun, *MidPoint*

__init__(*diff_eq*, *beta=0.6666666666666666*)

Initialize self. See help(type(self)) for accurate signature.

[3] https://lpsa.swarthmore.edu/NumInt/NumIntSecond.html

Methods

`__init__`(diff_eq[, beta])	Initialize self.
`get_nb_step`(diff_eq[, beta])	

Attributes

`code_scope`
`py_func_name`
`update_code`
`update_func`

17.2.8 brainpy.integration.RK3

class brainpy.integration.**RK3**(*diff_eq*)

Kutta's third-order method (commonly known as RK3). Also named as `RK3`[678] .

$$
\begin{aligned}
k_1 &= f(y_n, t_n) \\
k_2 &= f(y_n + \frac{\Delta t}{2} k_1, tn + \frac{\Delta t}{2}) \\
k_3 &= f(y_n - \Delta t k_1 + 2\Delta t k_2, t_n + \Delta t) \\
y_{n+1} &= y_n + \frac{\Delta t}{6}(k_1 + 4k_2 + k_3)
\end{aligned}
$$

Parameters **diff_eq** (`DiffEquation`) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

References

__init__(*diff_eq*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(diff_eq)	Initialize self.
`get_nb_step`(diff_eq, *args)	

[6] http://mathworld.wolfram.com/Runge-KuttaMethod.html
[7] https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods
[8] https://zh.wikipedia.org/wiki/

Attributes

code_scope
py_func_name
update_code
update_func

17.2.9 brainpy.integration.RK4

class brainpy.integration.**RK4**(*diff_eq*)

Fourth-order Runge-Kutta (RK4)[9][10][11] .

$$
\begin{aligned}
k_1 &= f(y_n, t_n) \\
k_2 &= f(y_n + \frac{\Delta t}{2} k_1, t_n + \frac{\Delta t}{2}) \\
k_3 &= f(y_n + \frac{\Delta t}{2} k_2, t_n + \frac{\Delta t}{2}) \\
k_4 &= f(y_n + \Delta t k_3, t_n + \Delta t) \\
y_{n+1} &= y_n + \frac{\Delta t}{6}(k_1 + 2 * k_2 + 2 * k_3 + k_4)
\end{aligned}
$$

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

References

__init__(*diff_eq*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(diff_eq)	Initialize self.
get_nb_step(diff_eq, *args)	

[9] http://mathworld.wolfram.com/Runge-KuttaMethod.html
[10] https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods
[11] https://zh.wikipedia.org/wiki/

Attributes

code_scope
py_func_name
update_code
update_func

17.2.10 brainpy.integration.RK4Alternative

class brainpy.integration.**RK4Alternative**(*diff_eq*)

An alternative of fourth-order Runge-Kutta method. Also named as RK4_alternative ("3/8" rule).

It is a less often used fourth-order explicit RK method, and was also proposed by Kutta[12]:

$$
\begin{aligned}
k_1 &= f(y_n, t_n) \\
k_2 &= f(y_n + \frac{\Delta t}{3} k_1, t_n + \frac{\Delta t}{3}) \\
k_3 &= f(y_n - \frac{\Delta t}{3} k_1 + \Delta t k_2, t_n + \frac{2\Delta t}{3}) \\
k_4 &= f(y_n + \Delta t k_1 - \Delta t k_2 + \Delta t k_3, t_n + \Delta t) \\
y_{n+1} &= y_n + \frac{\Delta t}{8}(k_1 + 3 * k_2 + 3 * k_3 + k_4)
\end{aligned}
$$

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

References

__init__(*diff_eq*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(diff_eq)	Initialize self.
get_nb_step(diff_eq, *args)	

[12] https://en.wikipedia.org/wiki/List_of_Runge%E2%80%93Kutta_methods

Attributes

code_scope
py_func_name
update_code
update_func

17.2.11 brainpy.integration.ExponentialEuler

class brainpy.integration.**ExponentialEuler**(*diff_eq*)

First order, explicit exponential Euler method.

For an ODE equation of the form

$$y' = f(y), \quad y(0) = y_0$$

its schema is given by

$$y_{n+1} = y_n + h\varphi(hA)f(y_n)$$

where $A = f'(y_n)$ and $\varphi(z) = \frac{e^z - 1}{z}$.

For linear ODE system: $y' = Ay + B$, the above equation is equal to

$$y_{n+1} = y_n e^{hA} - B/A(1 - e^{hA})$$

For a SDE equation of the form

$$dy = (Ay + F(y))dt + g(y)dW(t) = f(y)dt + g(y)dW(t), \quad y(0) = y_0$$

its schema is given by[16]

$$\begin{aligned} y_{n+1} &= e^{\Delta t A}(y_n + g(y_n)\Delta W_n) + \varphi(\Delta t A)F(y_n)\Delta t \\ &= y_n + \Delta t\varphi(\Delta t A)f(y) + e^{\Delta t A}g(y_n)\Delta W_n \end{aligned}$$

where $\varphi(z) = \frac{e^z - 1}{z}$.

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

References

__init__(*diff_eq*)

Initialize self. See help(type(self)) for accurate signature.

[16] Erdoğan, Utku, and Gabriel J. Lord. "A new class of exponential integrators for stochastic differential equations with multiplicative noise." arXiv preprint arXiv:1608.07096 (2016).

Methods

`__init__`(diff_eq)	Initialize self.
`get_nb_step`(diff_eq, *args)	

Attributes

`code_scope`
`py_func_name`
`update_code`
`update_func`

17.2.12 brainpy.integration.MilsteinIto

class brainpy.integration.**MilsteinIto**(*diff_eq*)

Itô stochastic integral. The derivative-free Milstein method is an order 1.0 strong Taylor schema.

The following implementation approximates this derivative thanks to a Runge-Kutta approach[13].

In Itô scheme, it is expressed as

$$Y_{n+1} = Y_n + f_n h + g_n \Delta W_n + \frac{1}{2\sqrt{h}}[g(\overline{Y_n}) - g_n][(\Delta W_n)^2 - h]$$

where $\overline{Y_n} = Y_n + f_n h + g_n \sqrt{h}$.

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

References

__init__(*diff_eq*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(diff_eq)	Initialize self.
`get_nb_step`(diff_eq, *args)	

[13] P.reversal_potential. Kloeden, reversal_potential. Platen, and H. Schurz, Numerical solution of SDE through computer experiments, Springer, 1994.

Attributes

code_scope
py_func_name
update_code
update_func

17.2.13 brainpy.integration.MilsteinStra

class brainpy.integration.**MilsteinStra**(*diff_eq*)

Heun two-stage stochastic numerical method for Stratonovich integral.

Use the Stratonovich Heun algorithm to integrate Stratonovich equation, according to paper[14][15].

$$Y_{n+1} = Y_n + f_n h + \frac{1}{2}[g_n + g(\overline{Y}_n)]\Delta W_n$$
$$\overline{Y}_n = Y_n + g_n \Delta W_n$$

Or, it is written as

$$Y_1 = y_n + f(y_n)h + g_n \Delta W_n$$
$$y_{n+1} = y_n + \frac{1}{2}[f(y_n) + f(Y_1)]h + \frac{1}{2}[g(y_n) + g(Y_1)]\Delta W_n$$

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

References

See also:

MilsteinIto

__init__(*diff_eq*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(diff_eq)	Initialize self.
get_nb_step(diff_eq, *args)	

[14] H. Gilsing and T. Shardlow, SDELab: A package for solving stochastic differential equations in MATLAB, Journal of Computational and Applied Mathematics 205 (2007), no. 2, 1002-1018.

[15] P.reversal_potential. Kloeden, reversal_potential. Platen, and H. Schurz, Numerical solution of SDE through computer experiments, Springer, 1994.

Attributes

code_scope
py_func_name
update_code
update_func

class brainpy.integration.**Integrator**(*diff_eq*)

class brainpy.integration.**Euler**(*diff_eq*)

Forward Euler method. Also named as explicit_Euler.

The simplest way for solving ordinary differential equations is "the Euler method" by Press et al. (1992)[1] :

$$y_{n+1} = y_n + f(y_n, t_n)\Delta t$$

This formula advances a solution from y_n to $y_{n+1} = y_n + h$. Note that the method increments a solution through an interval h while using derivative information from only the beginning of the interval. As a result, the step's error is $O(h^2)$.

For SDE equations, this approximation is a continuous time stochastic process that satisfy the iterative scheme[1].

$$Y_{n+1} = Y_n + f(Y_n)h_n + g(Y_n)\Delta W_n$$

where $n = 0, 1, \cdots, N-1$, $Y_0 = x_0$, $Y_n = Y(t_n)$, $h_n = t_{n+1} - t_n$ is the step size, $\Delta W_n = [W(t_{n+1}) - W(t_n)] \sim N(0, h_n) = \sqrt{h}N(0, 1)$ with $W(t_0) = 0$.

For simplicity, we rewrite the above equation into

$$Y_{n+1} = Y_n + f_n h + g_n \Delta W_n$$

As the order of convergence for the Euler-Maruyama method is low (strong order of convergence 0.5, weak order of convergence 1), the numerical results are inaccurate unless a small step size is used. By adding one more term from the stochastic Taylor expansion, one obtains a 1.0 strong order of convergence scheme known as *Milstein scheme*[2].

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

References

class brainpy.integration.**Heun**(*diff_eq*)

Two-stage method for numerical integrator.

For ODE, please see "RK2".

For stochastic Stratonovich integral, the Heun algorithm is given by, according to paper[4][5].

$$Y_{n+1} = Y_n + f_n h + \frac{1}{2}[g_n + g(\overline{Y}_n)]\Delta W_n$$
$$\overline{Y}_n = Y_n + g_n \Delta W_n$$

[1] W. H.; Flannery, B. P.; Teukolsky, S. A.; and Vetterling, W. T. Numerical Recipes in FORTRAN: The Art of Scientific Computing, 2nd ed. Cambridge, England: Cambridge University Press, p. 710, 1992.

[2] U. Picchini, Sde toolbox: Simulation and estimation of stochastic differential equations with matlab.

[4] H. Gilsing and T. Shardlow, SDELab: A package for solving stochastic differential equations in MATLAB, Journal of Computational and Applied Mathematics 205 (2007), no. 2, 1002-1018.

[5] P.reversal_potential. Kloeden, reversal_potential. Platen, and H. Schurz, Numerical solution of SDE through computer experiments, Springer, 1994.

Or, it is written as

$$Y_1 = y_n + f(y_n)h + g_n \Delta W_n$$
$$y_{n+1} = y_n + \frac{1}{2}[f(y_n) + f(Y_1)]h + \frac{1}{2}[g(y_n) + g(Y_1)]\Delta W_n$$

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

References

See also:

RK2, *MidPoint*, *MilsteinStra*

class brainpy.integration.**MidPoint**(*diff_eq*)
Explicit midpoint Euler method. Also named as modified_Euler.

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

See also:

RK2, *Heun*

class brainpy.integration.**RK2**(*diff_eq*, *beta=0.6666666666666666*)
Parametric second-order Runge-Kutta (RK2). Also named as RK2.

It is given in parametric form by[3] .

$$k_1 = f(y_n, t_n)$$
$$k_2 = f(y_n + \beta \Delta t k_1, t_n + \beta \Delta t)$$
$$y_{n+1} = y_n + \Delta t[(1 - \frac{1}{2\beta})k_1 + \frac{1}{2\beta}k_2]$$

Parameters

- **diff_eq** (DiffEquation) – The differential equation.
- **beta** (*float*) – Popular choices for 'beta': 1/2 : explicit midpoint method 2/3 : Ralston's method 1 : Heun's method, also known as the explicit trapezoid rule

Returns **func** – The one-step numerical integrator function.

Return type callable

[3] https://lpsa.swarthmore.edu/NumInt/NumIntSecond.html

References

See also:

Heun, *MidPoint*

class brainpy.integration.**RK3**(*diff_eq*)
Kutta's third-order method (commonly known as RK3). Also named as RK3[6][7][8] .

$$
\begin{aligned}
k_1 &= f(y_n, t_n) \\
k_2 &= f(y_n + \frac{\Delta t}{2} k_1, tn + \frac{\Delta t}{2}) \\
k_3 &= f(y_n - \Delta t k_1 + 2\Delta t k_2, t_n + \Delta t) \\
y_{n+1} &= y_n + \frac{\Delta t}{6}(k_1 + 4k_2 + k_3)
\end{aligned}
$$

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

References

class brainpy.integration.**RK4**(*diff_eq*)
Fourth-order Runge-Kutta (RK4)[9][10][11] .

$$
\begin{aligned}
k_1 &= f(y_n, t_n) \\
k_2 &= f(y_n + \frac{\Delta t}{2} k_1, t_n + \frac{\Delta t}{2}) \\
k_3 &= f(y_n + \frac{\Delta t}{2} k_2, t_n + \frac{\Delta t}{2}) \\
k_4 &= f(y_n + \Delta t k_3, t_n + \Delta t) \\
y_{n+1} &= y_n + \frac{\Delta t}{6}(k_1 + 2 * k_2 + 2 * k_3 + k_4)
\end{aligned}
$$

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

References

class brainpy.integration.**RK4Alternative**(*diff_eq*)
An alternative of fourth-order Runge-Kutta method. Also named as RK4_alternative ("3/8" rule).

[6] http://mathworld.wolfram.com/Runge-KuttaMethod.html
[7] https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods
[8] https://zh.wikipedia.org/wiki/

It is a less often used fourth-order explicit RK method, and was also proposed by Kutta[12]:

$$
\begin{aligned}
k_1 &= f(y_n, t_n) \\
k_2 &= f(y_n + \frac{\Delta t}{3}k_1, t_n + \frac{\Delta t}{3}) \\
k_3 &= f(y_n - \frac{\Delta t}{3}k_1 + \Delta t k_2, t_n + \frac{2\Delta t}{3}) \\
k_4 &= f(y_n + \Delta t k_1 - \Delta t k_2 + \Delta t k_3, t_n + \Delta t) \\
y_{n+1} &= y_n + \frac{\Delta t}{8}(k_1 + 3 * k_2 + 3 * k_3 + k_4)
\end{aligned}
$$

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

References

class brainpy.integration.**ExponentialEuler**(*diff_eq*)

First order, explicit exponential Euler method.

For an ODE equation of the form

$$y' = f(y), \quad y(0) = y_0$$

its schema is given by

$$y_{n+1} = y_n + h\varphi(hA)f(y_n)$$

where $A = f'(y_n)$ and $\varphi(z) = \frac{e^z - 1}{z}$.

For linear ODE system: $y' = Ay + B$, the above equation is equal to

$$y_{n+1} = y_n e^{hA} - B/A(1 - e^{hA})$$

For a SDE equation of the form

$$dy = (Ay + F(y))dt + g(y)dW(t) = f(y)dt + g(y)dW(t), \quad y(0) = y_0$$

its schema is given by[16]

$$
\begin{aligned}
y_{n+1} &= e^{\Delta t A}(y_n + g(y_n)\Delta W_n) + \varphi(\Delta t A)F(y_n)\Delta t \\
&= y_n + \Delta t\varphi(\Delta t A)f(y) + e^{\Delta t A}g(y_n)\Delta W_n
\end{aligned}
$$

where $\varphi(z) = \frac{e^z - 1}{z}$.

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

[12] https://en.wikipedia.org/wiki/List_of_Runge%E2%80%93Kutta_methods

[16] Erdoğan, Utku, and Gabriel J. Lord. "A new class of exponential integrators for stochastic differential equations with multiplicative noise." arXiv preprint arXiv:1608.07096 (2016).

References

class brainpy.integration.**MilsteinIto**(*diff_eq*)

Itô stochastic integral. The derivative-free Milstein method is an order 1.0 strong Taylor schema.

The following implementation approximates this derivative thanks to a Runge-Kutta approach[13].

In Itô scheme, it is expressed as

$$Y_{n+1} = Y_n + f_n h + g_n \Delta W_n + \frac{1}{2\sqrt{h}}[g(\overline{Y_n}) - g_n][(\Delta W_n)^2 - h]$$

where $\overline{Y_n} = Y_n + f_n h + g_n \sqrt{h}$.

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

References

class brainpy.integration.**MilsteinStra**(*diff_eq*)

Heun two-stage stochastic numerical method for Stratonovich integral.

Use the Stratonovich Heun algorithm to integrate Stratonovich equation, according to paper[14][15].

$$\begin{aligned} Y_{n+1} &= Y_n + f_n h + \frac{1}{2}[g_n + g(\overline{Y}_n)]\Delta W_n \\ \overline{Y}_n &= Y_n + g_n \Delta W_n \end{aligned}$$

Or, it is written as

$$\begin{aligned} Y_1 &= y_n + f(y_n)h + g_n \Delta W_n \\ y_{n+1} &= y_n + \frac{1}{2}[f(y_n) + f(Y_1)]h + \frac{1}{2}[g(y_n) + g(Y_1)]\Delta W_n \end{aligned}$$

Parameters **diff_eq** (DiffEquation) – The differential equation.

Returns **func** – The one-step numerical integrator function.

Return type callable

References

See also:

MilsteinIto

[13] P.reversal_potential. Kloeden, reversal_potential. Platen, and H. Schurz, Numerical solution of SDE through computer experiments, Springer, 1994.

[14] H. Gilsing and T. Shardlow, SDELab: A package for solving stochastic differential equations in MATLAB, Journal of Computational and Applied Mathematics 205 (2007), no. 2, 1002-1018.

[15] P.reversal_potential. Kloeden, reversal_potential. Platen, and H. Schurz, Numerical solution of SDE through computer experiments, Springer, 1994.

17.3 sympy_tools module

SympyRender()	
SympyPrinter([settings])	Printer that overrides the printing of some basic sympy objects.
str2sympy(str_expr)	
sympy2str(sympy_expr)	

17.3.1 brainpy.integration.SympyRender

class brainpy.integration.**SympyRender**

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
render_Assign(node)	
render_Attribute(node)	
render_AugAssign(node)	
render_BinOp(node)	
render_BinOp_parentheses(left, right, op)	Use a simplified checking whether it is possible to omit parentheses: only omit parentheses for numbers, variable names or function calls.
render_BoolOp(node)	
render_Call(node)	
render_Compare(node)	
render_Constant(node)	
render_Name(node)	
render_NameConstant(node)	
render_Num(node)	
render_UnaryOp(node)	
render_code(code)	
render_element_parentheses(node)	Render an element with parentheses around it or leave them away for numbers, names and function calls.
render_expr(expr[, strip])	
render_func(node)	
render_node(node)	

Attributes

expression_ops

17.3.2 brainpy.integration.SympyPrinter

class brainpy.integration.**SympyPrinter**(*settings=None*)

Printer that overrides the printing of some basic sympy objects. reversal_potential.g. print "a and b" instead of "And(a, b)".

__init__(*settings=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([settings])	Initialize self.
doprint(expr)	Returns printer's representation for expr (as a string)
emptyPrinter(expr)	
parenthesize(item, level[, strict])	
set_global_settings(**settings)	Set system-wide printing settings.
stringify(args, sep[, level])	

Attributes

order
printmethod

17.3.3 brainpy.integration.str2sympy

brainpy.integration.**str2sympy**(*str_expr*)

17.3.4 brainpy.integration.sympy2str

brainpy.integration.**sympy2str**(*sympy_expr*)

class brainpy.integration.**SympyRender**

render_BinOp_parentheses(*left*, *right*, *op*)

Use a simplified checking whether it is possible to omit parentheses: only omit parentheses for numbers, variable names or function calls. This means we still put needless parentheses because we ignore precedence rules, e.g. we write "3 + (4 * 5)" but at least we do not do "(3) + ((4) + (5))"

render_element_parentheses(*node*)

Render an element with parentheses around it or leave them away for numbers, names and function calls.

class brainpy.integration.**SympyPrinter**(*settings=None*)

Printer that overrides the printing of some basic sympy objects. reversal_potential.g. print "a and b" instead of "And(a, b)".

CHAPTER

EIGHTEEN

BRAINPY.DYNAMICS PACKAGE

PhasePortraitAnalyzer(model, target_vars[, ...])	
BifurcationAnalyzer(model, target_pars, ...)	A tool class for bifurcation analysis.

18.1 brainpy.dynamics.PhasePortraitAnalyzer

class brainpy.dynamics.**PhasePortraitAnalyzer**(*model*, *target_vars*, *fixed_vars=None*, *pars_update=None*, *lim_scale=1.05*)

__init__(*model*, *target_vars*, *fixed_vars=None*, *pars_update=None*, *lim_scale=1.05*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(model, target_vars[, fixed_vars, ...])	Initialize self.
plot_fixed_point([resolution, show])	
plot_nullcline([resolution, show])	
plot_trajectory(target_setting[, axes, ...])	
plot_vector_field([resolution, lw_lim, show])	

18.2 brainpy.dynamics.BifurcationAnalyzer

class brainpy.dynamics.**BifurcationAnalyzer**(*model*, *target_pars*, *dynamical_vars*, *fixed_vars=None*, *pars_update=None*, *par_resolution=0.1*, *var_resolution=0.1*)

A tool class for bifurcation analysis.

The bifurcation analyzer is restricted to analyze the bifurcation relation between membrane potential and a given model parameter (codimension-1 case) or two model parameters (codimension-2 case).

Externally injected current is also treated as a model parameter in this class, instead of a model state.

Parameters **model** (NeuType) – An abstract neuronal type defined in BrainPy.

__init__(*model*, *target_pars*, *dynamical_vars*, *fixed_vars=None*, *pars_update=None*, *par_resolution=0.1*, *var_resolution=0.1*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(model, target_pars, dynamical_vars)	Initialize self.
`plot_bifurcation`([plot_vars, show])	

CHAPTER

NINETEEN

BRAINPY.CONNECT PACKAGE

19.1 `formatter` functions

ij2mat(i, j[, num_pre, num_post])	Convert i-j connection to matrix connection.
mat2ij(conn_mat)	Get the i-j connections from connectivity matrix.
pre2post(i, j[, num_pre])	Get pre2post connections from *i* and *j* indexes.
post2pre(i, j[, num_post])	Get post2pre connections from *i* and *j* indexes.
pre2syn(i[, num_pre])	Get pre2syn connections from *i* and *j* indexes.
post2syn(j[, num_post])	Get post2syn connections from *i* and *j* indexes.
pre_slice_syn(i, j[, num_pre])	Get post slicing connections by pre-synaptic ids.
post_slice_syn(i, j[, num_post])	Get pre slicing connections by post-synaptic ids.

19.1.1 brainpy.connectivity.ij2mat

brainpy.connectivity.**ij2mat**(*i*, *j*, *num_pre=None*, *num_post=None*)

Convert i-j connection to matrix connection.

Parameters

- **i** (*list, np.ndarray*) – Pre-synaptic neuron index.
- **j** (*list, np.ndarray*) – Post-synaptic neuron index.
- **num_pre** (*int*) – The number of the pre-synaptic neurons.
- **num_post** (*int*) – The number of the post-synaptic neurons.

Returns **conn_mat** – A 2D ndarray connectivity matrix.

Return type np.ndarray

19.1.2 brainpy.connectivity.mat2ij

brainpy.connectivity.**mat2ij**(*conn_mat*)

Get the i-j connections from connectivity matrix.

Parameters **conn_mat** (*np.ndarray*) – Connectivity matrix with *(num_pre, num_post)* shape.

Returns

conn_tuple –

(Pre-synaptic neuron indexes, post-synaptic neuron indexes).

Return type tuple

19.1.3 brainpy.connectivity.pre2post

brainpy.connectivity.**pre2post**(*i*, *j*, *num_pre=None*)
Get pre2post connections from *i* and *j* indexes.

Parameters

- **i** (*list, np.ndarray*) – The pre-synaptic neuron indexes.
- **j** (*list, np.ndarray*) – The post-synaptic neuron indexes.
- **num_pre** (*int, None*) – The number of the pre-synaptic neurons.

Returns **conn** – The conn list of pre2post.

Return type list

19.1.4 brainpy.connectivity.post2pre

brainpy.connectivity.**post2pre**(*i*, *j*, *num_post=None*)
Get post2pre connections from *i* and *j* indexes.

Parameters

- **i** (*list, np.ndarray*) – The pre-synaptic neuron indexes.
- **j** (*list, np.ndarray*) – The post-synaptic neuron indexes.
- **num_post** (*int, None*) – The number of the post-synaptic neurons.

Returns **conn** – The conn list of post2pre.

Return type list

19.1.5 brainpy.connectivity.pre2syn

brainpy.connectivity.**pre2syn**(*i*, *num_pre=None*)
Get pre2syn connections from *i* and *j* indexes.

Parameters

- **i** (*list, np.ndarray*) – The pre-synaptic neuron indexes.
- **num_pre** (*int*) – The number of the pre-synaptic neurons.

Returns **conn** – The conn list of pre2syn.

Return type list

19.1.6 brainpy.connectivity.post2syn

brainpy.connectivity.**post2syn**(*j*, *num_post=None*)
Get post2syn connections from *i* and *j* indexes.

Parameters

- **j** (*list, np.ndarray*) – The post-synaptic neuron indexes.
- **num_post** (*int*) – The number of the post-synaptic neurons.

Returns **conn** – The conn list of post2syn.

Return type list

19.1.7 brainpy.connectivity.pre_slice_syn

brainpy.connectivity.**pre_slice_syn**(*i*, *j*, *num_pre=None*)
Get post slicing connections by pre-synaptic ids.

Parameters

- **i** (*list, np.ndarray*) – The pre-synaptic neuron indexes.
- **j** (*list, np.ndarray*) – The post-synaptic neuron indexes.
- **num_pre** (*int*) – The number of the pre-synaptic neurons.

Returns **conn** – The conn list of post2syn.

Return type list

19.1.8 brainpy.connectivity.post_slice_syn

brainpy.connectivity.**post_slice_syn**(*i*, *j*, *num_post=None*)
Get pre slicing connections by post-synaptic ids.

Parameters

- **i** (*list, np.ndarray*) – The pre-synaptic neuron indexes.
- **j** (*list, np.ndarray*) – The post-synaptic neuron indexes.
- **num_post** (*int*) – The number of the post-synaptic neurons.

Returns **conn** – The conn list of post2syn.

Return type list

19.2 connector methods

Connector()	Abstract connector class.
One2One()	Connect two neuron groups one by one.
All2All([include_self])	Connect each neuron in first group to all neurons in the post-synaptic neuron groups.
GridFour([include_self])	The nearest four neighbors conn method.
GridEight([include_self])	The nearest eight neighbors conn method.

continues on next page

Table 2 – continued from previous page

GridN([n, include_self])	The nearest (2*N+1) * (2*N+1) neighbors conn method.
FixedPostNum(num[, include_self, seed])	Connect the post-synaptic neurons with fixed number for each pre-synaptic neuron.
FixedPreNum(num[, include_self, seed])	Connect the pre-synaptic neurons with fixed number for each post-synaptic neuron.
FixedProb(prob[, include_self, seed])	Connect the post-synaptic neurons with fixed probability.
GaussianProb(sigma[, p_min, normalize, ...])	Builds a Gaussian conn pattern between the two populations, where the conn probability decay according to the gaussian function.
GaussianWeight(sigma, w_max[, w_min, ...])	Builds a Gaussian conn pattern between the two populations, where the weights decay with gaussian function.
DOG(sigmas, ws_max[, w_min, normalize, ...])	Builds a Difference-Of-Gaussian (dog) conn pattern between the two populations.
SmallWorld()	
ScaleFree()	

19.2.1 brainpy.connectivity.Connector

class brainpy.connectivity.**Connector**

Abstract connector class.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
make_conn_mat()	
make_mat2ij()	
make_post2pre()	
make_post2syn()	
make_post_slice_syn()	
make_pre2post()	
make_pre2syn()	
make_pre_slice_syn()	
set_requires(syn_requires)	
set_size(num_pre, num_post)	

19.2.2 brainpy.connectivity.One2One

class brainpy.connectivity.**One2One**

Connect two neuron groups one by one. This means The two neuron groups should have the same size.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
make_conn_mat()	
make_mat2ij()	
make_post2pre()	
make_post2syn()	
make_post_slice_syn()	
make_pre2post()	
make_pre2syn()	
make_pre_slice_syn()	
set_requires(syn_requires)	
set_size(num_pre, num_post)	

19.2.3 brainpy.connectivity.All2All

class brainpy.connectivity.**All2All**(*include_self=True*)

Connect each neuron in first group to all neurons in the post-synaptic neuron groups. It means this kind of conn will create (num_pre x num_post) synapses.

__init__(*include_self=True*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([include_self])	Initialize self.
make_conn_mat()	
make_mat2ij()	
make_post2pre()	
make_post2syn()	
make_post_slice_syn()	
make_pre2post()	
make_pre2syn()	
make_pre_slice_syn()	
set_requires(syn_requires)	
set_size(num_pre, num_post)	

19.2.4 brainpy.connectivity.GridFour

class brainpy.connectivity.**GridFour**(*include_self=False*)
The nearest four neighbors conn method.

__init__(*include_self=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([include_self])	Initialize self.
make_conn_mat()	
make_mat2ij()	
make_post2pre()	
make_post2syn()	
make_post_slice_syn()	
make_pre2post()	
make_pre2syn()	
make_pre_slice_syn()	
set_requires(syn_requires)	
set_size(num_pre, num_post)	

19.2.5 brainpy.connectivity.GridEight

class brainpy.connectivity.**GridEight**(*include_self=False*)
The nearest eight neighbors conn method.

__init__(*include_self=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([include_self])	Initialize self.
make_conn_mat()	
make_mat2ij()	
make_post2pre()	
make_post2syn()	
make_post_slice_syn()	
make_pre2post()	
make_pre2syn()	
make_pre_slice_syn()	
set_requires(syn_requires)	
set_size(num_pre, num_post)	

19.2.6 brainpy.connectivity.GridN

class brainpy.connectivity.**GridN**(*n=1*, *include_self=False*)
The nearest (2*N+1) * (2*N+1) neighbors conn method.

Parameters

- **N** (*int*) – Extend of the conn scope. For example: When N=1,

  [x x x] [x I x] [x x x]

  When N=2, [x x x x x] [x x x x x] [x x I x x] [x x x x x] [x x x x x]

- **include_self** (*bool*) – Whether create (i, i) conn ?

__init__(*n=1*, *include_self=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([n, include_self])	Initialize self.
make_conn_mat()	
make_mat2ij()	
make_post2pre()	
make_post2syn()	
make_post_slice_syn()	
make_pre2post()	
make_pre2syn()	
make_pre_slice_syn()	
set_requires(syn_requires)	
set_size(num_pre, num_post)	

19.2.7 brainpy.connectivity.FixedPostNum

class brainpy.connectivity.**FixedPostNum**(*num*, *include_self=True*, *seed=None*)
Connect the post-synaptic neurons with fixed number for each pre-synaptic neuron.

Parameters

- **num** (*float, int*) – The conn probability (if "num" is float) or the fixed number of connectivity (if "num" is int).
- **include_self** (*bool*) – Whether create (i, i) conn ?
- **seed** (*None, int*) – Seed the random generator.

__init__(*num*, *include_self=True*, *seed=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(num[, include_self, seed])	Initialize self.
make_conn_mat()	
make_mat2ij()	
make_post2pre()	
make_post2syn()	
make_post_slice_syn()	
make_pre2post()	
make_pre2syn()	
make_pre_slice_syn()	
set_requires(syn_requires)	
set_size(num_pre, num_post)	

19.2.8 brainpy.connectivity.FixedPreNum

class brainpy.connectivity.**FixedPreNum**(*num*, *include_self=True*, *seed=None*)

Connect the pre-synaptic neurons with fixed number for each post-synaptic neuron.

Parameters

- **num** (*float, int*) – The conn probability (if "num" is float) or the fixed number of connectivity (if "num" is int).
- **include_self** (*bool*) – Whether create (i, i) conn ?
- **seed** (*None, int*) – Seed the random generator.

__init__(*num*, *include_self=True*, *seed=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(num[, include_self, seed])	Initialize self.
make_conn_mat()	
make_mat2ij()	
make_post2pre()	
make_post2syn()	
make_post_slice_syn()	
make_pre2post()	
make_pre2syn()	
make_pre_slice_syn()	
set_requires(syn_requires)	
set_size(num_pre, num_post)	

19.2.9 brainpy.connectivity.FixedProb

class brainpy.connectivity.**FixedProb**(*prob*, *include_self=True*, *seed=None*)
Connect the post-synaptic neurons with fixed probability.

Parameters

- **prob** (*float*) – The conn probability.
- **include_self** (*bool*) – Whether create (i, i) conn ?
- **seed** (*None, int*) – Seed the random generator.

__init__(*prob*, *include_self=True*, *seed=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(prob[, include_self, seed])	Initialize self.
make_conn_mat()	
make_mat2ij()	
make_post2pre()	
make_post2syn()	
make_post_slice_syn()	
make_pre2post()	
make_pre2syn()	
make_pre_slice_syn()	
set_requires(syn_requires)	
set_size(num_pre, num_post)	

19.2.10 brainpy.connectivity.GaussianProb

class brainpy.connectivity.**GaussianProb**(*sigma*, *p_min=0.0*, *normalize=True*, *include_self=True*, *seed=None*)
Builds a Gaussian conn pattern between the two populations, where the conn probability decay according to the gaussian function.

Specifically,

$$p = \exp(-\frac{(x-x_c)^2+(y-y_c)^2}{2\sigma^2})$$

where (x, y) is the position of the pre-synaptic neuron and (x_c, y_c) is the position of the post-synaptic neuron.

Parameters

- **sigma** (*float*) – Width of the Gaussian function.
- **normalize** (*bool*) – Whether normalize the coordination.
- **include_self** (*bool*) – Whether create the conn at the same position.
- **seed** (*bool*) – The random seed.

__init__(*sigma*, *p_min=0.0*, *normalize=True*, *include_self=True*, *seed=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(sigma[, p_min, normalize, ...])	Initialize self.
`make_conn_mat`()	
`make_mat2ij`()	
`make_post2pre`()	
`make_post2syn`()	
`make_post_slice_syn`()	
`make_pre2post`()	
`make_pre2syn`()	
`make_pre_slice_syn`()	
`set_requires`(syn_requires)	
`set_size`(num_pre, num_post)	

19.2.11 brainpy.connectivity.GaussianWeight

class brainpy.connectivity.**GaussianWeight**(*sigma*, *w_max*, *w_min=None*, *normalize=True*, *include_self=True*)

Builds a Gaussian conn pattern between the two populations, where the weights decay with gaussian function.

Specifically,

$$w(x,y) = w_{max} \cdot \exp(-\frac{(x-x_c)^2+(y-y_c)^2}{2\sigma^2})$$

where (x, y) is the position of the pre-synaptic neuron (normalized to [0,1]) and (x_c, y_c) is the position of the post-synaptic neuron (normalized to [0,1]), w_{max} is the maximum weight. In order to void creating useless synapses, w_{min} can be set to restrict the creation of synapses to the cases where the value of the weight would be superior to w_{min}. Default is $0.01 w_{max}$.

Parameters

- **sigma** (*float*) – Width of the Gaussian function.
- **w_max** (*float*) – The weight amplitude of the Gaussian function.
- **w_min** (*float, None*) – The minimum weight value below which synapses are not created (default: 0.01 * *w_max*).
- **normalize** (*bool*) – Whether normalize the coordination.
- **include_self** (*bool*) – Whether create the conn at the same position.

__init__(*sigma*, *w_max*, *w_min=None*, *normalize=True*, *include_self=True*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(sigma, w_max[, w_min, normalize, ...])	Initialize self.
`make_conn_mat`()	
`make_mat2ij`()	
`make_post2pre`()	
`make_post2syn`()	

continues on next page

Table 13 – continued from previous page

make_post_slice_syn()
make_pre2post()
make_pre2syn()
make_pre_slice_syn()
set_requires(syn_requires)
set_size(num_pre, num_post)

19.2.12 brainpy.connectivity.DOG

class brainpy.connectivity.**DOG**(*sigmas*, *ws_max*, *w_min=None*, *normalize=True*, *include_self=True*)

Builds a Difference-Of-Gaussian (dog) conn pattern between the two populations.

Mathematically,

$$w(x,y) = w^+_{max} \cdot \exp(-\frac{(x-x_c)^2+(y-y_c)^2}{2\sigma^2_+}) - w^-_{max} \cdot \exp(-\frac{(x-x_c)^2+(y-y_c)^2}{2\sigma^2_-})$$

where weights smaller than $0.01 * abs(w_{max} - w_{min})$ are not created and self-connections are avoided by default (parameter allow_self_connections).

Parameters

- **sigmas** (*tuple*) – Widths of the positive and negative Gaussian functions.
- **ws_max** (*tuple*) – The weight amplitudes of the positive and negative Gaussian functions.
- **w_min** (*float, None*) – The minimum weight value below which synapses are not created (default: $0.01 * w^+_{max} - w^-_{min}$).
- **normalize** (*bool*) – Whether normalize the coordination.
- **include_self** (*bool*) – Whether create the conn at the same position.

__init__(*sigmas*, *ws_max*, *w_min=None*, *normalize=True*, *include_self=True*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(sigmas, ws_max[, w_min, normalize, ...])	Initialize self.
make_conn_mat()	
make_mat2ij()	
make_post2pre()	
make_post2syn()	
make_post_slice_syn()	
make_pre2post()	
make_pre2syn()	
make_pre_slice_syn()	
set_requires(syn_requires)	
set_size(num_pre, num_post)	

19.2.13 brainpy.connectivity.SmallWorld

class brainpy.connectivity.**SmallWorld**

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
make_conn_mat()	
make_mat2ij()	
make_post2pre()	
make_post2syn()	
make_post_slice_syn()	
make_pre2post()	
make_pre2syn()	
make_pre_slice_syn()	
set_requires(syn_requires)	
set_size(num_pre, num_post)	

19.2.14 brainpy.connectivity.ScaleFree

class brainpy.connectivity.**ScaleFree**

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
make_conn_mat()	
make_mat2ij()	
make_post2pre()	
make_post2syn()	
make_post_slice_syn()	
make_pre2post()	
make_pre2syn()	
make_pre_slice_syn()	
set_requires(syn_requires)	
set_size(num_pre, num_post)	

class brainpy.connectivity.**Connector**
Abstract connector class.

class brainpy.connectivity.**One2One**
Connect two neuron groups one by one. This means The two neuron groups should have the same size.

class brainpy.connectivity.**All2All**(*include_self=True*)

Connect each neuron in first group to all neurons in the post-synaptic neuron groups. It means this kind of conn will create (num_pre x num_post) synapses.

class brainpy.connectivity.**GridFour**(*include_self=False*)
The nearest four neighbors conn method.

class brainpy.connectivity.**GridEight**(*include_self=False*)
The nearest eight neighbors conn method.

class brainpy.connectivity.**GridN**(*n=1*, *include_self=False*)
The nearest (2*N+1) * (2*N+1) neighbors conn method.

Parameters

- **N** (*int*) – Extend of the conn scope. For example: When N=1,

  [x x x] [x I x] [x x x]

  When N=2, [x x x x x] [x x x x x] [x x I x x] [x x x x x] [x x x x x]

- **include_self** (*bool*) – Whether create (i, i) conn ?

class brainpy.connectivity.**FixedPostNum**(*num*, *include_self=True*, *seed=None*)
Connect the post-synaptic neurons with fixed number for each pre-synaptic neuron.

Parameters

- **num** (*float, int*) – The conn probability (if "num" is float) or the fixed number of connectivity (if "num" is int).
- **include_self** (*bool*) – Whether create (i, i) conn ?
- **seed** (*None, int*) – Seed the random generator.

class brainpy.connectivity.**FixedPreNum**(*num*, *include_self=True*, *seed=None*)
Connect the pre-synaptic neurons with fixed number for each post-synaptic neuron.

Parameters

- **num** (*float, int*) – The conn probability (if "num" is float) or the fixed number of connectivity (if "num" is int).
- **include_self** (*bool*) – Whether create (i, i) conn ?
- **seed** (*None, int*) – Seed the random generator.

class brainpy.connectivity.**FixedProb**(*prob*, *include_self=True*, *seed=None*)
Connect the post-synaptic neurons with fixed probability.

Parameters

- **prob** (*float*) – The conn probability.
- **include_self** (*bool*) – Whether create (i, i) conn ?
- **seed** (*None, int*) – Seed the random generator.

class brainpy.connectivity.**GaussianProb**(*sigma*, *p_min=0.0*, *normalize=True*, *include_self=True*, *seed=None*)
Builds a Gaussian conn pattern between the two populations, where the conn probability decay according to the gaussian function.

Specifically,

$$p = \exp(-\frac{(x-x_c)^2 + (y-y_c)^2}{2\sigma^2})$$

where (x, y) is the position of the pre-synaptic neuron and (x_c, y_c) is the position of the post-synaptic neuron.

Parameters

- **sigma** (*float*) – Width of the Gaussian function.
- **normalize** (*bool*) – Whether normalize the coordination.
- **include_self** (*bool*) – Whether create the conn at the same position.
- **seed** (*bool*) – The random seed.

class brainpy.connectivity.**GaussianWeight**(*sigma*, *w_max*, *w_min=None*, *normalize=True*, *include_self=True*)

Builds a Gaussian conn pattern between the two populations, where the weights decay with gaussian function.

Specifically,

$$w(x, y) = w_{max} \cdot \exp(-\frac{(x - x_c)^2 + (y - y_c)^2}{2\sigma^2})$$

where (x, y) is the position of the pre-synaptic neuron (normalized to [0,1]) and (x_c, y_c) is the position of the post-synaptic neuron (normalized to [0,1]), w_{max} is the maximum weight. In order to void creating useless synapses, w_{min} can be set to restrict the creation of synapses to the cases where the value of the weight would be superior to w_{min}. Default is $0.01 w_{max}$.

Parameters

- **sigma** (*float*) – Width of the Gaussian function.
- **w_max** (*float*) – The weight amplitude of the Gaussian function.
- **w_min** (*float, None*) – The minimum weight value below which synapses are not created (default: 0.01 * *w_max*).
- **normalize** (*bool*) – Whether normalize the coordination.
- **include_self** (*bool*) – Whether create the conn at the same position.

class brainpy.connectivity.**DOG**(*sigmas*, *ws_max*, *w_min=None*, *normalize=True*, *include_self=True*)

Builds a Difference-Of-Gaussian (dog) conn pattern between the two populations.

Mathematically,

$$w(x, y) = w^+_{max} \cdot \exp(-\frac{(x - x_c)^2 + (y - y_c)^2}{2\sigma^2_+}) - w^-_{max} \cdot \exp(-\frac{(x - x_c)^2 + (y - y_c)^2}{2\sigma^2_-})$$

where weights smaller than $0.01 * abs(w_{max} - w_{min})$ are not created and self-connections are avoided by default (parameter allow_self_connections).

Parameters

- **sigmas** (*tuple*) – Widths of the positive and negative Gaussian functions.
- **ws_max** (*tuple*) – The weight amplitudes of the positive and negative Gaussian functions.
- **w_min** (*float, None*) – The minimum weight value below which synapses are not created (default: $0.01 * w^+_{max} - w^-_{min}$).
- **normalize** (*bool*) – Whether normalize the coordination.
- **include_self** (*bool*) – Whether create the conn at the same position.

class brainpy.connectivity.**SmallWorld**

class brainpy.connectivity.**ScaleFree**

CHAPTER

TWENTY

BRAINPY.VISUALIZE PACKAGE

Visualization toolkit.

get_figure(n_row, n_col[, len_row, len_col])	Get the constrained_layout figure.
plot_style1([fontsize, axes_edgecolor, ...])	Plot style for publication.
line_plot(ts, val_matrix[, plot_ids, ax, ...])	Show the specified value in the given object (Neurons or Synapses.)
raster_plot(ts, sp_matrix[, ax, marker, ...])	Show the rater plot of the spikes.
animate_2D(values, net_size[, dt, val_min, ...])	Animate the potentials of the neuron group.
animate_1D(dynamical_vars[, static_vars, ...])	Animation of one-dimensional data.

20.1 brainpy.visualization.get_figure

brainpy.visualization.**get_figure**(*n_row*, *n_col*, *len_row=3*, *len_col=6*)

Get the constrained_layout figure.

Parameters

- **n_row** (*int*) – The row number of the figure.
- **n_col** (*int*) – The column number of the figure.
- **len_row** (*int, float*) – The length of each row.
- **len_col** (*int, float*) – The length of each column.

Returns **fig_and_gs** – Figure and GridSpec.

Return type tuple

20.2 brainpy.visualization.plot_style1

brainpy.visualization.**plot_style1**(*fontsize=22*, *axes_edgecolor='black'*, *figsize='5,4'*, *lw=1*)

Plot style for publication.

Parameters

- **fontsize** (*int*) – The font size.
- **axes_edgecolor** (*str*) – The exes edge color.
- **figsize** (*str, tuple*) – The figure size.
- **lw** (*int*) – Line width.

20.3 brainpy.visualization.line_plot

brainpy.visualization.**line_plot**(*ts*, *val_matrix*, *plot_ids=None*, *ax=None*, *xlim=None*, *ylim=None*, *xlabel='Time (ms)'*, *ylabel='value'*, *legend=None*, *title=None*, *show=False*)

Show the specified value in the given object (Neurons or Synapses.)

Parameters

- **ts** (*np.ndarray*) – The time steps.
- **val_matrix** (*np.ndarray*) – The value matrix which record the history trajectory. It can be easily accessed by specifying the `monitors` of NeuGroup/SynConn by: `neu/syn = NeuGroup/SynConn(..., monitors=[k1, k2])`
- **plot_ids** (*None, int, tuple, a_list*) – The index of the value to plot.
- **ax** (*None, Axes*) – The figure to plot.
- **xlim** (*list, tuple*) – The xlim.
- **ylim** (*list, tuple*) – The ylim.
- **xlabel** (*str*) – The xlabel.
- **ylabel** (*str*) – The ylabel.
- **legend** (*str*) – The prefix of legend for plot.
- **show** (*bool*) – Whether show the figure.

20.4 brainpy.visualization.raster_plot

brainpy.visualization.**raster_plot**(*ts*, *sp_matrix*, *ax=None*, *marker='.'*, *markersize=2*, *color='k'*, *xlabel='Time (ms)'*, *ylabel='Neuron index'*, *xlim=None*, *ylim=None*, *title=None*, *show=False*)

Show the rater plot of the spikes.

Parameters

- **ts** (*np.ndarray*) – The run times.
- **sp_matrix** (*np.ndarray*) – The spike matrix which records the spike information. It can be easily accessed by specifying the `monitors` of NeuGroup by: `neu = NeuGroup(..., monitors=['spike'])`
- **ax** (*Axes*) – The figure.
- **markersize** (*int*) – The size of the marker.
- **color** (*str*) – The color of the marker.
- **xlim** (*list, tuple*) – The xlim.
- **ylim** (*list, tuple*) – The ylim.
- **xlabel** (*str*) – The xlabel.
- **ylabel** (*str*) – The ylabel.
- **show** (*bool*) – Show the figure.

20.5 brainpy.visualization.animate_2D

brainpy.visualization.**animate_2D**(*values*, *net_size*, *dt=None*, *val_min=None*, *val_max=None*, *cmap=None*, *frame_delay=1.0*, *frame_step=1*, *title_size=10*, *figsize=None*, *gif_dpi=None*, *video_fps=None*, *save_path=None*, *show=True*)

Animate the potentials of the neuron group.

Parameters

- **values** (*np.ndarray*) – The membrane potentials of the neuron group.
- **net_size** (*tuple*) – The size of the neuron group.
- **dt** (*float*) – The time duration of each step.
- **val_min** (*float, int*) – The minimum of the potential.
- **val_max** (*float, int*) – The maximum of the potential.
- **cmap** (*str*) – The colormap.
- **frame_delay** (*int, float*) – The delay to show each frame.
- **frame_step** (*int*) – The step to show the potential. If *frame_step=3*, then each frame shows one of the every three steps.
- **title_size** (*int*) – The size of the title.
- **figsize** (*None, tuple*) – The size of the figure.
- **gif_dpi** (*int*) – Controls the dots per inch for the movie frames. This combined with the figure's size in inches controls the size of the movie. If `None`, use defaults in matplotlib.
- **video_fps** (*int*) – Frames per second in the movie. Defaults to `None`, which will use the animation's specified interval to set the frames per second.
- **save_path** (*None, str*) – The save path of the animation.
- **show** (*bool*) – Whether show the animation.

Returns **figure** – The created figure instance.

Return type plt.figure

20.6 brainpy.visualization.animate_1D

brainpy.visualization.**animate_1D**(*dynamical_vars*, *static_vars=()*, *dt=None*, *xlim=None*, *ylim=None*, *xlabel=None*, *ylabel=None*, *frame_delay=50.0*, *frame_step=1*, *title_size=10*, *figsize=None*, *gif_dpi=None*, *video_fps=None*, *save_path=None*, *show=True*)

Animation of one-dimensional data.

Parameters

- **dynamical_vars** (*dict, np.ndarray, list of np.ndarray, list of dict*) – The dynamical variables which will be animated.
- **static_vars** (*dict, np.ndarray, list of np.ndarray, list of dict*) – The static variables.
- **xticks** (*list, np.ndarray*) – The xticks.

- **dt** (*float*) – The numerical integration step.
- **xlim** (*tuple*) – The xlim.
- **ylim** (*tuple*) – The ylim.
- **xlabel** (*str*) – The xlabel.
- **ylabel** (*str*) – The ylabel.
- **frame_delay** (*int, float*) – The delay to show each frame.
- **frame_step** (*int*) – The step to show the potential. If *frame_step=3*, then each frame shows one of the every three steps.
- **title_size** (*int*) – The size of the title.
- **figsize** (*None, tuple*) – The size of the figure.
- **gif_dpi** (*int*) – Controls the dots per inch for the movie frames. This combined with the figure's size in inches controls the size of the movie. If `None`, use defaults in matplotlib.
- **video_fps** (*int*) – Frames per second in the movie. Defaults to `None`, which will use the animation's specified interval to set the frames per second.
- **save_path** (*None, str*) – The save path of the animation.
- **show** (*bool*) – Whether show the animation.

Returns **figure** – The created figure instance.

Return type plt.figure

CHAPTER

TWENTYONE

BRAINPY.MEASURE PACKAGE

cross_correlation(spikes, bin_size)	Calculate cross correlation index between neurons.
voltage_fluctuation(potentials)	Calculate neuronal synchronization via voltage variance.
raster_plot(sp_matrix, times)	Get spike raster plot which displays the spiking activity of a group of neurons over time.
firing_rate(sp_matrix, width[, window])	Calculate the mean firing rate over in a neuron group.

21.1 brainpy.measure.cross_correlation

brainpy.measure.**cross_correlation**(*spikes*, *bin_size*)

Calculate cross correlation index between neurons.

The coherence[1] between two neurons i and j is measured by their cross-correlation of spike trains at zero time lag within a time bin of $\Delta t = \tau$. More specifically, suppose that a long time interval T is divided into small bins of Δt and that two spike trains are given by $X(l) = 0$ or 1, $Y(l) = 0$ or 1, $l = 1, 2, \ldots, K(T/K = \tau)$. Thus, we define a coherence measure for the pair as:

$$\kappa_{ij}(\tau) = \frac{\sum_{l=1}^{K} X(l)Y(l)}{\sqrt{\sum_{l=1}^{K} X(l) \sum_{l=1}^{K} Y(l)}}$$

The population coherence measure $\kappa(\tau)$ is defined by the average of $\kappa_{ij}(\tau)$ over many pairs of neurons in the network.

Parameters

- **spikes** (*bnp.ndarray*) – The history of spike states of the neuron group. It can be easily get via *StateMonitor(neu, ['spike'])*.
- **bin_size** (*int*) – The bin size to normalize spike states.

Returns **cc_index** – The cross correlation value which represents the synchronization index.

Return type float

[1] Wang, Xiao-Jing, and György Buzsáki. "Gamma oscillation by synaptic inhibition in a hippocampal interneuronal network model." Journal of neuroscience 16.20 (1996): 6402-6413.

References

21.2 brainpy.measure.voltage_fluctuation

brainpy.measure.**voltage_fluctuation**(*potentials*)

Calculate neuronal synchronization via voltage variance.

The method comes from[1][2][3].

First, average over the membrane potential V

$$V(t) = \frac{1}{N}\sum_{i=1}^{N} V_i(t)$$

The variance of the time fluctuations of $V(t)$ is

$$\sigma_V^2 = \left\langle [V(t)]^2 \right\rangle_t - [\langle V(t) \rangle_t]^2$$

where $\langle \ldots \rangle_t = (1/T_m)\int_0^{T_m} dt \ldots$ denotes time-averaging over a large time, τ_m. After normalization of σ_V to the average over the population of the single cell membrane potentials

$$\sigma_{V_i}^2 = \left\langle [V_i(t)]^2 \right\rangle_t - [\langle V_i(t) \rangle_t]^2$$

one defines a synchrony measure, $\chi(N)$, for the activity of a system of N neurons by:

$$\chi^2(N) = \frac{\sigma_V^2}{\frac{1}{N}\sum_{i=1}^{N}\sigma_{V_i}^2}$$

Parameters **potentials** (*numpy.ndarray*) – The membrane potentials of the neuron group, which can be easily accessed by *StateMonitor(neu, ['V'])*.

Returns **sync_index** – The synchronization index.

Return type float

References

21.3 brainpy.measure.raster_plot

brainpy.measure.**raster_plot**(*sp_matrix*, *times*)

Get spike raster plot which displays the spiking activity of a group of neurons over time.

Parameters

- **sp_matrix** (*bnp.ndarray*) – The matrix which record spiking activities.
- **times** (*bnp.ndarray*) – The time steps.

Returns **raster_plot** – Include (neuron index, spike time).

Return type tuple

[1] Golomb, D. and Rinzel J. (1993) Dynamics of globally coupled inhibitory neurons with heterogeneity. Phys. Rev. reversal_potential 48:4810-4814.

[2] Golomb D. and Rinzel J. (1994) Clustering in globally coupled inhibitory neurons. Physica D 72:259-282.

[3] David Golomb (2007) Neuronal synchrony measures. Scholarpedia, 2(1):1347.

21.4 brainpy.measure.firing_rate

brainpy.measure.**firing_rate**(*sp_matrix*, *width*, *window='gaussian'*)

Calculate the mean firing rate over in a neuron group.

This method is adopted from Brian2.

The firing rate in trial k is the spike count n_k^{sp} in an interval of duration T divided by T:

$$v_k = \frac{n_k^{sp}}{T}$$

Parameters

- **sp_matrix** (*bnp.ndarray*) – The spike matrix which record spiking activities.
- **width** (*int, float*) – The width of the `window` in millisecond.
- **window** (*str*) – The window to use for smoothing. It can be a string to chose a predefined window:
  - *flat*: a rectangular,
  - *gaussian*: a Gaussian-shaped window.

  For the *Gaussian* window, the *width* parameter specifies the standard deviation of the Gaussian, the width of the actual window is *4 * width + dt*. For the *flat* window, the width of the actual window is *2 * width/2 + dt*.

Returns **rate** – The population rate in Hz, smoothed with the given window.

Return type numpy.ndarray

CHAPTER

TWENTYTWO

BRAINPY.RUNNING PACKAGE

process_pool(func, all_net_params, nb_process)	Run multiple models in multi-processes.
process_pool_lock(func, all_net_params, ...)	Run multiple models in multi-processes with lock.

22.1 brainpy.running.process_pool

brainpy.running.**process_pool**(*func*, *all_net_params*, *nb_process*)

Run multiple models in multi-processes.

Parameters

- **func** (*callable*) – The function to run model.
- **all_net_params** (*a_list, tuple*) – The parameters of the function arguments. The parameters for each process can be a tuple, or a dictionary.
- **nb_process** (*int*) – The number of the processes.

Returns **results** – Process results.

Return type list

22.2 brainpy.running.process_pool_lock

brainpy.running.**process_pool_lock**(*func*, *all_net_params*, *nb_process*)

Run multiple models in multi-processes with lock.

Sometimes, you want to synchronize the processes. For example, if you want to write something in a document, you cannot let multi-process simultaneously open this same file. So, you need add a *lock* argument in your defined *func*:

In such case, you can use *process_pool_lock()* to run your model.

Parameters

- **func** (*callable*) – The function to run model.
- **all_net_params** (*a_list, tuple*) – The parameters of the function arguments.
- **nb_process** (*int*) – The number of the processes.

Returns **results** – Process results.

Return type list

CHAPTER

TWENTYTHREE

BRAINPY.INPUTS PACKAGE

constant_current(Iext[, dt])	Format constant input in durations.
spike_current(points, lengths, sizes, duration)	Format current input like a series of short-time spikes.
ramp_current(c_start, c_end, duration[, ...])	Get the gradually changed input current.
PoissonInput(geometry, freqs[, monitors, name])	The Poisson input neuron group.
SpikeTimeInput(geometry, times[, indices, ...])	The input neuron group characterized by spikes emitting at given times.
FreqInput(geometry, freqs[, start_time, ...])	The input neuron group characterized by frequency.

23.1 brainpy.inputs.constant_current

brainpy.inputs.**constant_current**(*Iext*, *dt=None*)

Format constant input in durations.

For example:

If you want to get an input where the size is 0 bwteen 0-100 ms, and the size is 1. between 100-200 ms. >>> constant_current([(0, 100), (1, 100)]) >>> constant_current([(np.zeros(100), 100), (np.random.rand(100), 100)])

Parameters

- **Iext** (*list*) – This parameter receives the current size and the current duration pairs, like *[(size1, duration1), (size2, duration2)]*.
- **dt** (*float*) – Default is None.

Returns **current_and_duration** – (The formatted current, total duration)

Return type tuple

23.2 brainpy.inputs.spike_current

brainpy.inputs.**spike_current**(*points*, *lengths*, *sizes*, *duration*, *dt=None*)

Format current input like a series of short-time spikes.

For example:

If you want to generate a spike train at 10 ms, 20 ms, 30 ms, 200 ms, 300 ms, and each spike lasts 1 ms and the spike current is 0.5, then you can use the following funtions:

```
>>> spike_current(points=[10, 20, 30, 200, 300],
>>>               lengths=1.,  # can be a list to specify the spike length at␣
↪each point
>>>               sizes=0.5,  # can be a list to specify the current size at each␣
↪point
>>>               duration=400.)
```

Parameters

- **points** (*list, tuple*) – The spike time-points. Must be an iterable object.
- **lengths** (*int, float, list, tuple*) – The length of each point-current, mimicking the spike durations.
- **sizes** (*int, float, list, tuple*) – The current sizes.
- **duration** (*int, float*) – The total current duration.
- **dt** (*float*) – The default is None.

Returns **current_and_duration** – (The formatted current, total duration)

Return type tuple

23.3 brainpy.inputs.ramp_current

brainpy.inputs.**ramp_current**(*c_start*, *c_end*, *duration*, *t_start=0*, *t_end=None*, *dt=None*)

Get the gradually changed input current.

Parameters

- **c_start** (*float*) – The minimum (or maximum) current size.
- **c_end** (*float*) – The maximum (or minimum) current size.
- **duration** (*int, float*) – The total duration.
- **t_start** (*float*) – The ramped current start time-point.
- **t_end** (*float*) – The ramped current end time-point. Default is the None.
- **dt** –

Returns **current_and_duration** – (The formatted current, total duration)

Return type tuple

23.4 brainpy.inputs.PoissonInput

class brainpy.inputs.**PoissonInput**(*geometry*, *freqs*, *monitors=None*, *name=None*)

The Poisson input neuron group.

Note: The `PoissonGroup` does not work for high-frequency rates. This is because more than one spike might fall into a single time step (`dt`). However, you can split high frequency rates into several neurons with lower frequency rates. For example, use `PoissonGroup(10, 100)` instead of `PoissonGroup(1, 1000)`.

Parameters

- **geometry** (*int, tuple, list*) – The neuron group geometry.

- **freqs** (*float, int, np.ndarray*) – The spike rates.
- **monitors** (*list, tuple*) – The targets for monitoring.
- **name** (*str*) – The neuron group name.

__init__(*geometry*, *freqs*, *monitors=None*, *name=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(geometry, freqs[, monitors, name])	Initialize self.
get_schedule()	
run(duration[, inputs, report, report_percent])	
set_schedule(schedule)	
type_checking()	

Attributes

requires

23.5 brainpy.inputs.SpikeTimeInput

class brainpy.inputs.**SpikeTimeInput**(*geometry*, *times*, *indices=None*, *monitors=None*, *name=None*)
The input neuron group characterized by spikes emitting at given times.

```
>>> # Get 2 neurons, firing spikes at 10 ms and 20 ms.
>>> SpikeTimeInput(2, times=[10, 20])
>>> # or
>>> # Get 2 neurons, the neuron 0 fires spikes at 10 ms and 20 ms.
>>> SpikeTimeInput(2, times=[10, 20], indices=0)
>>> # or
>>> # Get 2 neurons, neuron 0 fires at 10 ms and 30 ms, neuron 1 fires at 20 ms.
>>> SpikeTimeInput(2, times=[10, 20, 30], indices=[0, 1, 0])
>>> # or
>>> # Get 2 neurons; at 10 ms, neuron 0 fires; at 20 ms, neuron 0 and 1 fire;
>>> # at 30 ms, neuron 1 fires.
>>> SpikeTimeInput(2, times=[10, 20, 30], indices=[0, [0, 1], 1])
```

Parameters

- **geometry** (*int, tuple, list*) – The neuron group geometry.
- **indices** (*int, list, tuple*) – The neuron indices at each time point to emit spikes.
- **times** (*list, np.ndarray*) – The time points which generate the spikes.
- **monitors** (*list, tuple*) – The targets for monitoring.
- **name** (*str*) – The group name.

__init__ (*geometry*, *times*, *indices=None*, *monitors=None*, *name=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(geometry, times[, indices, ...])	Initialize self.
`get_schedule`()	
`run`(duration[, inputs, report, report_percent])	
`set_schedule`(schedule)	
`type_checking`()	

Attributes

`requires`

23.6 brainpy.inputs.FreqInput

class brainpy.inputs.**FreqInput** (*geometry*, *freqs*, *start_time=0.0*, *monitors=None*, *name=None*)
The input neuron group characterized by frequency.

For examples:

```
>>> # Get 2 neurons, with 10 Hz firing rate.
>>> FreqInput(2, freq=10.)
>>> # Get 4 neurons, with 20 Hz firing rate. The neurons
>>> # start firing at [10, 30] ms randomly.
>>> FreqInput(4, freq=20., start_time=np.random.randint(10, 30, (4,)))
```

Parameters

- **geometry** (*int, list, tuple*) – The geometry of neuron group.
- **freqs** (*int, float, np.ndarray*) – The output spike frequency.
- **start_time** (*float*) – The time of the first spike.
- **monitors** (*list, tuple*) – The targets for monitoring.
- **name** (*str*) – The name of the neuron group.

__init__ (*geometry*, *freqs*, *start_time=0.0*, *monitors=None*, *name=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(geometry, freqs[, start_time, ...])	Initialize self.
`get_schedule`()	
`run`(duration[, inputs, report, report_percent])	
`set_schedule`(schedule)	
`type_checking`()	

Attributes

`requires`

CHAPTER

TWENTYFOUR

BRAINPY.ERRORS PACKAGE

ModelDefError	Model definition error.
ModelUseError	Model use error.
TypeMismatchError	
IntegratorError	
DiffEquationError	
CodeError	

24.1 brainpy.errors.ModelDefError

exception brainpy.errors.**ModelDefError**
Model definition error.

24.2 brainpy.errors.ModelUseError

exception brainpy.errors.**ModelUseError**
Model use error.

24.3 brainpy.errors.TypeMismatchError

exception brainpy.errors.**TypeMismatchError**

24.4 brainpy.errors.IntegratorError

exception brainpy.errors.**IntegratorError**

24.5 brainpy.errors.DiffEquationError

exception brainpy.errors.**DiffEquationError**

24.6 brainpy.errors.CodeError

exception brainpy.errors.**CodeError**

CHAPTER

TWENTYFIVE

BRAINPY.TOOLS PACKAGE

25.1 ast2code module

ast2code(ast_node[, indent, line_length])	Decompiles an AST into Python code.

25.1.1 brainpy.tools.ast2code

brainpy.tools.**ast2code**(*ast_node*, *indent=4*, *line_length=100*)

Decompiles an AST into Python code.

Parameters

- **ast_node** (*ast.Node*) – Code to decompile, using AST objects as generated by the standard library ast module.
- **indent** (*int*) – Indent level of lines.
- **line_length** (*int*) – If the line become longer than "line_length", this module will break them up

Returns **code** – Python code string.

Return type str

25.2 codes module

CodeLineFormatter()	
format_code(code_string)	Get code lines from the string.
LineFormatterForTrajectory(fixed_vars)	
format_code_for_trajectory(code_string, ...)	Get _code lines from the string.
FindAtomicOp(var2idx)	
find_atomic_op(code_line, var2idx)	
FuncCallFinder(func_name)	
replace_func(code, func_name)	
DiffEquationAnalyser()	
analyse_diff_eq(eq_code)	

continues on next page

Table 2 – continued from previous page

get_identifiers(expr[, include_numbers])	Return all the identifiers in a given string `expr`, that is everything that matches a programming language variable like expression, which is here implemented as the regexp `\b[A-Za-z_][A-Za-z0-9_]*\b`.
get_main_code(func)	Get the main function _code string.
get_line_indent(line[, spaces_per_tab])	
indent(text[, num_tabs, spaces_per_tab, tab])	
deindent(text[, num_tabs, spaces_per_tab, ...])	
word_replace(expr, substitutions)	Applies a dict of word substitutions.
is_lambda_function(func)	Check whether the function is a `lambda` function.
func_call(args)	
get_func_source(func)	

25.2.1 brainpy.tools.CodeLineFormatter

class brainpy.tools.**CodeLineFormatter**

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
generic_visit(node)	Called if no explicit visitor function exists for a node.
visit(node)	Visit a node.
visit_AnnAssign(node)	
visit_Assert(node[, level])	
visit_Assign(node[, level])	
visit_AugAssign(node[, level])	
visit_Delete(node)	
visit_Expr(node[, level])	
visit_Expression(node[, level])	
visit_For(node[, level])	
visit_If(node[, level])	
visit_Raise(node)	
visit_Try(node)	
visit_While(node[, level])	
visit_With(node)	
visit_content_in_condition_control(node, level)	
visit_node_not_assign(node[, level])	

25.2.2 brainpy.tools.format_code

brainpy.tools.**format_code**(*code_string*)
Get code lines from the string.

Parameters **code_string** –

Returns code_lines

Return type list

25.2.3 brainpy.tools.LineFormatterForTrajectory

class brainpy.tools.**LineFormatterForTrajectory**(*fixed_vars*)

__init__(*fixed_vars*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(fixed_vars)	Initialize self.
generic_visit(node)	Called if no explicit visitor function exists for a node.
visit(node)	Visit a node.
visit_AnnAssign(node)	
visit_Assert(node[, level])	
visit_Assign(node[, level])	
visit_AugAssign(node[, level])	
visit_Delete(node)	
visit_Expr(node[, level])	
visit_Expression(node[, level])	
visit_For(node[, level])	
visit_If(node[, level])	
visit_Raise(node)	
visit_Try(node)	
visit_While(node[, level])	
visit_With(node)	
visit_content_in_condition_control(node, level)	
visit_node_not_assign(node[, level])	

25.2.4 brainpy.tools.format_code_for_trajectory

brainpy.tools.**format_code_for_trajectory**(*code_string*, *fixed_vars*)
Get _code lines from the string.

Parameters **code_string** –

Returns code_lines

Return type list

25.2.5 brainpy.tools.FindAtomicOp

class brainpy.tools.**FindAtomicOp**(*var2idx*)

__init__(*var2idx*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(var2idx)	Initialize self.
generic_visit(node)	Called if no explicit visitor function exists for a node.
visit(node)	Visit a node.
visit_Assign(node)	
visit_AugAssign(node)	

25.2.6 brainpy.tools.find_atomic_op

brainpy.tools.**find_atomic_op**(*code_line*, *var2idx*)

25.2.7 brainpy.tools.FuncCallFinder

class brainpy.tools.**FuncCallFinder**(*func_name*)

__init__(*func_name*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(func_name)	Initialize self.
generic_visit(node)	Called if no explicit visitor function exists for a node.
visit(node)	Visit a node.
visit_Call(node)	

25.2.8 brainpy.tools.replace_func

brainpy.tools.**replace_func**(*code*, *func_name*)

25.2.9 brainpy.tools.DiffEquationAnalyser

class brainpy.tools.**DiffEquationAnalyser**

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
generic_visit(node)	Called if no explicit visitor function exists for a node.
visit(node)	Visit a node.
visit_AnnAssign(node)	
visit_Assign(node)	
visit_AugAssign(node)	
visit_Delete(node)	
visit_For(node)	
visit_If(node)	
visit_IfExp(node)	
visit_Raise(node)	
visit_Return(node)	
visit_Try(node)	
visit_While(node)	
visit_With(node)	

Attributes

expression_ops

25.2.10 brainpy.tools.analyse_diff_eq

brainpy.tools.**analyse_diff_eq**(*eq_code*)

25.2.11 brainpy.tools.get_identifiers

brainpy.tools.**get_identifiers**(*expr*, *include_numbers=False*)
Return all the identifiers in a given string `expr`, that is everything that matches a programming language variable like expression, which is here implemented as the regexp `\b[A-Za-z_][A-Za-z0-9_]*\b`.

Parameters

- **expr** (*str*) – The string to analyze
- **include_numbers** (*bool, optional*) – Whether to include number literals in the output. Defaults to `False`.

Returns **identifiers** – A set of all the identifiers (and, optionally, numbers) in *expr*.

Return type set

Examples

```
>>> expr = '3-a*_b+c5+8+f(A - .3e-10, tau_2)*17'
>>> ids = get_identifiers(expr)
>>> print(sorted(list(ids)))
['A', '_b', 'a', 'c5', 'f', 'tau_2']
>>> ids = get_identifiers(expr, include_numbers=True)
>>> print(sorted(list(ids)))
['.3e-10', '17', '3', '8', 'A', '_b', 'a', 'c5', 'f', 'tau_2']
```

25.2.12 brainpy.tools.get_main_code

brainpy.tools.**get_main_code**(*func*)

Get the main function _code string.

For lambda function, return the

Parameters **func**(*callable, Optional, int, float*) –

25.2.13 brainpy.tools.get_line_indent

brainpy.tools.**get_line_indent**(*line*, *spaces_per_tab=4*)

25.2.14 brainpy.tools.indent

brainpy.tools.**indent**(*text*, *num_tabs=1*, *spaces_per_tab=4*, *tab=None*)

25.2.15 brainpy.tools.deindent

brainpy.tools.**deindent**(*text*, *num_tabs=None*, *spaces_per_tab=4*, *docstring=False*)

25.2.16 brainpy.tools.word_replace

brainpy.tools.**word_replace**(*expr*, *substitutions*)

Applies a dict of word substitutions.

The dict `substitutions` consists of pairs `(word, rep)` where each word `word` appearing in `expr` is replaced by `rep`. Here a 'word' means anything matching the regexp `\bword\b`.

Examples

```
>>> expr = 'a*_b+c5+8+f(A)'
>>> print(word_replace(expr, {'a':'banana', 'f':'func'}))
banana*_b+c5+8+func(A)
```

25.2.17 brainpy.tools.is_lambda_function

brainpy.tools.**is_lambda_function**(*func*)

Check whether the function is a `lambda` function. Comes from https://stackoverflow.com/questions/23852423/how-to-check-that-variable-is-a-lambda-function

Parameters **func** (*callable function*) – The function.

Returns True of False.

Return type bool

25.2.18 brainpy.tools.func_call

brainpy.tools.**func_call**(*args*)

25.2.19 brainpy.tools.get_func_source

brainpy.tools.**get_func_source**(*func*)

25.3 dicts module

DictPlus(*args, **kwargs)	Python dictionaries with advanced dot notation access.

25.3.1 brainpy.tools.DictPlus

class brainpy.tools.**DictPlus**(**args*, ***kwargs*)

Python dictionaries with advanced dot notation access.

For example:

```
>>> d = DictPlus({'a': 10, 'b': 20})
>>> d.a
10
>>> d['a']
10
>>> d.c  # this will raise a KeyError
KeyError: 'c'
>>> d.c = 30  # but you can assign a value to a non-existing item
>>> d.c
30
```

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(*args, **kwargs)	Initialize self.
clear()	
copy()	
deepcopy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	2-tuple; but raise KeyError if D is empty.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
to_dict()	
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

class brainpy.tools.**DictPlus**(**args*, ***kwargs*)

Python dictionaries with advanced dot notation access.

For example:

```
>>> d = DictPlus({'a': 10, 'b': 20})
>>> d.a
10
>>> d['a']
10
>>> d.c  # this will raise a KeyError
KeyError: 'c'
>>> d.c = 30  # but you can assign a value to a non-existing item
>>> d.c
30
```

copy() → a shallow copy of D

setdefault(*key*, *default=None*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update(\[*E*\], ***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

25.4 functions module

jit([func])	JIT user defined functions.
func_copy(f)	Make a deepcopy of a python function.
numba_func(func[, params])	
get_func_name(func[, replace])	
get_func_scope(func[, include_dispatcher])	Get function scope variables.

25.4.1 brainpy.tools.jit

brainpy.tools.**jit**(*func=None*)

JIT user defined functions.

Parameters **func** (*callable, a_list, str*) – The function to be jit.

Returns **jit_func** – function.

Return type callable

25.4.2 brainpy.tools.func_copy

brainpy.tools.**func_copy**(*f*)

Make a deepcopy of a python function.

This method is adopted from http://stackoverflow.com/a/6528148/190597 (Glenn Maynard).

Parameters **f** (*callable*) – Function to copy.

Returns **g** – Copied function.

Return type callable

25.4.3 brainpy.tools.numba_func

brainpy.tools.**numba_func**(*func*, *params={}*)

25.4.4 brainpy.tools.get_func_name

brainpy.tools.**get_func_name**(*func*, *replace=False*)

25.4.5 brainpy.tools.get_func_scope

brainpy.tools.**get_func_scope**(*func*, *include_dispatcher=False*)

Get function scope variables.

Parameters

- **func** (*callable, Integrator*) –
- **include_dispatcher** –

25.5 logger module

show_code_scope(code_scope[, ignores])	
show_code_str(func_code)	

25.5.1 brainpy.tools.show_code_scope

brainpy.tools.**show_code_scope**(*code_scope*, *ignores=()*)

25.5.2 brainpy.tools.show_code_str

brainpy.tools.**show_code_str**(*func_code*)

CHAPTER

TWENTYSIX

RELEASE NOTES

26.1 BrainPy 0.3.1

- Add a more flexible way for NeuState/SynState initialization
- Fix bugs of “is_multi_return”
- Add “hand_overs”, “requires” and “satisfies”.
- Update documentation
- Auto-transform *range* to *numba.prange*
- Support *_obj_i*, *_pre_i*, *_post_i* for more flexible operation in scalar-based models

26.2 BrainPy 0.3.0

26.2.1 Computation API

- Rename “brainpy.numpy” to “brainpy.backend”
- Delete “pytorch”, “tensorflow” backends
- Add “numba” requirement
- Add GPU support

26.2.2 Profile setting

- Delete “backend” profile setting, add “jit”

26.2.3 Core systems

- Delete “autopepe8” requirement
- Delete the format code prefix
- Change keywords “_t_, _dt_, _i_” to “_t, _dt, _i”
- Change the “ST” declaration out of “requires”
- Add “repeat” mode run in Network
- Change “vector-based” to “mode” in NeuType and SynType definition

26.2.4 Package installation

- Remove “pypi” installation, installation now only rely on “conda”

26.3 BrainPy 0.2.4

26.3.1 API changes

- Fix bugs

26.4 BrainPy 0.2.3

26.4.1 API changes

- Add “animate_1D” in `visualization` module
- Add “PoissonInput”, “SpikeTimeInput” and “FreqInput” in `inputs` module
- Update phase_portrait_analyzer.py

26.4.2 Models and examples

- Add CANN examples

26.5 BrainPy 0.2.2

26.5.1 API changes

- Redesign visualization
- Redesign connectivity
- Update docs

26.6 BrainPy 0.2.1

26.6.1 API changes

- Fix bugs in *numba import*
- Fix bugs in *numpy* mode with *scalar* model

26.7 BrainPy 0.2.0

26.7.1 API changes

- For computation: `numpy`, `numba`
- For model definition: `NeuType`, `SynConn`
- For model running: `Network`, `NeuGroup`, `SynConn`, `Runner`
- For numerical integration: `integrate`, `Integrator`, `DiffEquation`
- For connectivity: `One2One`, `All2All`, `GridFour`, `grid_four`, `GridEight`, `grid_eight`, `GridN`, `FixedPostNum`, `FixedPreNum`, `FixedProb`, `GaussianProb`, `GaussianWeight`, `DOG`
- For visualization: `plot_value`, `plot_potential`, `plot_raster`, `animation_potential`
- For measurement: `cross_correlation`, `voltage_fluctuation`, `raster_plot`, `firing_rate`
- For inputs: `constant_current`, `spike_current`, `ramp_current`.

26.7.2 Models and examples

- Neuron models: `HH model`, `LIF model`, `Izhikevich model`
- Synapse models: `AMPA`, `GABA`, `NMDA`, `STP`, `GapJunction`
- Network models: `gamma oscillation`

CHAPTER

TWENTYSEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W