BrainPy

Release 1.0.2

Chaoming Wang

May 29, 2021

QUICKSTART

Brain modeling heavily relies on calculus. Focused on differential equations, BrainPy provides an integrative simulation and analysis framework for neurodynamics in computational neuroscience and brain-inspired computation. It provides three core functions:

- *General numerical solvers* for ODEs and SDEs (future will support DDEs and FDEs).
- *Neurodynamics simulation tools* for various brain objects, such like neurons, synapses and networks (future will support soma and dendrites).
- *Neurodynamics analysis tools* for differential equations, including phase plane analysis and bifurcation analysis (future will support continuation analysis and sensitive analysis).

Intuitive tutorials of BrainPy please see our handbook, and comprehensive examples of BrainPy please see BrainModels.

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 pip

You can install BrainPy from the pypi. To do so, use:

```
pip install -U brainpy-simulator
```

1.2 Installation with Anaconda

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

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

1.3 Installation from source

If you decide not to use conda or pip, you can install BrainPy from 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@V1.0.0
```

1.4 Package Dependency

The normal functions of `BrainPy` for dynamics simulation only relies on NumPy and Matplotlib. You can install these two packages through

```
pip install numpy matplotlib

# or

conda install numpy matplotlib
```

Some numerical solvers (such like *exponential euler* methods) and the dynamics analysis module heavily rely on symbolic mathematics library SymPy. Therefore, we highly recommend you to install sympy via

```
pip install sympy

# or

conda install sympy
```

If you use `BrainPy` for your computational neuroscience project, we recommend you to install Numba. This is because BrainPy heavily rely on Numba for speed acceleration in almost its every module, such like connectivity, simulation, analysis, and measurements. Numba is also a suitable framework for the computation of sparse synaptic connections commonly used in the computational neuroscience project. Install Numba is a piece of cake. You just need type the following commands in you terminal:

```
pip install numba

# or

conda install numba
```

As we stated later, `BrainPy` is a backend-independent neural simulator. You can define and run your models on nearly any computation backends you prefer. These packages can be installed by your project's need.

CHAPTER

TWO

NUMERICAL SOLVERS

Brain modeling toolkit provided in BrainPy is focused on **differential equations**. How to solve differential equations is the essence of the neurodynamics simulation. The exact algebraic solutions are only available for low-order differential equations. For the coupled high-dimensional non-linear brain dynamical systems, we need to resort to using numerical methods for solving such differential equations. In this section, I will illustrate how to define ordinary differential quations (ODEs), stochastic differential equations (SDEs), and how to define the numerical integration methods in BrainPy for these difined DEs.

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

2.1 ODEs

2.1.1 How to define ODE functions?

BrainPy provides a convenient and intuitive way to define ODE systems. For the ODE

$$\frac{dx}{dt} = f_1(x, t, y, p_1)$$
$$\frac{dy}{dt} = f_2(y, t, x, p_2)$$

we can define this system as a Python function:

[2]:
```
def diff(x, y, t, p1, p2):
    dx = f1(x, t, y, p1)
    dy = g1(y, t, x, p2)
    return dx, dy
```

where `t` denotes the current time, `p1` and `p2` which after the `t` are represented as parameters needed in this system, and `x` and `y` passed before `t` denotes the dynamical variables. In the function body, the derivative for each variable can be customized by the user need `f1` and `f2`. Finally, we return the corresponding derivatives `dx` and `dy` with the order the same as the variables in the function arguments.

For each variable `x` or `y`, it can be a scalar (`var_type = bp.SCALAR_VAR`), a vector/matrix (`var_type = bp.POPU_VAR`), or a system (`var_type = bp.SYSTEM_VAR`). Here, the "system" means that the argument `x` denotes an array of vairables. Take the above example as the demonstration again, we can redefine it as:

[3]:
```
import numpy as np

def diff(xy, t, p1, p2):
    x, y = xy
```

(continues on next page)

(continued from previous page)

```
    dx = f1(x, t, y, p1)
    dy = g1(y, t, x, p2)
    return np.array([dx, dy])
```

2.1.2 How to define the numerical integration for ODEs?

After the definition of ODE functions, the numerical integration of these functions are very easy in BrainPy. We just need put a decorator (`bp.odeint`).

[4]:
```
@bp.odeint
def diff(x, y, t, p1, p2):
    dx = f1(x, t, y, p1)
    dy = g1(y, t, x, p2)
    return dx, dy
```

`bp.odeint` receives "method", "dt" etc. specification. By providing "method", user can specify the numerical methods to integrate the ODE functions. The supported ODE method can be found by

[5]:
```
bp.integrators.SUPPORTED_ODE_METHODS
```

[5]:
```
['bs',
 'ck',
 'euler',
 'exponential_euler',
 'heun2',
 'heun3',
 'heun_euler',
 'midpoint',
 'ralston2',
 'ralston3',
 'ralston4',
 'rk2',
 'rk3',
 'rk4',
 'rk4_38rule',
 'rkdp',
 'rkf12',
 'rkf45',
 'ssprk3']
```

Moreover, "dt" is a float which denotes the numerical integration precision. Here, for the above ODE function, we can define a four-order Runge-Kutta method for it:

[6]:
```
@bp.odeint(method='rk4', dt=0.01)
def diff(x, y, t, p1, p2):
    dx = f1(x, t, y, p1)
    dy = g1(y, t, x, p2)
    return dx, dy
```

2.1.3 Example 1: FitzHugh–Nagumo model

Now, let's take the well known FitzHugh–Nagumo model as an exmaple to illustrate how to define ODE solvers for brain modeling. The FitzHugh–Nagumo model (FHN) model has two dynamical variables, which are governed by the following equations:

$$\tau \dot{w} = v + a - bw \tag{2.1}$$

$$\dot{v} = v - \frac{v^3}{3} - w + I_{ext} \tag{2.2}$$

(2.3)

For this FHN model, we can code it in BrainPy like this:

```
[7]: @bp.odeint(dt=0.01)
     def integral(V, w, t, Iext, a, b, tau):
         dw = (V + a - b * w) / tau
         dV = V - V * V * V / 3 - w + Iext
         return dV, dw
```

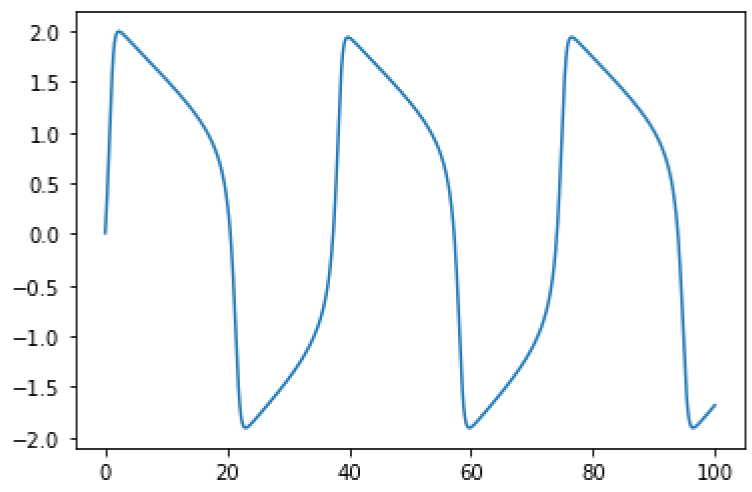
After defining the numerical solver, the solution of the ODE system in the given times can be easily solved. For example, for the given parameters,

```
[8]: a=0.7;   b=0.8;   tau=12.5;   Iext=1.
```

the solution of the FHN model between 0 and 100 ms can be approximated by

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

     hist_times = np.arange(0, 100, integral.dt)
     hist_V = []
     V, w = 0., 0.
     for t in hist_times:
         V, w = integral(V, w, t, Iext, a, b, tau)
         hist_V.append(V)

     plt.plot(hist_times, hist_V)
```

```
[9]: [<matplotlib.lines.Line2D at 0x2935bd6ff40>]
```



2.1.4 Example 2: Hodgkin–Huxley model

Another more complex example is the classical Hodgkin–Huxley neuron model. In HH model, four dynamical variables (`V, m, n, h`) are used for modeling the initiation and propagration of the action potential. Specificaly, they are governed by the following equations:

$$\begin{aligned}
C_m \frac{dV}{dt} &= -\bar{g}_\text{K} n^4 (V - V_K) - \bar{g}_\text{Na} m^3 h (V - V_{Na}) - \bar{g}_l (V - V_l) + I_{syn} \\
\frac{dm}{dt} &= \alpha_m(V)(1-m) - \beta_m(V)m \\
\frac{dh}{dt} &= \alpha_h(V)(1-h) - \beta_h(V)h \\
\frac{dn}{dt} &= \alpha_n(V)(1-n) - \beta_n(V)n
\end{aligned}$$

In BrainPy, such dynamical system can be coded as:

```
[10]: @bp.odeint(method='rk4', dt=0.01)
def integral(V, m, h, n, t, Iext, gNa, ENa, gK, EK, gL, EL, C):
    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

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

    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

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

    return dVdt, dmdt, dhdt, dndt
```

Same as the FHN model, we can also integrate the HH model in the given parameters and time interval:

```
[11]: Iext=10.;   ENa=50.;   EK=-77.;   EL=-54.387
C=1.0;      gNa=120.;  gK=36.;    gL=0.03
```

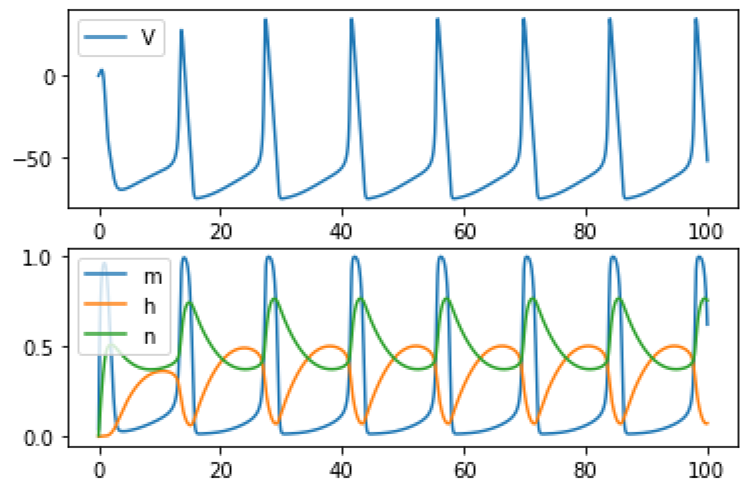
```
[12]: hist_times = np.arange(0, 100, integral.dt)
hist_V, hist_m, hist_h, hist_n = [], [], [], []
V, m, h, n = 0., 0., 0., 0.
for t in hist_times:
    V, m, h, n = integral(V, m, h, n, t, Iext,
                          gNa, ENa, gK, EK, gL, EL, C)
    hist_V.append(V)
    hist_m.append(m)
    hist_h.append(h)
    hist_n.append(n)
```

(continues on next page)

(continued from previous page)

```
plt.subplot(211)
plt.plot(hist_times, hist_V, label='V')
plt.legend()
plt.subplot(212)
plt.plot(hist_times, hist_m, label='m')
plt.plot(hist_times, hist_h, label='h')
plt.plot(hist_times, hist_n, label='n')
plt.legend()
```

```
[12]: <matplotlib.legend.Legend at 0x2935d5afd60>
```



2.2 SDEs

2.2.1 How to define SDE functions?

For a one-dimensional stochastic differentiable equation (SDE) with scalar Wiener noise, it is given by

$$dX_t = f\left(X_t, t, p_1\right) dt + g\left(X_t, t, p_2\right) dW_t \quad (1) \tag{2.4}$$

where $X_t = X(t)$ is the realization of a stochastic process or random variable, $f(X_t, t)$ is the drift coefficient, $g(X_t, t)$ denotes the diffusion coefficient, the stochastic process W_t is called Wiener process.

For this SDE system, we can define two Python funtions f and g to represent it.

```
[13]: def g_part(x, t, p1, p2):
          dg = g(x, t, p2)
          return dg

      def f_part(x, t, p1, p2):
          df = f(x, t, p1)
          return df
```

Same with the ODE functions, the arguments before t denotes the random variables, while the arguments defined after t represents the parameters. For the SDE function with scalar noise, the size of the return data dg and df should be the same. For example, $df \in R^d, dg \in R^d$.

However, for a more general SDE system, it usually has multi-dimensional driving Wiener process:

$$dX_t = f(X_t)dt + \sum_{\alpha=1}^{m} g_\alpha(X_t)dW_t^\alpha$$

For such m-dimensional noise system, the coding schema is the same with the scalar ones, but with the difference of that the data size of dg has one more dimension. For example, $df \in R^d, dg \in R^{d\times m}$.

2.2.2 How to define the numerical integration for SDEs?

Brefore the numerical integration of SDE functions, we should distinguish two kinds of SDE integrals. For the integration of system (1), we can get

$$X_t = X_{t_0} + \int_{t_0}^{t} f\left(X_s, s\right) ds + \int_{t_0}^{t} g\left(X_s, s\right) dW_s \quad (2) \tag{2.5}$$

In 1940s, the Japanese mathematician K. Ito denoted a type of integral called *Ito stochastic integral*. In 1960s, the Russian physicist R. L. Stratonovich proposed an other kind of stochastic integral called *Stratonovich stochastic integral* and used the symbol "$\circ$" to distinct it from the former Ito integral.

$$\begin{aligned} dX_t &= f\left(X_t, t\right) dt + g\left(X_t, t\right) \circ dW_t \\ X_t &= X_{t_0} + \int_{t_0}^{t} f\left(X_s, s\right) ds + \int_{t_0}^{t} g\left(X_s, s\right) \circ dW_s \quad (3) \end{aligned}$$

The difference of Ito integral (2) and Stratonovich integral (3) lies at the second integral term, which can be written in a general form as

$$\int_{t_0}^{t} g\left(X_s, s\right) dW_s = \lim_{h\to 0} \sum_{k=0}^{m-1} g\left(X_{\tau_k}, \tau_k\right)\left(W\left(t_{k+1}\right) - W\left(t_k\right)\right) \tag{2.6}$$

$$\text{where} \quad h = t_{k+1} - t_k \tag{2.7}$$

$$\tau_k = (1-\lambda)t_k + \lambda t_{k+1} \tag{2.8}$$

- In the stochastic integral of the Ito SDE, $\lambda = 0$, thus $\tau_k = t_k$;
- In the definition of the Stratonovich integral, $\lambda = 0.5$, thus $\tau_k = (t_{k+1} + t_k)/2$.

In BrainPy, these two different integrals can be easily implemented. What need the users do is to provide a keyword `sde_type` in decorator `bp.sdeint`. `sde_type` can be "bp.STRA_SDE" or "bp.ITO_SDE" (default). Also, the different type of Wiener process can also be easily distinguished by the `wiener_type` keyword. It can be "bp.SCALAR_WIENER" (default) or "bp.VECTOR_WIENER".

Now, let's numerically integrate the SDE (1) by the Ito way with the Milstein method:

[14]:
```python
def g_part(x, t, p1, p2):
    dg = g(x, t, p2)
    return dg  # shape=(d,)

@bp.sdeint(g=g_part, method='milstein')
def f_part(x, t, p1, p2):
    df = f(x, t, p1)
    return df  # shape=(d,)
```

Or, it can be expressed as:

[15]:
```
def g_part(x, t, p1, p2):
    dg = g(x, t, p2)
    return dg  # shape=(d,)

def f_part(x, t, p1, p2):
    df = f(x, t, p1)
    return df  # shape=(d,)

integral = bp.sdeint(f=f_part, g=g_part, method='milstein')
```

However, if you try to numerically integrate the SDE with multi-dimensional Wiener process by the Stratonovich ways, you can code it like this:

[16]:
```
def g_part(x, t, p1, p2):
    dg = g(x, t, p2)
    return dg  # shape=(d, m)

def f_part(x, t, p1, p2):
    df = f(x, t, p1)
    return df  # shape=(d,)

integral = bp.sdeint(f=f_part, g=g_part, method='milstein',
                     sde_type=bp.STRA_SDE,
                     wiener_type=bp.SCALAR_WIENER)
```

2.2.3 Example 3: Noisy Lorenz system

Here, let's demenstrate how to define a numerical solver for SDEs with the famous Lorenz system:

$$\frac{dx}{dt} = \sigma(y - x) + px * \xi_x$$
$$\frac{dy}{dt} = x(\rho - z) - y + py * \xi_y$$
$$\frac{dz}{dt} = xy - \beta z + pz * \xi_z$$

[17]:
```
sigma = 10; beta = 8/3;
rho = 28;   p = 0.1

def lorenz_g(x, y, z, t):
    return p * x, p * y, p * z

def lorenz_f(x, y, z, t):
    dx = sigma * (y - x)
    dy = x * (rho - z) - y
    dz = x * y - beta * z
    return dx, dy, dz

```

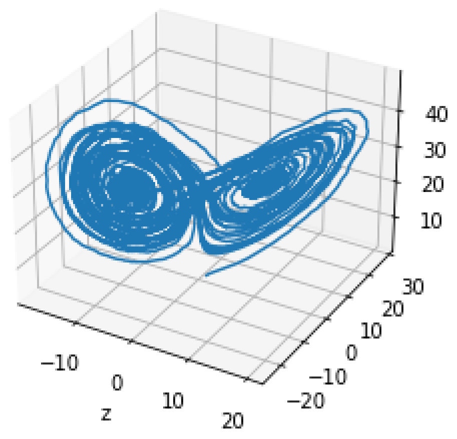
(continues on next page)

(continued from previous page)

```
lorenz = bp.sdeint(f=lorenz_f, g=lorenz_g,
                   sde_type=bp.ITO_SDE,
                   wiener_type=bp.SCALAR_WIENER,
                   dt=0.005)
```

[18]:
```
hist_times = np.arange(0, 50, lorenz.dt)
hist_x, hist_y, hist_z = [], [], []
x, y, z = 1., 1., 1.
for t in hist_times:
    x, y, z = lorenz(x, y, z, t)
    hist_x.append(x)
    hist_y.append(y)
    hist_z.append(z)

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot3D(hist_x, hist_y, hist_z)
ax.set_xlabel('x')
ax.set_xlabel('y')
ax.set_xlabel('z')
```

[18]:
```
Text(0.5, 0, 'z')
```



2.3 Backend-independent Property

Actually, BrainPy provides general numerical solvers for user-defined differential equations. It is backend-independent. Users can define their differential equations with any computation backend they prefer, such as NumPy, PyTorch, TensorFlow, Jax. The only thing need to do is to provide the necessary operations to `brainpy.ops`. For the needed operations, you can inspect them by

[19]:
```
bp.ops.OPS_FOR_SOLVER
```

[19]:
```
['normal', 'sum', 'exp', 'shape']
```

```
[20]: bp.ops.OPS_FOR_SIMULATION
```

```
[20]: ['as_tensor', 'zeros', 'ones', 'arange', 'concatenate', 'where', 'reshape']
```

After you define the necessary functions, you need to register them by `bp.ops.set_ops(**kwargs)`. Or, you can put all these functions into a ".py" file, then import this python script as a module and set `bp.ops.set_ops_from_module(module)`.

Currently, BrainPy inherently supports NumPy, Numba, PyTorch, TensorFlow. You can easily switch backend just by typing `bp.backend.set(backend_name)`. For example,

```
[21]: bp.backend.set('numpy')
```

switch the backend to NumPy.

```
[22]: bp.backend.set('numba')
```

switch the backend to Numba.

BrainPy also provides the interface for users to define the unified operations across various backends. For example, if you want to define a `clip` operation with the unified calling function `bp.ops.clip` under various bakcends, you can register this buffer by

```
[23]: # clip operation under "NumPy" backend

bp.ops.set_buffer('numpy', clip=np.clip)
```

```
[24]: # clip operation under "PyTorch" backend

import torch

bp.ops.set_buffer('pytorch', clip=torch.clamp)
```

```
[25]: # clip operation under "Numba" backend

import numba as nb

@nb.njit
def nb_clip(x, x_min, x_max):
    x = np.maximum(x, x_min)
    x = np.minimum(x, x_max)
    return x

bp.ops.set_buffer('numba', clip=nb_clip)
bp.ops.set_buffer('numba-parallel', clip=nb_clip)
```

After this setting and the backend switch calling `bp.backend.set(backend_name)`, BrainPy will automatically find the corresponding function with `bp.ops.clip` for the switched backend.

Author:

- Chaoming Wang
- Email: adaduo@outlook.com

- Date: 2021.03.25

CHAPTER

THREE

NEURODYNAMICS SIMULATION

Contents

For brain modeling, BrainPy provides the interface of `brainpy.NeuGroup`, `brainpy.TwoEndConn`, and `brainpy.Network` for convenient neurodynamics simulation.

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

3.1 brainpy.NeuGroup

`brainpy.NeuGroup` is used for neuron group modeling. User-defined neuron group models must inherit from the `brainpy.NeuGroup`. Let's take the leaky integrate-and-fire (LIF) model and Hodgkin–Huxley neuron model as the illustrated examples.

3.1.1 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.

As stated above, the numerical integration of the differential equation in LIF model can be coded as:

[2]:
```
@bp.odeint
def int_V(V, t, Iext, V_rest, R, tau):
    return (- (V - V_rest) + R * Iext) / tau
```

Then, we will define the following items to store the 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.

Based on these states, the updating logic of LIF model from the current time t to the next time $t + dt$ will be coded as:

[3]:
```
class LIF(bp.NeuGroup):
    target_backend = ['numpy', 'numba']

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

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

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

    @staticmethod
    @bp.odeint
    def int_V(V, t, Iext, V_rest, R, tau):
        return (- (V - V_rest) + R * Iext) / tau

    def update(self, _t):
        for i in range(self.size[0]):
            if _t - self.t_last_spike[i] <= self.t_refractory:
                self.refractory[i] = 1.
            else:
                self.refractory[0] = 0.
                V = self.int_V(self.V[i], _t, self.input[i], self.V_rest, self.R, self.
↪tau)
                if V >= self.V_th:
                    self.V[i] = self.V_reset
                    self.spike[i] = 1.
                    self.t_last_spike[i] = _t
                else:
                    self.spike[i] = 0.
                    self.V[i] = V
            self.input[i] = 0.
```

That's all, we have coded a LIF neuron model.

Each NeuGroup 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`).

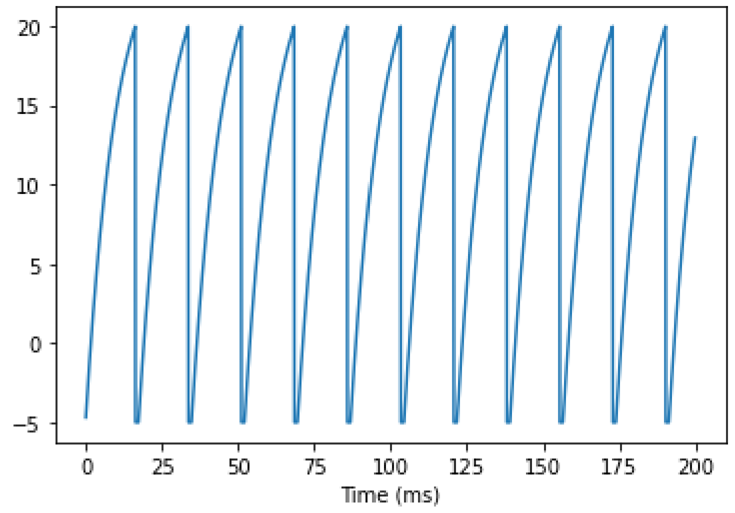
- `inputs`: Specify the inputs for each model component. With the format of `(target, value, [operation])`. The default operation is +, which means the input `value` will be added to the `target`. Or, the operation can be +, −, *, /, or =.

Now, let's run it.

[4]:
```
group = LIF(100, monitors=['V'])
```

[5]:
```
group.run(duration=200., inputs=('input', 26.), report=True)
bp.visualize.line_plot(group.mon.ts, group.mon.V, show=True)
```

```
Compilation used 0.0000 s.
Start running ...
Run 10.0% used 0.054 s.
Run 20.0% used 0.121 s.
Run 30.0% used 0.177 s.
Run 40.0% used 0.231 s.
Run 50.0% used 0.283 s.
Run 60.0% used 0.337 s.
Run 70.0% used 0.387 s.
Run 80.0% used 0.440 s.
Run 90.0% used 0.496 s.
Run 100.0% used 0.550 s.
Simulation is done in 0.550 s.
```



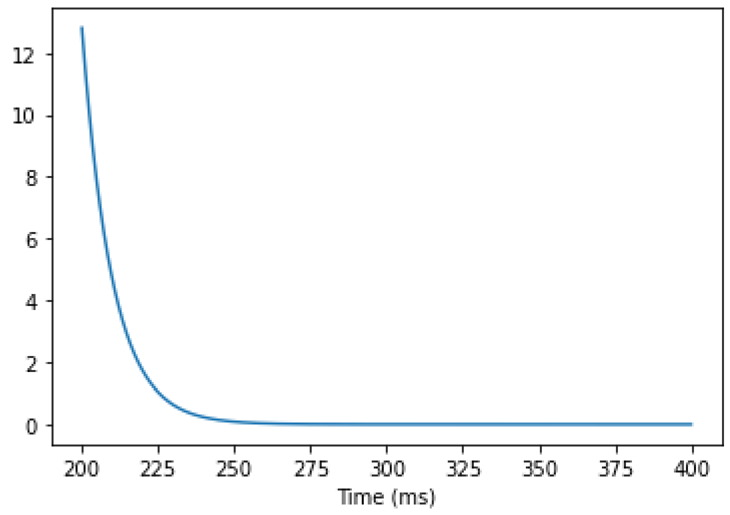
[6]:
```
group.run(duration=(200, 400.), report=True)
bp.visualize.line_plot(group.mon.ts, group.mon.V, show=True)
```

```
Compilation used 0.0010 s.
Start running ...
Run 10.0% used 0.052 s.
Run 20.0% used 0.108 s.
Run 30.0% used 0.161 s.
Run 40.0% used 0.214 s.
Run 50.0% used 0.273 s.
```

(continues on next page)

(continued from previous page)

```
Run 60.0% used 0.319 s.
Run 70.0% used 0.379 s.
Run 80.0% used 0.432 s.
Run 90.0% used 0.483 s.
Run 100.0% used 0.542 s.
Simulation is done in 0.542 s.
```



As you experienced just now, the benefit of inheriting `brainpy.NeuGroup` lies at the following several ways:

- Easy way to monitor variable trajectories.
- Powerful "inputs" support.
- Continuous running support.
- Progress report.

On the model definition, BrainPy endows you the fully data/logic flow control. You can define models with any data you need and any logic you want. There are little limitations/constrains on your customization. 1, you should set what computing backend do your defined model support by the keyword `target_backend`. 2, you should "super()" initialize the `brainpy.NeuGroup` with the keyword of the group `size`. 3, you should define the `update` function.

3.1.2 Hodgkin–Huxley model

The updating logic in the above LIF model is coded with a for loop, which is very suitable for Numba backend (because Numba is a Just-In-Time compiler, and it is good at the for loop optimization). However, for array-oriented programming languages, such as NumPy, PyTorch and TensorFlow, this coding schema is inefficient. Here, let's use the HH neuron model as example to demonstrate how to code an array-based neuron model for general backends.

[7]:
```
class HH(bp.NeuGroup):
    target_backend = 'general'

    @staticmethod
    def diff(V, m, h, n, t, Iext, gNa, ENa, gK, EK, gL, EL, C):
        alpha = 0.1 * (V + 40) / (1 - bp.ops.exp(-(V + 40) / 10))
        beta = 4.0 * bp.ops.exp(-(V + 65) / 18)
```

(continues on next page)

(continued from previous page)

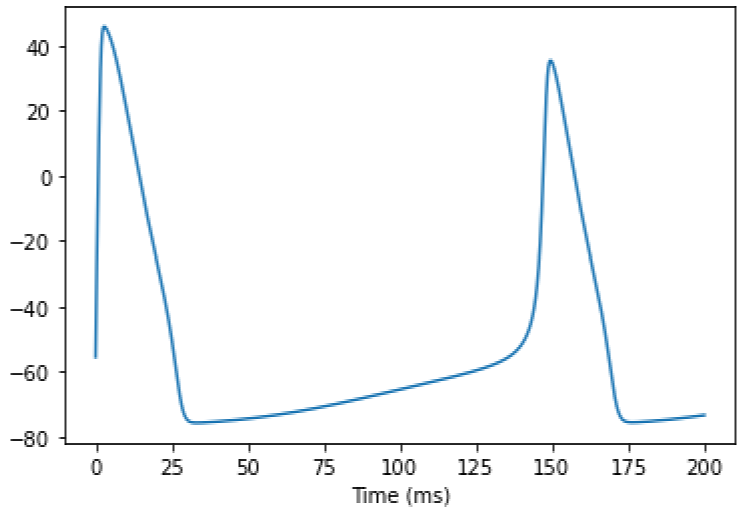
```
        dmdt = alpha * (1 - m) - beta * m

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

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

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

        return dVdt, dmdt, dhdt, dndt

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

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

        self.integral = bp.odeint(f=self.diff, method='rk4', dt=0.01)
        super(HH, self).__init__(size=size, **kwargs)

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

In HH example, all the operations (including “zeros”, “ones” and “exp”) are used from the `brainpy.ops` as `bp.ops`.

zeros, bp.ops.ones and bp.ops.exp. What's more, we set the "target_backend" as general, which means it can run on any backends. So, let's try to run this model on various backends.

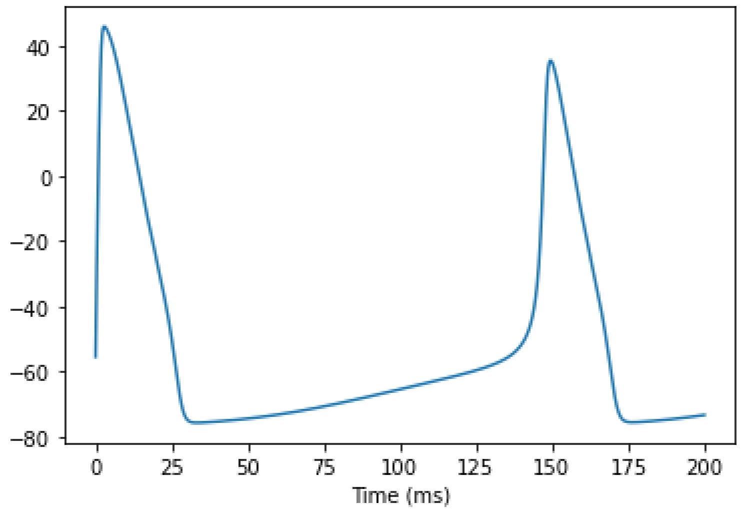
First is PyTorch.

[8]:
```
bp.backend.set('pytorch')

group = HH(100, monitors=['V'])
group.run(200., inputs=('input', 10.))
bp.visualize.line_plot(group.mon.ts, group.mon.V, show=True)
```



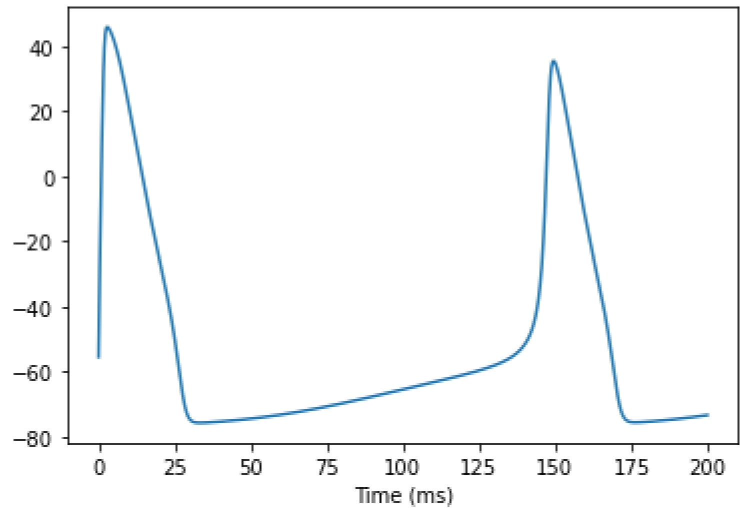
Second is NumPy.

[9]:
```
bp.backend.set('numpy')

group = HH(100, monitors=['V'])

group.run(200., inputs=('input', 10.))

bp.visualize.line_plot(group.mon.ts, group.mon.V, show=True)
```



The last is Numba.

```
[10]: bp.backend.set('numba')

group = HH(100, monitors=['V'])
group.run(200., inputs=('input', 10.))
bp.visualize.line_plot(group.mon.ts, group.mon.V, show=True)
```



3.2 brainpy.TwoEndConn

For synaptic connections, BrainPy provides `brainpy.TwoEndConn` to help you construct the projection between pre-synaptic and post-synaptic neuron groups, and provides `brainpy.connect.Connector` for synaptic connectivity between pre- and post- groups.

- The benefit of using `brainpy.TwoEndConn` lies at the **automatical synaptic delay**. The synapse modeling usually includes a delay time (typically 0.3–0.5 ms) required for a neurotransmitter to be released from a presynaptic membrane, diffuse across the synaptic cleft, and bind to a receptor site on the post-synaptic membrane. BrainPy provides `register_constant_dely()` for automatical state delay.
- Another benefit of using `brainpy.connect.Connector` lies at the **connectivity structure construction**. `brainpy.connect.Connector` provides various synaptic structures, like "pre_ids", "post_ids", "conn_mat", "pre2post", "post2pre", "pre2syn", "post2syn", "pre_slice", and "post_slice". Users can "requires" such data structures by calling `connector.requires('pre_ids', 'post_ids', ...)`. We will detail this function in *Synaptic Connections*.

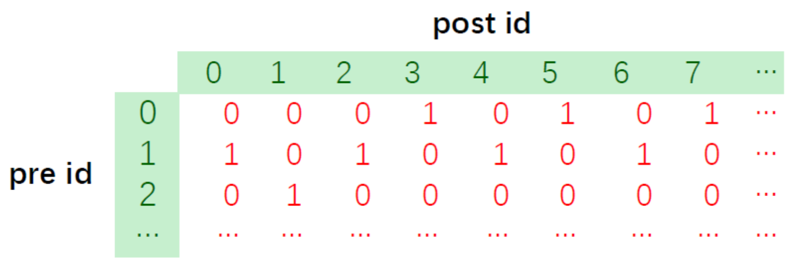
Here, let's illustrate how to use `brainpy.TwoEndConn` with the AMPA synapse model.

3.2.1 AMPA Synapse Model

[11]:
```
class AMPA(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

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

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

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

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

    @staticmethod
    @bp.odeint(dt=0.01)
    def int_s(s, t, tau):
        return - s / tau

    def update(self, _t):
        self.s = self.int_s(self.s, _t, self.tau)
        for i in range(self.pre.size[0]):
            if self.pre.spike[i] > 0:
                self.s[i] += self.conn_mat[i]
        self.g.push(self.g_max * self.s)
        g = self.g.pull()
        self.post.input -= bp.ops.sum(g, axis=0) * (self.post.V - self.E)
```

To define a two-end synaptic projection is very much like the NeuGroup. Users need to inherit the `brainpy.TwoEndConn`, and provide the "target_backend" specification, "update" function and then "super()" initialize the parent class. But what different are two aspects: 1. connection. We need construct the synaptic connectivity by "connector.requires". 2. delay. We can register a constant delay variable by "self.register_constant_delay()".

Here, we create a matrix-based connectivity (with the shape of `(num_pre, num_post)`).



post id

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

And then register a delay variable "self.g" with the shape of `(num_pre, num_post)`.

3.3 brainpy.Network

Now, let's put the above defined HH model and AMPA synapse together to construct a network with `brainpy.Network`.

[12]:
```
bp.backend.set('numpy')
```

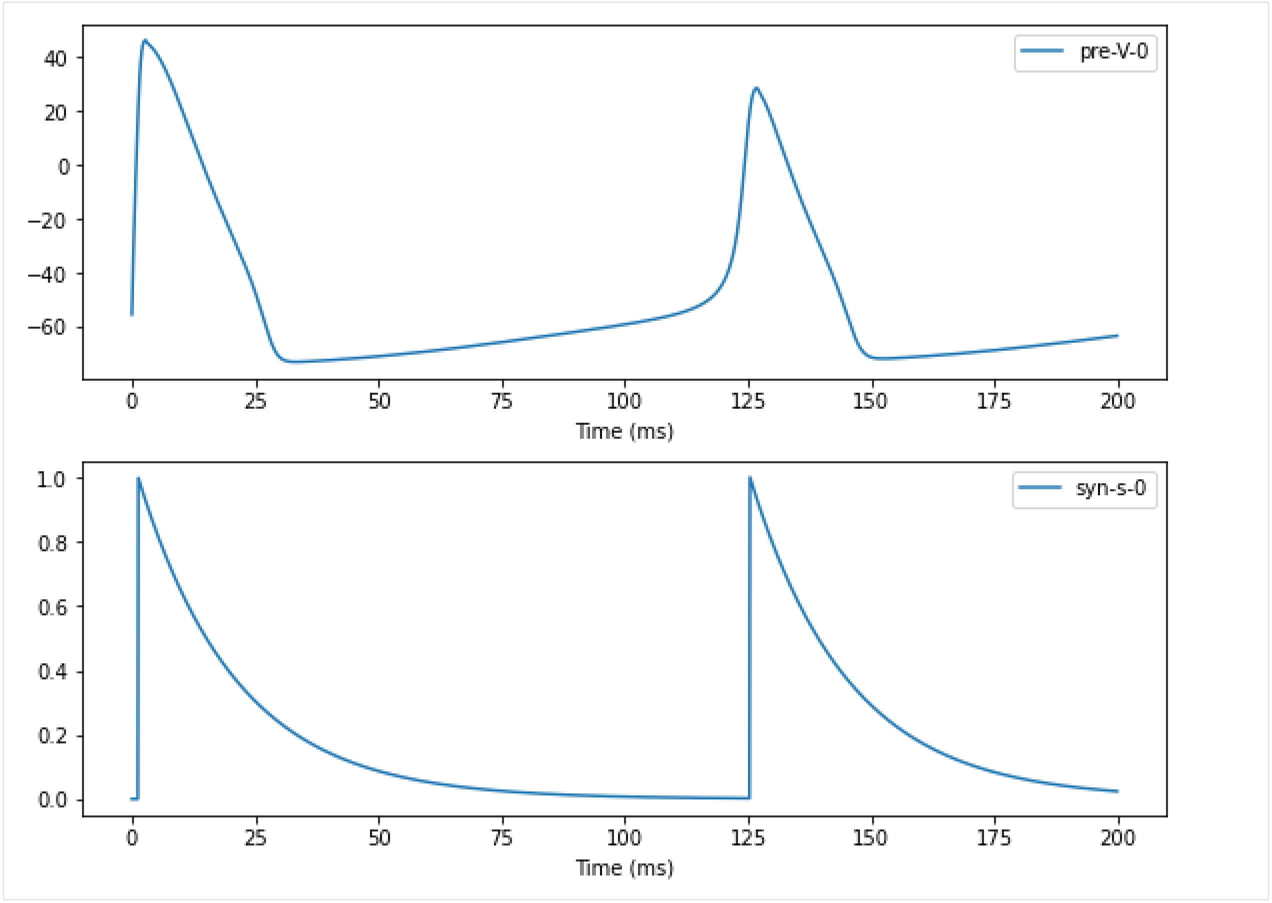
[13]:
```
group = HH(10, monitors=['V', 'spike'])
syn = AMPA(pre=group, post=group, conn=bp.connect.All2All(), delay=1.5, monitors=['s'])
```

[14]:
```
net = bp.Network(group, syn)
net.run(duration=200., inputs=(group, "input", 20.), report=True)
```

```
Compilation used 0.3254 s.
Start running ...
Run 10.0% used 0.060 s.
Run 20.0% used 0.120 s.
Run 30.0% used 0.180 s.
Run 40.0% used 0.250 s.
Run 50.0% used 0.310 s.
Run 60.0% used 0.370 s.
Run 70.0% used 0.420 s.
Run 80.0% used 0.490 s.
Run 90.0% used 0.547 s.
Run 100.0% used 0.611 s.
Simulation is done in 0.611 s.
```

[14]:
```
0.6110615730285645
```

[15]:
```
fig, gs = bp.visualize.get_figure(2, 1, 3, 8)

fig.add_subplot(gs[0, 0])
bp.visualize.line_plot(group.mon.ts, group.mon.V, legend='pre-V')

fig.add_subplot(gs[1, 0])
bp.visualize.line_plot(syn.mon.ts, syn.mon.s, legend='syn-s', show=True)
```



3.4 Backend-independent Property

Neurodynamics simulation in BrainPy has characteristics of high portability. It's also backend-independent. Currently, BrainPy inherently supports the tensor-oriented backends (such like NumPy, PyTorch, TensorFlow), and the JIT compilers (like Numba). After model coding, users can switch the backend easily by using `brainpy.backend.set(backend_name)`:

[16]:
```
# deploy all models to NumPy backend

bp.backend.set('numpy')
```

[17]:
```
# deploy all models to Numba CPU backend

bp.backend.set('numba')
```

Moreover, customize your preferred backend is also easy.

[18]:
```
bp.ops.OPS_FOR_SIMULATION
```

[18]:
```
['as_tensor', 'zeros', 'ones', 'arange', 'concatenate', 'where', 'reshape']
```

[19]:
```
bp.drivers.set_buffer('numpy',
                      node_driver=bp.drivers.GeneralNodeDriver,
                      net_driver=bp.drivers.GeneralNetDriver,
                      diffint_driver=bp.drivers.GeneralDiffIntDriver)
```

Author:

- Chaoming Wang
- Email: adaduo@outlook.com
- Date: 2021.03.25, update at 2021.05.29

CHAPTER

FOUR

NEURODYNAMICS ANALYSIS

Contents:

In addition to the flexible and effecient *neurodynamics simulation*, another ambition of BrainPy is to provide an integrative platform for **neurodynamics analysis**.

As is known to us all, dynamics analysis is necessary in neurodynamics. This is because blind simulation of nonlinear systems is likely to produce few results or misleading results. For example, attractors and repellors can be easily obtained through simulation by time forward and backward, while saddles can be hard to find.

Currently, BrainPy supports neurodynamics analysis for low-dimensional dynamical systems. Specifically, BrainPy provides the following methods for dynamics analysis:

1. phase plane analysis for one-dimensional and two-dimensional systems;
2. codimension one and codimension two bifurcation analysis;
3. bifurcation analysis of the fast-slow system.

In this section, I will illustrate how to do neuron dynamics analysis in BrainPy and how BrainPy implements it.

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

4.1 Phase Plane Analysis

Here, I will illustrate how to do phase plane analysis by using a well-known neuron model FitzHugh-Nagumo model.

FitzHugh-Nagumo model

The FitzHugh-Nagumo 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.

[2]:
```
a=0.7;  b=0.8;  tau=12.5;  Vth=1.9

@bp.odeint
def int_fhn(V, w, t, Iext):
    dw = (V + a - b * w) / tau
    dV = V - V * V * V / 3 - w + Iext
    return dV, dw
```

Phase Plane Analysis is implemented in `brainpy.analysis.PhasePlane`. It receives the following parameters:

- `integrals`: The integral functions to be analysis.
- `target_vars`: The variables to be analuzed. It must a dictionary with the format of `{var: variable range}`.
- `fixed_vars`: The variables to be fixed (optional).
- `pars_update`: Parameters to update (optional).

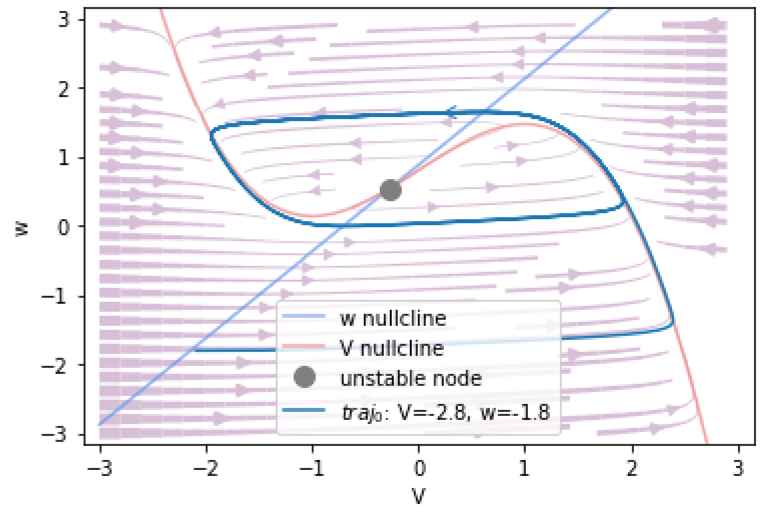
`brainpy.analysis.PhasePlane` provides interface to analyze the system's

- **nullcline**: The zero-growth isoclines, such as $g(x, y) = 0$ and $g(x, y) = 0$.
- **fixed points**: The equilibrium points of the system, which are located at all of the nullclines intersect.
- **vector filed**: The vector field of the system.
- **Trajectory**: A given simulation trajectory with the fixed variables.

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

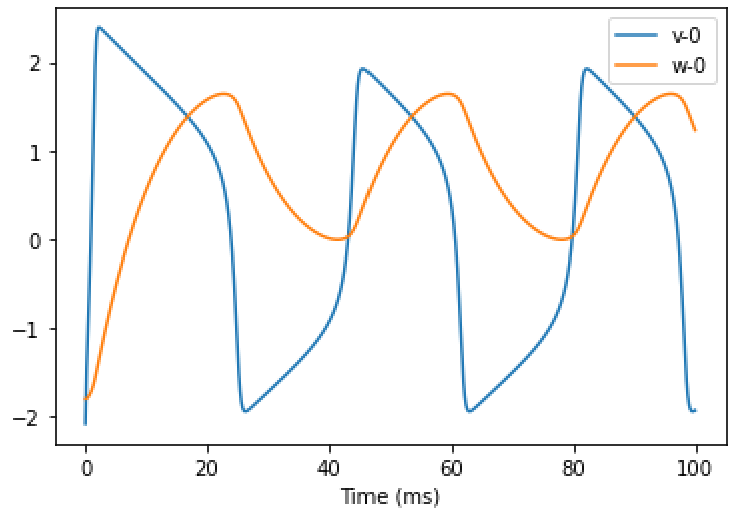
[3]:
```
analyzer = bp.analysis.PhasePlane(
    integrals=int_fhn,
    target_vars={'V': [-3, 3], 'w': [-3., 3.]},
    pars_update={'Iext': 0.8})
analyzer.plot_nullcline()
analyzer.plot_vector_field()
analyzer.plot_fixed_point()
analyzer.plot_trajectory([{'V': -2.8, 'w': -1.8}],
                         duration=100.,
                         show=True)
```

```
plot nullcline ...
SymPy solve "int_fhn(V, w) = 0" to "w = f(V, )", success.
SymPy solve "int_fhn(V, w) = 0" to "w = f(V, )", success.
plot vector field ...
plot fixed point ...
SymPy solve derivative of "int_fhn(V, w)" by "V", success.
SymPy solve derivative of "int_fhn(V, w)" by "w", success.
SymPy solve derivative of "int_fhn(V, w)" by "V", success.
SymPy solve derivative of "int_fhn(V, w)" by "w", success.
Fixed point #1 at V=-0.27290095899729705, w=0.5338738012533786 is a unstable node.
plot trajectory ...
```



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.

[4]:
```
class FHN(bp.NeuGroup):
    target_backend = 'numpy'

    def __init__(self, num, **kwargs):
        self.V = np.ones(num) * -2.8
        self.w = np.ones(num) * -1.8
        self.Iext = np.zeros(num)
        super(FHN, self).__init__(size=num, **kwargs)

    def update(self, _t):
        self.V, self.w = int_fhn(self.V, self.w, _t, self.Iext)


group = FHN(1, monitors=['V', 'w'])
group.run(100., inputs=('Iext', 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)
```



Note that the `fixed_vars` can be used to specify the neuron model's state `ST`, it can also be used to specify the functional arguments in integrators (like the `Iext` in `int_v()`).

4.2 Bifurcation Analysis

Bifurcation analysis is implemented within `brainpy.analysis.Bifurcation`. Which support codimension-1 and codimension-2 bifurcation analysis. Specifically, it receives the following parameter settings:

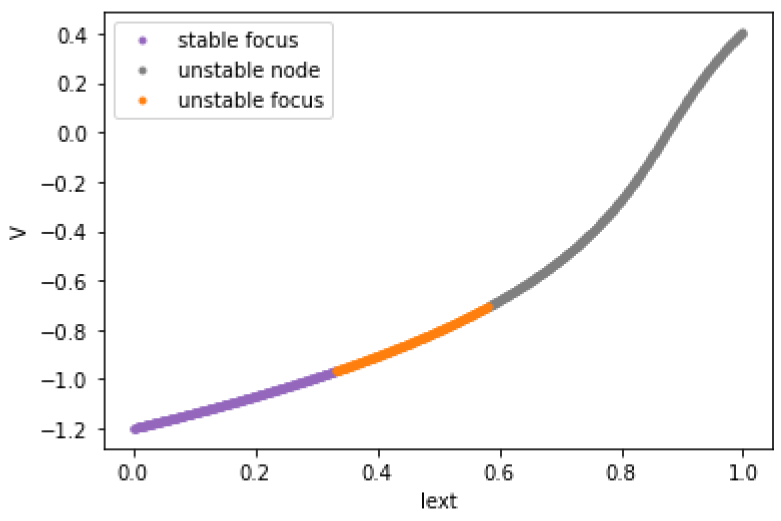
- `integrals`: The integral functions to be analysis.
- `target_pars`: The target parameters. Must be a dictionary with the format of `{par: parameter range}`.
- `target_vars`: The target variables. Must be a dictionary with the format of `{var: variable range}`.
- `fixed_vars`: The fixed variables.
- `pars_update`: The parameters to update.

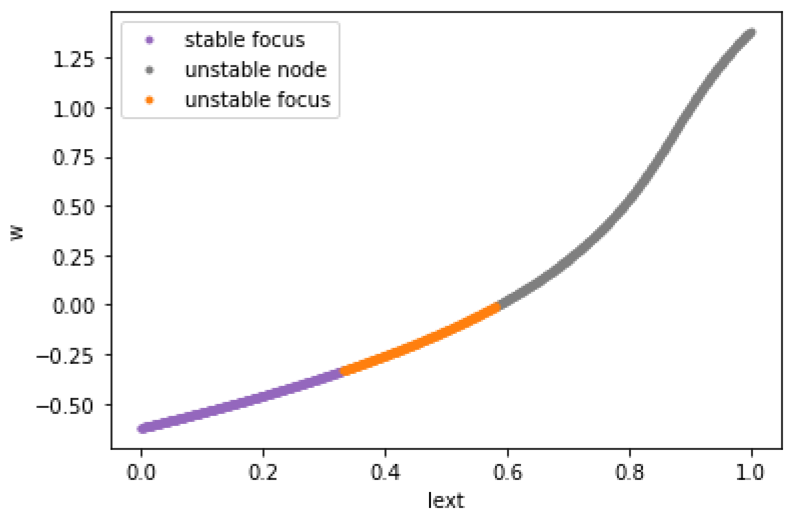
Codimension 1 bifurcation analysis

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.

[5]:
```
analyzer = bp.analysis.Bifurcation(
    integrals=int_fhn,
    target_pars={'Iext': [0., 1.]},
    target_vars={'V': [-3, 3], 'w': [-3., 3.]},
    numerical_resolution=0.001,
)
res = analyzer.plot_bifurcation(show=True)
```

```
plot bifurcation ...
SymPy solve "int_fhn(V, w, Iext) = 0" to "w = f(V, Iext)", success.
SymPy solve derivative of "int_fhn(V, w, Iext)" by "V", success.
SymPy solve derivative of "int_fhn(V, w, Iext)" by "w", success.
SymPy solve derivative of "int_fhn(V, w, Iext)" by "V", success.
SymPy solve derivative of "int_fhn(V, w, Iext)" by "w", success.
```





Codimension 2 bifurcation analysis

We simulaneously change I_{ext} and parameter a.

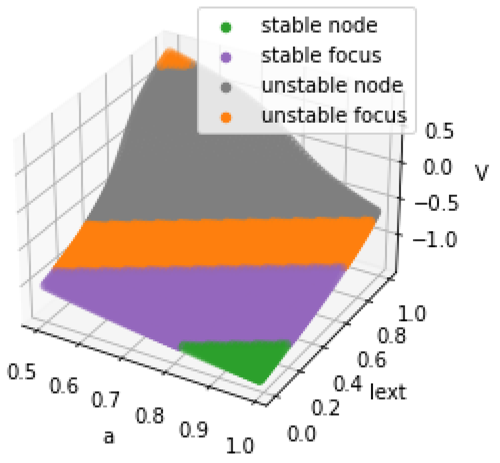
[6]:
```
from collections import OrderedDict

analyzer = bp.analysis.Bifurcation(
    integrals=int_fhn,
    target_pars=OrderedDict(a=[0.5, 1.], Iext=[0., 1.]),
    target_vars=OrderedDict(V=[-3, 3], w=[-3., 3.]),
    numerical_resolution=0.01,
)
res = analyzer.plot_bifurcation(show=True)
```

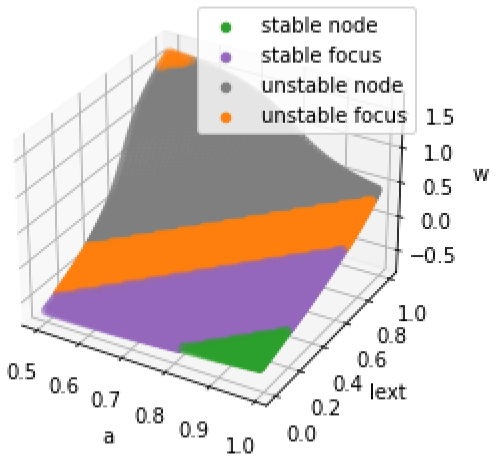
```
plot bifurcation ...
SymPy solve "int_fhn(V, w, a, Iext) = 0" to "w = f(V, a,Iext)", success.
SymPy solve derivative of "int_fhn(V, w, a, Iext)" by "V", success.
SymPy solve derivative of "int_fhn(V, w, a, Iext)" by "w", success.
SymPy solve derivative of "int_fhn(V, w, a, Iext)" by "V", success.
```

(continues on next page)

(continued from previous page)

```
SymPy solve derivative of "int_fhn(V, w, a, Iext)" by "w", success.
```





4.3 Fast-Slow System Bifurcation

BrainPy also provides a tool for fast-slow system bifurcation analysis by using `brainpy.analysis.FastSlowBifurcation`. This method is proposed by John Rinzel [1, 2, 3]. (J Rinzel, 1985, 1986, 1987) proposed that in a fast-slow dynamical system, we can treat the slow variables as the bifurcation parameters, and then study how the different value of slow variables affect the bifurcation of the fast sub-system.

`brainpy.analysis.FastSlowBifurcation` is very usefull in the bursting neuron analysis. I will illustrate this by using the Hindmarsh-Rose model. The Hindmarsh–Rose model of neuronal activity is aimed to study the spiking-bursting behavior of the membrane potential observed in experiments made with a single neuron. Its dynamics are governed by:

$$\frac{dV}{dt} = y - aV^3 + bV^2 - z + I \tag{4.1}$$

$$\frac{dy}{dt} = c - dV^2 \tag{4.2}$$

$$\frac{dz}{dt} = r(s(V - V_{rest}) \tag{4.3}$$

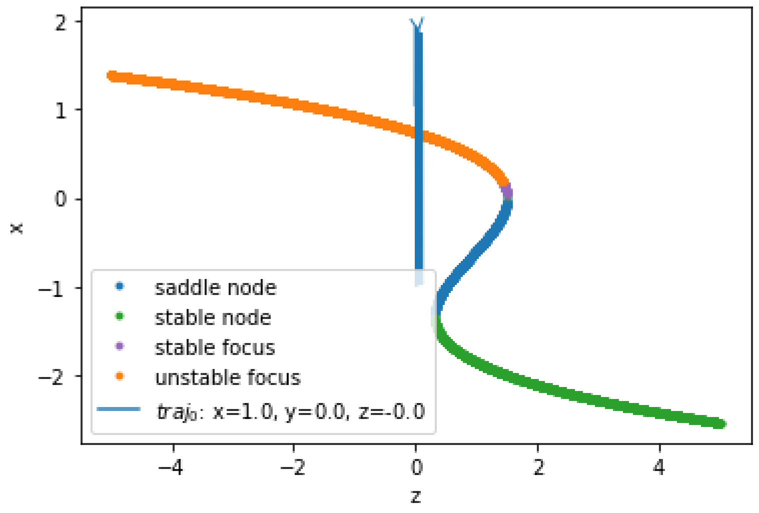
First of all, let's define the Hindmarsh–Rose model with BrainPy.

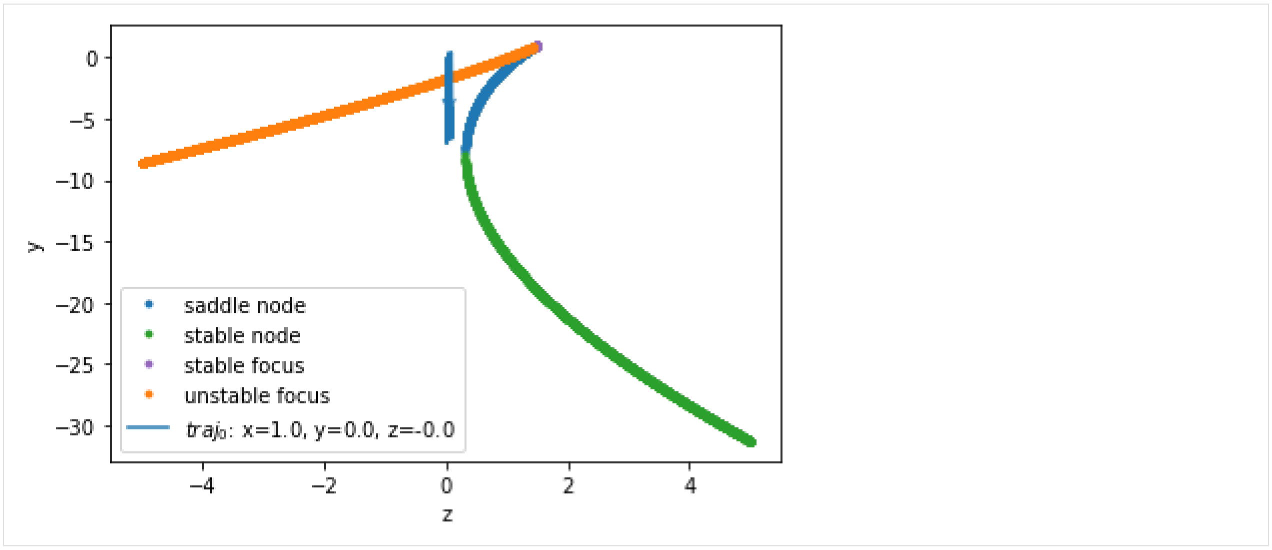
[7]:
```
a = 1.; b = 3.; c = 1.; d = 5.; s = 4.
x_r = -1.6; r = 0.001; Vth = 1.9

@bp.odeint(method='rk4', dt=0.02)
def int_hr(x, y, z, t, Isyn):
    dx = y - a * x ** 3 + b * x * x - z + Isyn
    dy = c - d * x * x - y
    dz = r * (s * (x - x_r) - z)
    return dx, dy, dz
```

We now can start to analysis the underlying bifurcation mechanism.

[8]:
```
analyzer = bp.analysis.FastSlowBifurcation(
    integrals=int_hr,
    fast_vars={'x': [-3, 3], 'y': [-10., 5.]},
    slow_vars={'z': [-5., 5.]},
    pars_update={'Isyn': 0.5},
    numerical_resolution=0.001
)
analyzer.plot_bifurcation()
analyzer.plot_trajectory([{'x': 1., 'y': 0., 'z': -0.0}],
                         duration=100.,
                         show=True)
```

```
plot bifurcation ...
SymPy solve "int_hr(x, y, z) = 0" to "y = f(x, z)", success.
SymPy solve derivative of "int_hr(x, y, z)" by "x", success.
SymPy solve derivative of "int_hr(x, y, z)" by "y", success.
SymPy solve derivative of "int_hr(x, y, z)" by "x", success.
SymPy solve derivative of "int_hr(x, y, z)" by "y", success.
plot trajectory ...
```





References:

[1] Rinzel, John. "Bursting oscillations in an excitable membrane model." In Ordinary and partial differential equations, pp. 304-316. Springer, Berlin, Heidelberg, 1985.

[2] Rinzel, John , and Y. S. Lee . On Different Mechanisms for Membrane Potential Bursting. Nonlinear Oscillations in Biology and Chemistry. Springer Berlin Heidelberg, 1986.

[3] Rinzel, John. "A formal classification of bursting mechanisms in excitable systems." In Mathematical topics in population biology, morphogenesis and neurosciences, pp. 267-281. Springer, Berlin, Heidelberg, 1987.

Author:

- Chaoming Wang
- Email: adaduo@outlook.com
- Date: 2021.03.25

CHAPTER

FIVE

SYNAPTIC CONNECTIVITY

Contents

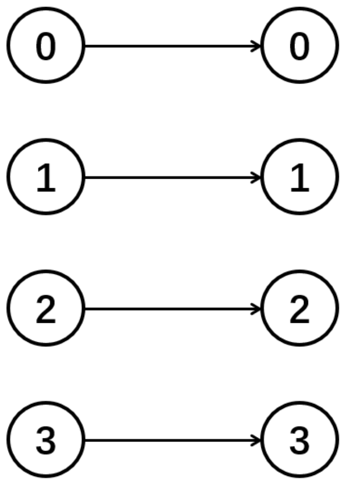
BrainPy provides several commonly used connection methods in `brainpy.connect` module (see the follows). They are all inherited from the base class `brainpy.connect.Connector`. Users can also customize their synaptic connectivity by the class inheritance.

[1]:
```
import brainpy as bp

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

5.1 Build-in regular connections

5.1.1 brainpy.connect.One2One

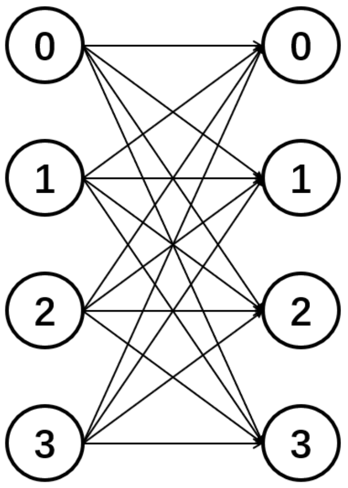
The neurons in the pre-synaptic neuron group only connect to the neurons in the same position of the post-synaptic group. Thus, this connection requires the indices of two neuron groups same. Otherwise, an error will occurs.



[2]:
```
conn = bp.connect.One2One()
```

5.1.2 brainpy.connect.All2All

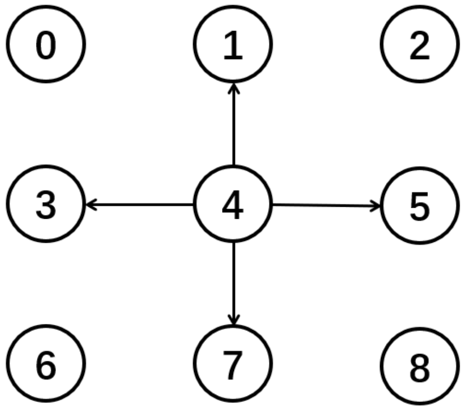
All neurons of the post-synaptic population form connections with all neurons of the pre-synaptic population (dense connectivity). Users can choose whether connect the neurons at the same position (`include_self=True or False`).



[3]:
```
conn = bp.connect.All2All(include_self=False)
```

5.1.3 brainpy.connect.GridFour

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

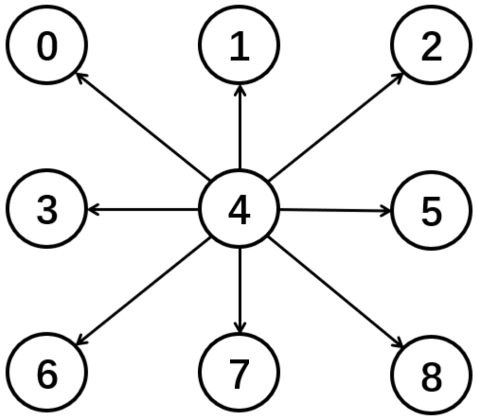


grid_four

```
[4]: conn = bp.connect.GridFour(include_self=False)
```

5.1.4 brainpy.connect.GridEight

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

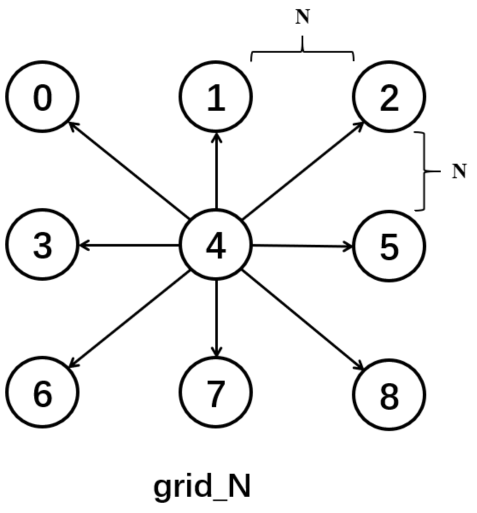


grid_eight

```
[5]: conn = bp.connect.GridEight(include_self=False)
```

5.1.5 brainpy.connect.GridN

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

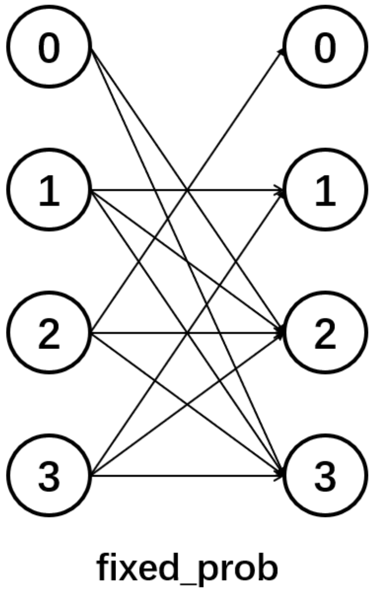


```
[6]: conn = bp.connect.GridN(N=2, include_self=False)
```

5.2 Build-in random connections

5.2.1 brainpy.connect.FixedProb

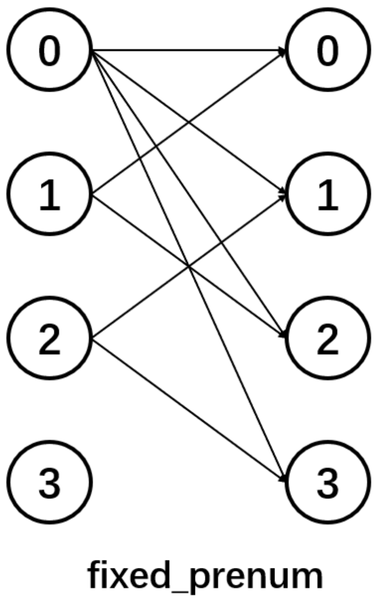
For each post-synaptic neuron, there is a fixed probability that it forms a connection with a neuron of the pre-synaptic population. It is basically a all_to_all projection, except some synapses are not created, making the projection sparser.



```
[7]: conn = bp.connect.FixedProb(prob=0.5, include_self=False, seed=1234)
```

5.2.2 brainpy.connect.FixedPreNum

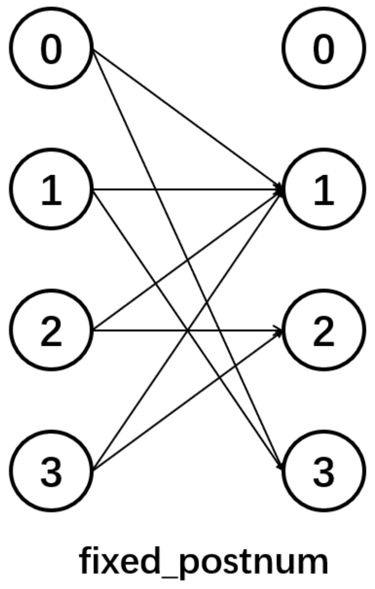
Each neuron in the post-synaptic population receives connections from a fixed number of neurons of the pre-synaptic population chosen randomly. It may happen that two post-synaptic neurons are connected to the same pre-synaptic neuron and that some pre-synaptic neurons are connected to nothing.



```
[8]: conn = bp.connect.FixedPreNum(num=10, include_self=True, seed=1234)
```

5.2.3 brainpy.connect.FixedPostNum

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.



```
[9]: conn = bp.connect.FixedPostNum(num=10, include_self=True, seed=1234)
```

5.2.4 brainpy.connect.GaussianProb

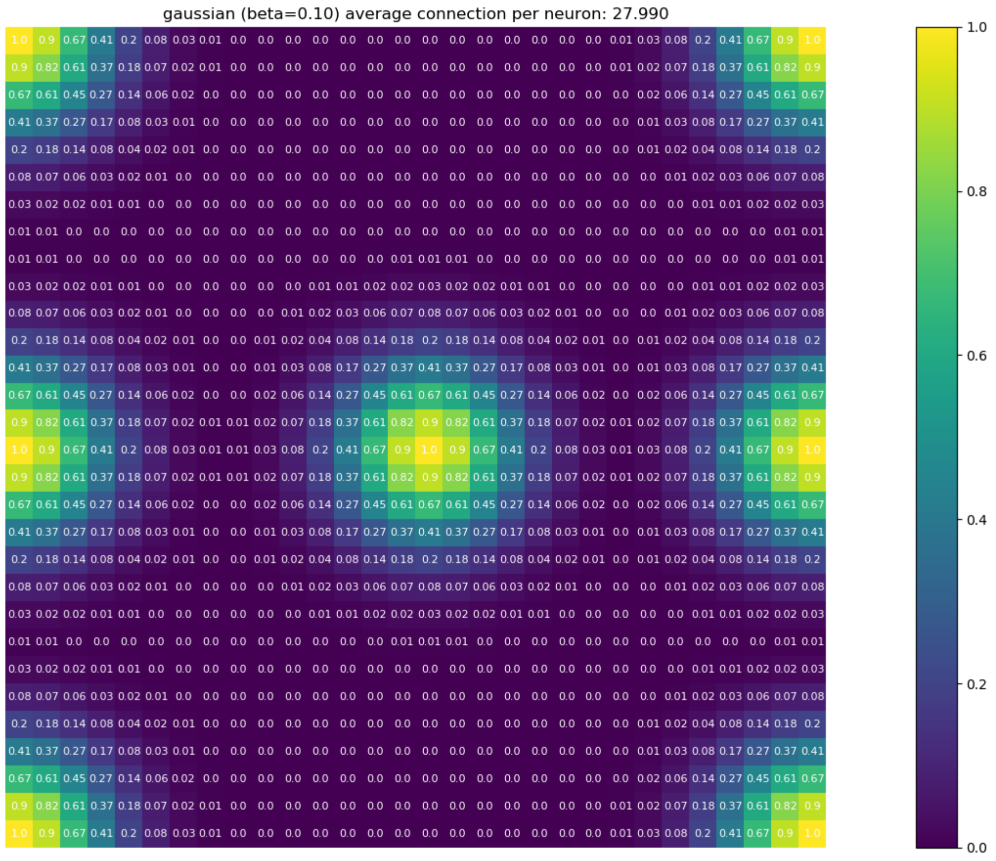
Builds a Gaussian connection pattern between the two populations, where the connection probability decay according to the gaussian function.

Specifically,

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

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:



```
[10]: conn = bp.connect.GaussianProb(sigma=0.2, p_min=0.01, normalize=True, include_self=True,
      ↪seed=1234)
```

5.2.5 brainpy.connect.GaussianWeight

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

Specifically,

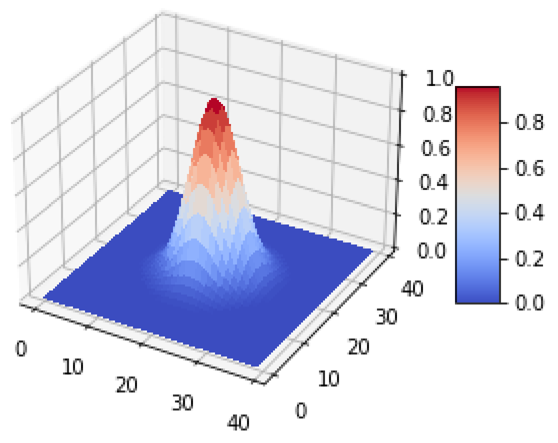
$$w(x, y) = w_{max} \cdot \exp\left(-\frac{(x - x_c)^2 + (y - y_c)^2}{2\sigma^2}\right)$$

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 $0.01 w_{max}$.

[11]:
```
def show_weight(pre_ids, post_ids, weights, geometry, neu_id):
    height, width = geometry
    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(geometry)
    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()
```

[12]:
```
conn = bp.connect.GaussianWeight(sigma=0.1, w_max=1., w_min=0.01,
                                 normalize=True, include_self=True)
```

[13]:
```
pre_geom = post_geom = (40, 40)
conn(pre_geom, post_geom)

pre_ids = conn.pre_ids
post_ids = conn.post_ids
weights = conn.weights
show_weight(pre_ids, post_ids, weights, pre_geom, 820)
```



5.2.6 brainpy.connect.DOG

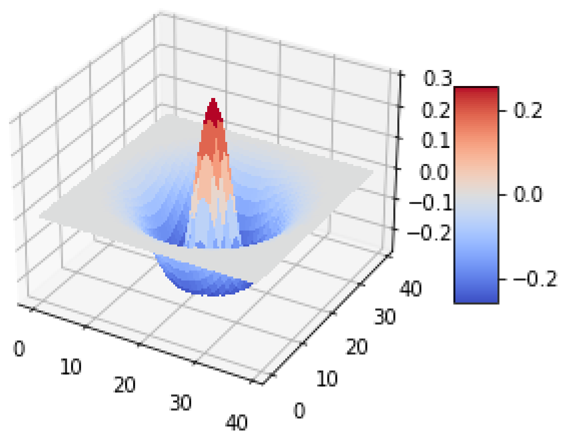
Builds a Difference-Of-Gaussian (dog) connection pattern between the two populations.

Mathematically,

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

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).

[14]:
```
dog = bp.connect.DOG(sigmas=(0.08, 0.15), ws_max=(1.0, 0.7), w_min=0.01,
                     normalize=True, include_self=True)
h = 40
pre_geom = post_geom = (h, h)
dog(pre_geom, post_geom)

pre_ids = dog.pre_ids
post_ids = dog.post_ids
weights = dog.weights
show_weight(pre_ids, post_ids, weights, (h, h), h * h // 2 + h // 2)
```



5.2.7 brainpy.connect.SmallWorld

SmallWorld is a connector class to help build a small-world network [1]. small-world network is defined to be a network where the typical distance L between two randomly chosen nodes (the number of steps required) grows proportionally to the logarithm of the number of nodes N in the network, that is:

$$L \propto \log N$$

[1] Duncan J. Watts and Steven H. Strogatz, Collective dynamics of small-world networks, Nature, 393, pp. 440–442, 1998.

Currently, SmallWorld only support a one-dimensional network with the ring structure. It receives four settings:

- num_neighbor: the number of the nearest neighbors to connect.
- prob: the probability of rewiring each edge.
- directed: whether the edge is the directed (“directed=True”) or undirected (“directed=False”) connection.
- include_self: whether allow to connect to itself.

[15]:
```
conn = bp.connect.SmallWorld(num_neighbor=5, prob=0.2, directed=False, include_
↪self=False)
```

5.2.8 brainpy.connect.ScaleFreeBA

ScaleFreeBA is a connector class to help build a random scale-free network according to the Barabási–Albert preferential attachment model [2]. ScaleFreeBA receives the following settings:

- m: Number of edges to attach from a new node to existing nodes.
- directed: whether the edge is the directed (“directed=True”) or undirected (“directed=False”) connection.
- seed: Indicator of random number generation state.

[2] A. L. Barabási and R. Albert “Emergence of scaling in random networks”, Science 286, pp 509-512, 1999.

[16]:
```
conn = bp.connect.ScaleFreeBA(m=5, directed=False, seed=12345)
```

5.2.9 brainpy.connect.ScaleFreeBADual

`ScaleFreeBADual` is a connector class to help build a random scale-free network according to the dual Barabási–Albert preferential attachment model [3]. ScaleFreeBA receives the following settings:

- `p`: The probability of attaching m_1 edges (as opposed to m_2 edges).
- `m1` : Number of edges to attach from a new node to existing nodes with probability p.
- `m2`: Number of edges to attach from a new node to existing nodes with probability $1 - p$.
- `directed`: whether the edge is the directed ("directed=True") or undirected ("directed=False") connection.
- `seed`: Indicator of random number generation state.

[3] N. Moshiri. "The dual-Barabasi-Albert model", arXiv:1810.10538.

[17]:
```
conn = bp.connect.ScaleFreeBADual(m1=3, m2=5, p=0.5, directed=False, seed=12345)
```

5.2.10 brainpy.connect.PowerLaw

`PowerLaw` is a connector class to help build a random graph with powerlaw degree distribution and approximate average clustering [4]. It receives the following settings:

- `m` : the number of random edges to add for each new node
- `p` : Probability of adding a triangle after adding a random edge
- `directed`: whether the edge is the directed ("directed=True") or undirected ("directed=False") connection.
- `seed` : Indicator of random number generation state.

[4] P. Holme and B. J. Kim, "Growing scale-free networks with tunable clustering", Phys. Rev. E, 65, 026107, 2002.

[18]:
```
conn = bp.connect.PowerLaw(m=3, p=0.5, directed=False, seed=12345)
```

5.3 Customize your connections

BrainPy also allows you to customize your model connections. What need users do is only two aspects:

- Your connection class should inherit `brainpy.connect.Connector`.
- Initialize the `conn_mat` or `pre_ids`+ `post_ids` synaptic structures.
- Provide `num_pre` and `num_post` information.

In such a way, based on this customized connection class, users can generate any other synaptic structures (such like `pre2post`, `pre2syn`, `pre_slice_syn`, etc.) easily.

Here, let's take a simple connection as an example. In this example, we create a connection method which receives users' handful index projection.

[19]:
```
class IndexConn(bp.connect.Connector):
    def __init__(self, i, j):
        super(IndexConn, self).__init__()

        # initialize the class via "pre_ids" and "post_ids"
        self.pre_ids = bp.ops.as_tensor(i)
```

(continues on next page)

(continued from previous page)

```
        self.post_ids = bp.ops.as_tensor(j)

    def __call__(self, pre_size, post_size):
        self.num_pre = bp.size2len(pre_size)  # this is ncessary when create "pre2post" ,
                                              # "pre2syn"  etc. structures
        self.num_post = bp.size2len(post_size) # this is ncessary when create "post2pre"␣
↪,
                                               # "post2syn"  etc. structures
        return self
```

Let's try to use it.

[20]:
```
conn = IndexConn(i=[0, 1, 2], j=[0, 0, 0])
conn = conn(pre_size=5, post_size=3)
```

[21]:
```
conn.requires('conn_mat')
```

[21]:
```
array([[1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

[22]:
```
conn.requires('pre2post')
```

[22]:
```
[array([0]),
 array([0]),
 array([0]),
 array([], dtype=int32),
 array([], dtype=int32)]
```

[23]:
```
conn.requires('pre2syn')
```

[23]:
```
[array([0]),
 array([1]),
 array([2]),
 array([], dtype=int32),
 array([], dtype=int32)]
```

[24]:
```
conn.requires('pre_slice_syn')
```

[24]:
```
array([[0, 1],
       [1, 2],
       [2, 3],
       [3, 3],
       [3, 3]])
```

Author:

- Chaoming Wang
- Email: adaduo@outlook.com
- Date: 2021.04.16

CHAPTER

SIX

EFFICIENT SYNAPTIC COMPUTATION

In a real project, the most of simulation time spends on the computation of the synapses. Therefore, figuring out what is the most efficient way to do synaptic computation is a necessary step to accelerate your computational project. Here, let's take an E/I balance network as an example to illustrate how to code an efficient synaptic computation.

[1]:
```
import brainpy as bp

import numpy as np
```

[2]:
```
import warnings
warnings.filterwarnings("ignore")
```

The E/I balance network COBA is adopted from (Vogels & Abbott, 2005) [1].

[3]:
```
# Parameters for network structure
num = 4000
num_exc = int(num * 0.75)
num_inh = int(num * 0.25)
```

Neuron Model

In COBA network, each integrate-and-fire neuron is characterized by a time constant, τ = 20 ms, and a resting membrane potential, V_{rest} = -60 mV. Whenever the membrane potential crosses a spiking threshold of -50 mV, an action potential is generated and the membrane potential is reset to the resting potential, where it remains clamped for a 5 ms refractory period. The membrane voltages are calculated as follows:

$$\tau \frac{dV}{dt} = (V_{rest} - V) + g_{exc}(E_{exc} - V) + g_{inh}(E_{inh} - V)$$

where reversal potentials are $E_{exc} = 0$ mV and $E_{inh} = -80$ mV.

[4]:
```
# Parameters for the neuron
tau = 20  # ms
Vt = -50  # mV
Vr = -60  # mV
El = -60  # mV
ref_time = 5.0  # refractory time, ms
I = 20.
```

[5]:
```
class LIF(bp.NeuGroup):
    target_backend = ['numpy', 'numba', 'numba-cuda']

    @staticmethod
```

(continues on next page)

(continued from previous page)

```
    def dev_V(V, t, Iexc):
        dV = (Iexc + El - V) / tau
        return dV

    def __init__(self, size, **kwargs):
        # variables
        self.V = bp.ops.zeros(size)
        self.spike = bp.ops.zeros(size)
        self.input = bp.ops.zeros(size)
        self.t_last_spike = bp.ops.ones(size) * -1e7

        # initialize
        self.int_V = bp.odeint(self.dev_V)
        super(LIF, self).__init__(size=size, **kwargs)

    def update(self, _t):
        for i in range(self.num):
            self.spike[i] = 0.
            if (_t - self.t_last_spike[i]) > ref_time:
                V = self.int_V(self.V[i], _t, self.input[i])
                if V >= Vt:
                    self.V[i] = Vr
                    self.spike[i] = 1.
                    self.t_last_spike[i] = _t
                else:
                    self.V[i] = V
            self.input[i] = I
```

Synapse Model

In COBA network, when a neuron fires, the appropriate synaptic variable of its postsynaptic targets are increased, $g_{exc} \leftarrow g_{exc} + \Delta g_{exc}$ for an excitatory presynaptic neuron and $g_{inh} \leftarrow g_{inh} + \Delta g_{inh}$ for an inhibitory presynaptic neuron. Otherwise, these parameters obey the following equations:

$$\tau_{exc} \frac{dg_{exc}}{dt} = -g_{exc} \qquad (1)$$

$$\tau_{inh} \frac{dg_{inh}}{dt} = -g_{inh} \qquad (2)$$

with synaptic time constants $\tau_{exc} = 5$ ms, $\tau_{inh} = 10$ ms, $\Delta g_{exc} = 0.6$ and $\Delta g_{inh} = 6.7$.

[6]:
```
# Parameters for the synapse
tau_exc = 5  # ms
tau_inh = 10  # ms
E_exc = 0.  # mV
E_inh = -80.  # mV
delta_exc = 0.6  # excitatory synaptic weight
delta_inh = 6.7  # inhibitory synaptic weight
```

[7]:
```
def run_net(neu_model, syn_model, backend='numba'):
    bp.backend.set(backend)

    E = neu_model(num_exc, monitors=['spike'])
```

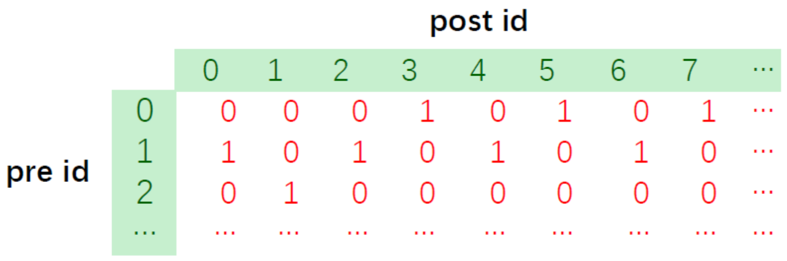
(continues on next page)

(continued from previous page)

```
    E.V = np.random.randn(num_exc) * 5. + Vr
    I = neu_model(num_inh, monitors=['spike'])
    I.V = np.random.randn(num_inh) * 5. + Vr
    E2E = syn_model(pre=E, post=E, conn=bp.connect.FixedProb(0.02),
                    tau=tau_exc, weight=delta_exc, E=E_exc)
    E2I = syn_model(pre=E, post=I, conn=bp.connect.FixedProb(0.02),
                    tau=tau_exc, weight=delta_exc, E=E_exc)
    I2E = syn_model(pre=I, post=E, conn=bp.connect.FixedProb(0.02),
                    tau=tau_inh, weight=delta_inh, E=E_inh)
    I2I = syn_model(pre=I, post=I, conn=bp.connect.FixedProb(0.02),
                    tau=tau_inh, weight=delta_inh, E=E_inh)

    net = bp.Network(E, I, E2E, E2I, I2E, I2I)
    t = net.run(100., report=True)

    fig, gs = bp.visualize.get_figure(row_num=5, col_num=1, row_len=1, col_len=10)
    fig.add_subplot(gs[:4, 0])
    bp.visualize.raster_plot(E.mon.ts, E.mon.spike, ylabel='E Group', xlabel='')
    fig.add_subplot(gs[4, 0])
    bp.visualize.raster_plot(I.mon.ts, I.mon.spike, ylabel='I Group', show=True)

    return t
```

6.1 Matrix-based connection

The matrix-based synaptic connection is one of the most intuitive way to build synaptic computations. The connection matrix between two neuron groups can be easily obtained through the function of `connector.requires('conn_mat')` (details please see *Synaptic Connectivity*). Each connection matrix is an array with the shape of `(num_pre, num_post)`, like



		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	…
	…	…	…	…	…	…	…	…	…	…

Based on `conn_mat`, the updating logic of the above synapses can be coded as:

[8]:
```
class SynMat1(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

    @staticmethod
    def dev_g(g, t, tau):
        dg = - g / tau
        return dg

    def __init__(self, pre, post, conn, tau, weight, E, **kwargs):
        # parameters
```

(continues on next page)

(continued from previous page)

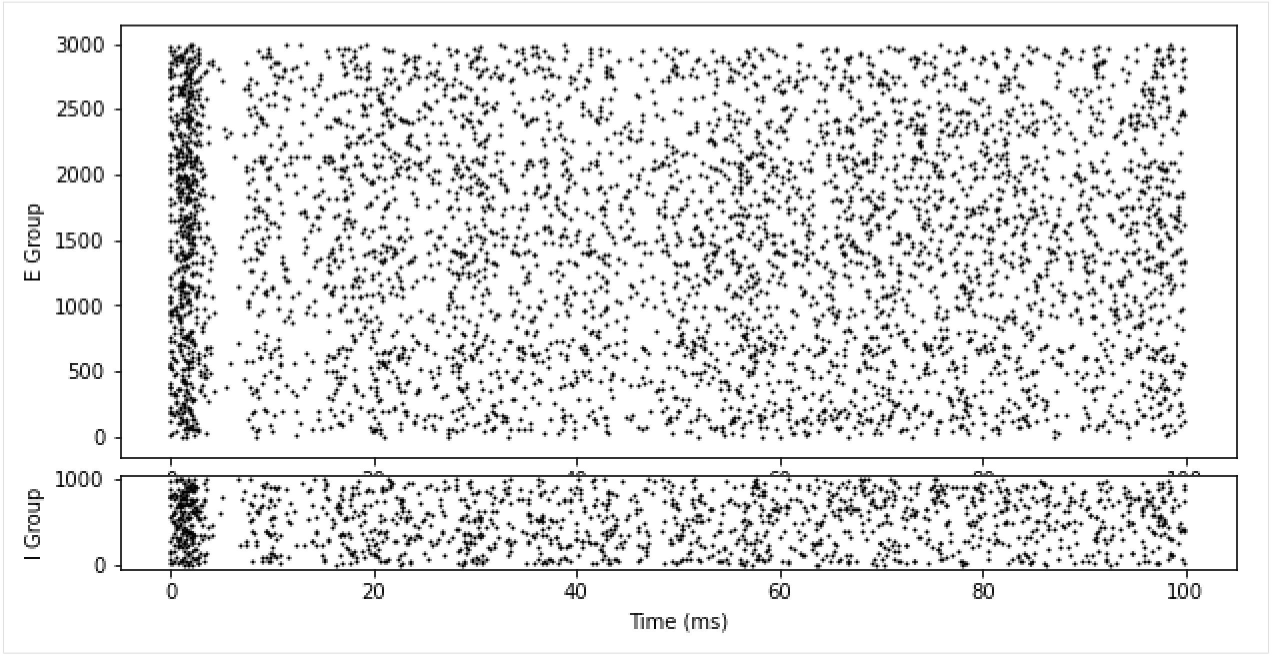
```
        self.tau = tau
        self.weight = weight
        self.E = E

        # p1: connections
        self.conn = conn(pre.size, post.size)
        self.conn_mat = self.conn.requires('conn_mat')

        # variables
        self.g = bp.ops.zeros(self.conn_mat.shape)

        # initialize
        self.int_g = bp.odeint(self.dev_g)
        super(SynMat1, self).__init__(pre=pre, post=post, **kwargs)

    def update(self, _t):
        self.g = self.int_g(self.g, _t, self.tau)
        # p2
        spike_on_syn = np.expand_dims(self.pre.spike, 1) * self.conn_mat
        # p3
        self.g += spike_on_syn * self.weight
        # p4
        self.post.input += np.sum(self.g, axis=0) * (self.E - self.post.V)
```

In the above defined `SynMat1` class, at "p1" line we requires a "conn_mat" structure for the later synaptic computation; at "p2" we get spikes for each synaptic connections according to "conn_mat" and "presynaptic spikes"; then at "p3", the spike-triggered synaptic variables are added onto its postsynaptic targets; at final "p4" code line, all connected synaptic values are summed to get the current effective conductance by *np.sum(self.g, axis=0)*.

Now, let's inspect the performance of this matrix-based synapse.

[9]:
```
t_syn_mat1 = run_net(neu_model=LIF, syn_model=SynMat1)
```

```
Compilation used 5.9524 s.
Start running ...
Run 10.0% used 18.150 s.
Run 20.0% used 36.369 s.
Run 30.0% used 55.145 s.
Run 40.0% used 73.973 s.
Run 50.0% used 92.088 s.
Run 60.0% used 110.387 s.
Run 70.0% used 128.580 s.
Run 80.0% used 146.776 s.
Run 90.0% used 165.066 s.
Run 100.0% used 183.273 s.
Simulation is done in 183.273 s.
```



This matrix-based synapse structure is very inefficient, because 99.9% time were wasted on the synaptic computation. We can inspect this by only running the neuron group models.

[10]:
```
group = LIF(num, monitors=['spike'])
group.V = np.random.randn(num) * 5. + Vr
group.run(100., inputs=('input', 5.), report=True)
```

```
Compilation used 0.2648 s.
Start running ...
Run 10.0% used 0.000 s.
Run 20.0% used 0.000 s.
Run 30.0% used 0.010 s.
Run 40.0% used 0.010 s.
Run 50.0% used 0.010 s.
Run 60.0% used 0.010 s.
Run 70.0% used 0.020 s.
Run 80.0% used 0.020 s.
Run 90.0% used 0.020 s.
Run 100.0% used 0.030 s.
Simulation is done in 0.030 s.
```

[10]:
```
0.030497789382934 57
```

As you can see, the neuron group only spends 0.030 s to run. After normalized by the total running time 183.273 s, the neuron group running only accounts for about 0.016369 percent.

6.1.1 Event-based updating

The inefficiency in the above matrix-based computation comes from the horrendous waste of time on synaptic computation. First, it is uncommon for a neuron to generate a spike; Second, in a group of neuron, the generated spikes (`self.pre.spike`) are usually sparse. Therefore, at many time points, there are many zeros in `self.pre.spike`, which results `self.g` add many unnecessary zeros (`self.g += spike_on_syn * self.weight`).

Alternatively, we can update `self.g` only when the pre-synaptic neuron produces a spike event (this is called as the **event-based updating** method):

[11]:
```
class SynMat2(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

    @staticmethod
    def dev_g(g, t, tau):
        dg = - g / tau
        return dg

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

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

        # variables
        self.g = bp.ops.zeros(self.conn_mat.shape)

        # initialize
        self.int_g = bp.odeint(self.dev_g)
        super(SynMat2, self).__init__(pre=pre, post=post, **kwargs)

    def update(self, _t):
        self.g = self.int_g(self.g, _t, self.tau)
        # p1
        for pre_i, spike in enumerate(self.pre.spike):
            if spike:
                self.g[pre_i] += self.conn_mat[pre_i] * self.weight
        self.post.input += np.sum(self.g, axis=0) * (self.E - self.post.V)
```

Compared to `SynMat1`, we replace "p2" and "p3" in `SynMat1` with "p1" in `SynMat2`. Now, the updating logic is only when the pre-synaptic neuron emits a spike (`if spike`), the connected post-synaptic state `g` will be updated (`self.g[pre_i] += self.conn_mat[pre_i] * self.weight`).

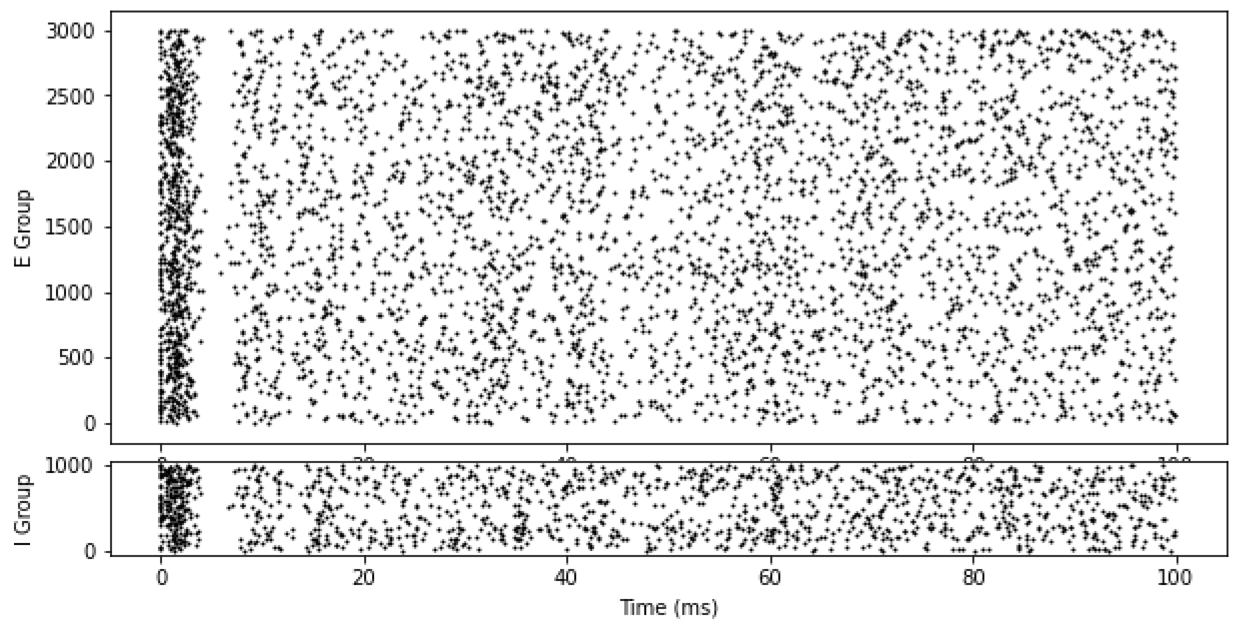
[12]:
```
t_syn_mat2 = run_net(neu_model=LIF, syn_model=SynMat2)
```

```
Compilation used 5.4851 s.
Start running ...
Run 10.0% used 9.456 s.
Run 20.0% used 18.824 s.
Run 30.0% used 28.511 s.
Run 40.0% used 38.090 s.
```

(continues on next page)

(continued from previous page)

```
Run 50.0% used 47.822 s.
Run 60.0% used 57.400 s.
Run 70.0% used 66.850 s.
Run 80.0% used 76.544 s.
Run 90.0% used 86.045 s.
Run 100.0% used 95.620 s.
Simulation is done in 95.620 s.
```



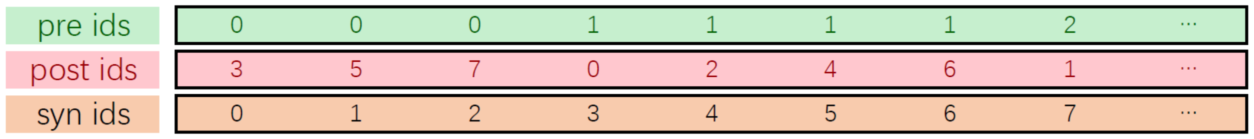
Such event-based matrix connection boosts the running speed nearly 2 times (compare 183.273 s with 95.620 s), but it's not good enough.

6.2 Vector-based connection

Matrix-based synaptic computation may be straightforward, but can cause severe wasted RAM memory and inefficient computation. Imaging you want to connect 10,000 pre-synaptic neurons to 10,000 post-synaptic neurons with a 10% random connection probability. Using *matrix*, you need 10^8 floats to save the synaptic state, and at each update step, you need do computation on 10^8 floats. Actually, the number of values you really needed is only 10^7. See, there is a huge memory waste and computing resource inefficiency.

6.2.1 `pre_ids` and `post_ids`

An effective method to solve this problem is to use *vector* to store the connectivity between neuron groups and the corresponding synaptic states. For the above defined connectivity `conn_mat`, we can align the connected pre-synaptic neurons and the post-synaptic neurons by two one-dimensional arrays: *pre_ids* and *post_ids*,



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	···

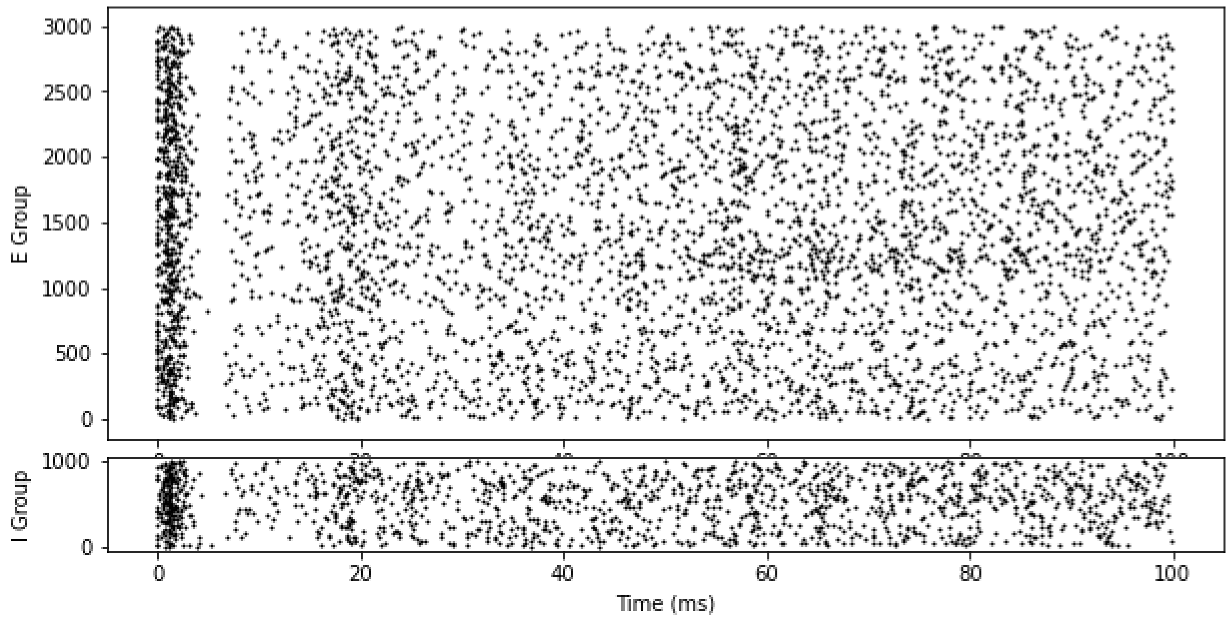
In such a way, we only need two vectors (`pre_ids` and `post_ids`, each has 10^7 floats) to store the synaptic connectivity. And, at each time step, we just need update a synaptic state vector with 10^7 floats.

[13]:
```
class SynVec1(bp.TwoEndConn):
    target_backend = ['numpy', 'numba', 'numba-cuda']

    @staticmethod
    def dev_g(g, t, tau):
        dg = - g / tau
        return dg

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

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

        # variables
        self.g = bp.ops.zeros(self.num)

        # initialize
        self.int_g = bp.odeint(self.dev_g)
        super(SynVec1, self).__init__(pre=pre, post=post, **kwargs)

    def update(self, _t):
        self.g = self.int_g(self.g, _t, self.tau)
        # p1: update
        for syn_i in range(self.num):
            pre_i = self.pre_ids[syn_i]
            if self.pre.spike[pre_i]:
                self.g[syn_i] += self.weight
        # p2: output
        for syn_i in range(self.num):
            post_i = self.post_ids[syn_i]
            self.post.input[post_i] += self.g[syn_i] * (self.E - self.post.V[post_i])
```

In SynVec1 class, we first update the synaptic state with "p1" code block, in which the synaptic state `self.g[syn_i]` is updated when the pre-synaptic neuron generates a spike (`if self.pre.spike[pre_i]`); then, at "p2" code block, we output the synaptic states onto the post-synaptic neurons.

```
[14]: t_syn_vec1 = run_net(neu_model=LIF, syn_model=SynVec1)
```

```
Compilation used 2.4805 s.
Start running ...
Run 10.0% used 0.190 s.
Run 20.0% used 0.391 s.
Run 30.0% used 0.611 s.
Run 40.0% used 0.802 s.
Run 50.0% used 0.995 s.
Run 60.0% used 1.185 s.
Run 70.0% used 1.391 s.
Run 80.0% used 1.621 s.
Run 90.0% used 1.819 s.
Run 100.0% used 2.009 s.
Simulation is done in 2.009 s.
```

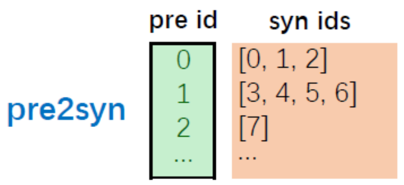


Great! Transform the matrix-based connection into the vector-based connection makes us get a huge speed boost (2.009 s vs 95.620 s). However, there also exists redundant part in `SynVec1` class. This is because a pre-synaptic neuron may connect to many post-synaptic neurons and thus at each step updating we will judge a pre-synaptic neuron whether generates a spike many times (`self.pre.spike[pre_i]`).

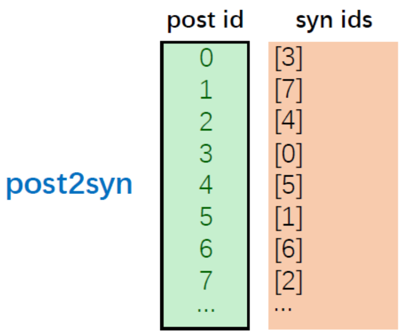
6.2.2 pre2syn and post2syn

In order to solve the above problem, here we create another two synaptic structures `pre2syn` and `post2syn` to help us retrieve the synapse states which connected with the pre-synaptic neuron i and the post-synaptic neuron j.

In a `pre2syn` list, each `pre2syn[i]` stores the synaptic state indexes projected from the pre-synaptic neuron i.



Similarly, we can create a `post2syn` list to indicate the connections between synapses and post-synaptic neurons. For each post-synaptic neuron j, `post2syn[j]` stores the indexes of synaptic elements which connected to the post neuron j.



Based on these connectivity mappings, we can define another version of synapse model by using `pre2syn` and `post2syn`:

[15]:
```
class SynVec2(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

    @staticmethod
    def dev_g(g, t, tau):
        dg = - g / tau
        return dg

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

        # connections
        self.conn = conn(pre.size, post.size)
        self.pre_ids, self.pre2syn, self.post2syn = self.conn.requires('pre_ids',
↪'pre2syn', 'post2syn')
        self.num = len(self.pre_ids)

        # variables
        self.g = bp.ops.zeros(self.num)
```

(continues on next page)

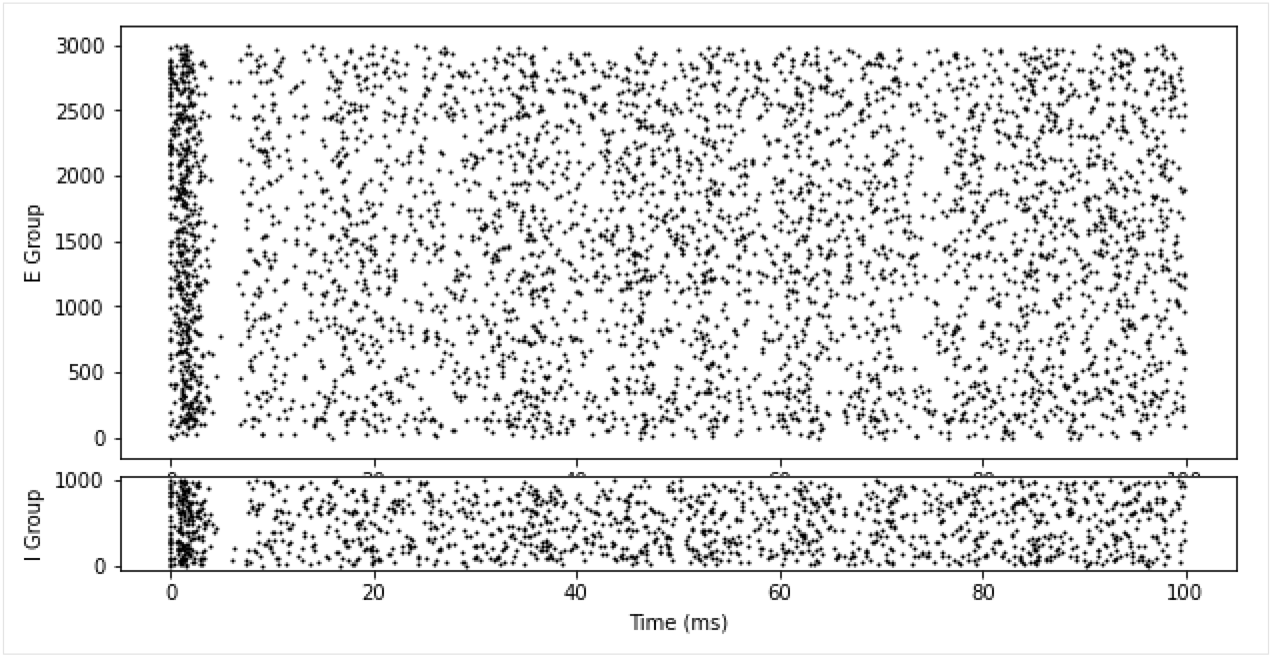
(continued from previous page)

```
        # initialize
        self.int_g = bp.odeint(self.dev_g)
        super(SynVec2, self).__init__(pre=pre, post=post, **kwargs)

    def update(self, _t):
        self.g = self.int_g(self.g, _t, self.tau)
        # p1: update
        for pre_i in range(self.pre.num):
            if self.pre.spike[pre_i]:
                for syn_i in self.pre2syn[pre_i]:
                    self.g[syn_i] += self.weight
        # p2: output
        for post_i in range(self.post.num):
            for syn_i in self.post2syn[post_i]:
                self.post.input[post_i] += self.g[syn_i] * (self.E - self.post.V[post_i])
```

In this `SynVec2` class, at “p1” code-block, we update synaptic states by the for-loop with the size of pre-synaptic number. If the pre-synaptic neuron elicits a spike `self.pre.spike[pre_i]`, we will for-loop its connected synaptic states by `for syn_i in self.pre2syn[pre_i]`. In such a way, we only need to judge the pre-synaptic neuron `pre_i` spike state once. Similarly, at “p2” code-block, the synaptic output is also implemented with the post-synaptic neuron for-loop.

[16]:
```
t_syn_vec2 = run_net(neu_model=LIF, syn_model=SynVec2)
```

```
Compilation used 2.7670 s.
Start running ...
Run 10.0% used 0.180 s.
Run 20.0% used 0.383 s.
Run 30.0% used 0.573 s.
Run 40.0% used 0.764 s.
Run 50.0% used 0.994 s.
Run 60.0% used 1.186 s.
Run 70.0% used 1.378 s.
Run 80.0% used 1.568 s.
Run 90.0% used 1.765 s.
Run 100.0% used 1.995 s.
Simulation is done in 1.995 s.
```



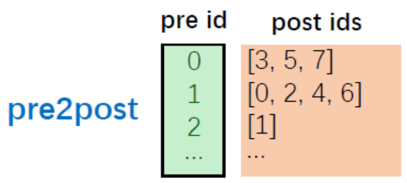
We only got a small increase in speed performance (1.995 s vs 2.009 s). This is because the optimization of the "update" block has run its course. Currently, the most of the running costs spend on the "output" block.

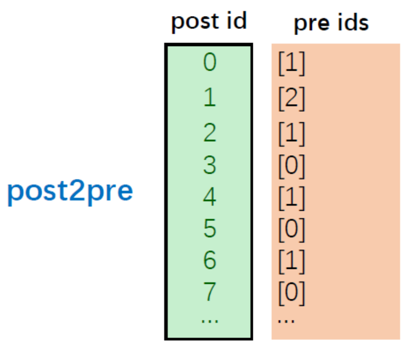
6.2.3 `pre2post` and `post2pre`

Notice that for this kind of synapse model, the synaptic states g onto a post-synaptic neuron can be modeled together. This is because the synaptic state evolution according to the differential equation (1) and (2) after the pre-synaptic spikes can be superposed. This means that we can declare a synaptic state `self.g` with the shape of `post.num`, not the shape of the synapse number.

In order to achieve this goal, we create another two synaptic structures (`pre2post` and `post2pre`) which establish the direct mapping between the pre-synaptic neurons and the post-synaptic neurons can be established. `pre2post` contains the connected post-synaptic neurons indexes, in which `pre2post[i]` retrieves the post neuron ids projected from pre-synaptic neuron i. `post2pre` contains the pre-synaptic neurons indexes, in which `post2pre[j]` retrieves the pre-syanptic neuron ids which project to post-synaptic neuron j.

Also,





[17]:
```
class SynVec3(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

    @staticmethod
    def dev_g(g, t, tau):
        dg = - g / tau
        return dg

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

        # connections
        self.conn = conn(pre.size, post.size)
        self.pre2post = self.conn.requires('pre2post')

        # variables
        self.g = bp.ops.zeros(post.num)

        # initialize
        self.int_g = bp.odeint(self.dev_g)
        super(SynVec3, self).__init__(pre=pre, post=post, **kwargs)

    def update(self, _t):
        self.g = self.int_g(self.g, _t, self.tau)
        # p1: update
        for pre_i in range(self.pre.num):
            if self.pre.spike[pre_i]:
                for post_i in self.pre2post[pre_i]:
                    self.g[post_i] += self.weight
        # p2: output
        self.post.input += self.g * (self.E - self.post.V)
```

In SynVec3 class, we require a `pre2post` structure, and then at "p1" code-block, when the pre-synaptic neuron `pre_i` emits a spike, the connected post-synaptic neurons' state `self.g[post_i]` will increase the conductance.

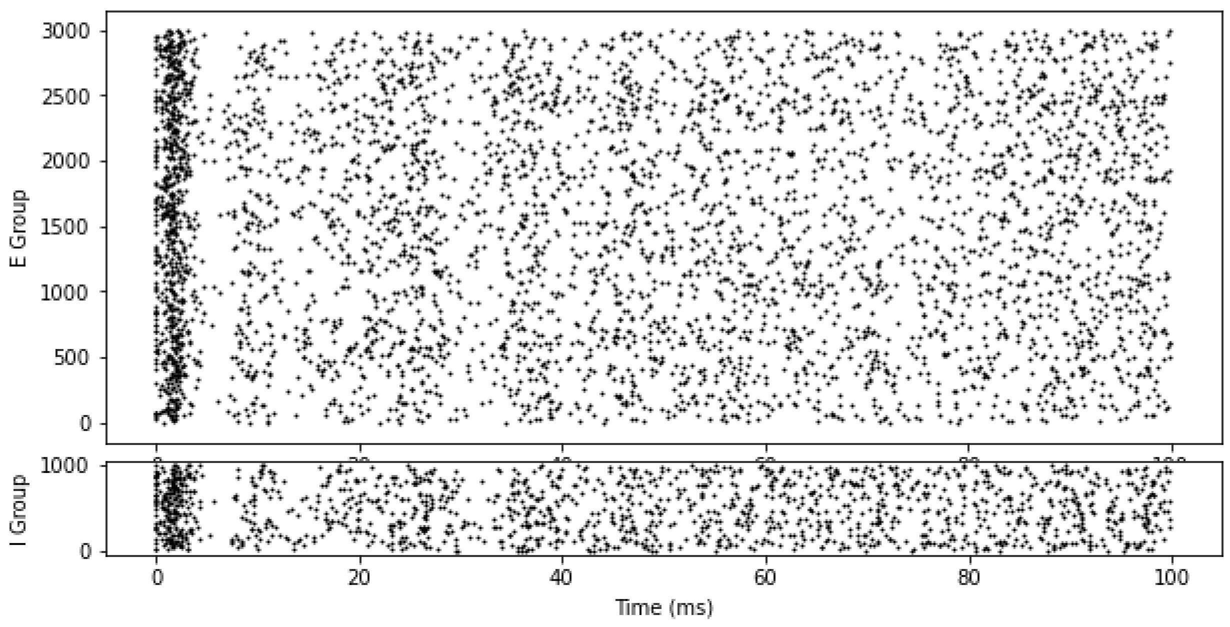
[18]:
```
t_syn_vec3 = run_net(neu_model=LIF, syn_model=SynVec3)
```

```
Compilation used 2.7279 s.
Start running ...
```

(continues on next page)

(continued from previous page)

```
Run 10.0% used 0.000 s.
Run 20.0% used 0.010 s.
Run 30.0% used 0.020 s.
Run 40.0% used 0.020 s.
Run 50.0% used 0.030 s.
Run 60.0% used 0.040 s.
Run 70.0% used 0.040 s.
Run 80.0% used 0.050 s.
Run 90.0% used 0.060 s.
Run 100.0% used 0.060 s.
Simulation is done in 0.060 s.
```

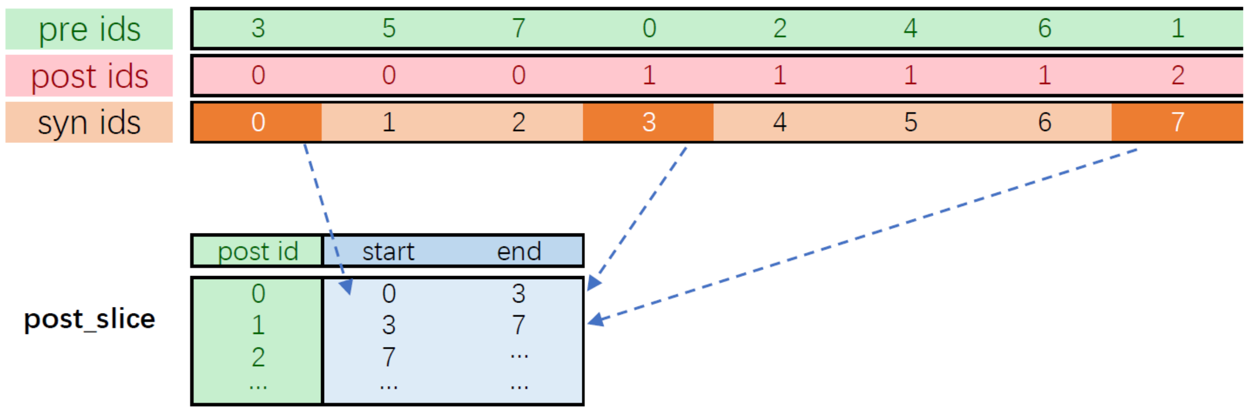


Yeah, the running speed gets a huge boosting (0.060 s vs 1.995 s), which demonstrates the super effectiveness of this kind of synaptic computation.

6.2.4 `pre_slice` and `post_slice`

However, it is not perfect. This is because `pre2syn`, `post2syn`, `pre2post` and `post2pre` are all the data with the `list` type, which can not be directly deployed to GPU devices. What the GPU device prefers are only arrays.

To solve this problem, we, instead, can create a `post_slice` connection structure which stores the *start* and the *end* position on the synpase state for each connected post-synaptic neuron j. `post_slice` can be implemented by aligning the pre ids according to the sequential post id $0, 1, 2, ...$ (look the following illustrating figure). For each post neuron j, `start, end = post_slice[j]` retrieves the start/end position of the connected synapse states.



Therefore, an alternative updating logic of `pre2syn` and `post2syn` (in `SynVec2` class) can be replaced by `post_slice` and `pre_ids`:

[19]:
```
class SynVec4(bp.TwoEndConn):
    target_backend = ['numpy', 'numba', 'numba-cuda']

    @staticmethod
    def dev_g(g, t, tau):
        dg = - g / tau
        return dg

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

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

        # variables
        self.g = bp.ops.zeros(self.num)

        # initialize
        self.int_g = bp.odeint(self.dev_g)
        super(SynVec4, self).__init__(pre=pre, post=post, **kwargs)

    def update(self, _t):
        self.g = self.int_g(self.g, _t, self.tau)
        # p1: update
        for syn_i in range(self.num):
            pre_i = self.pre_ids[syn_i]
            if self.pre.spike[pre_i]:
                self.g[syn_i] += self.weight
        # p2: output
        for post_i in range(self.post.num):
            start, end = self.post_slice[post_i]
            for syn_i in range(start, end):
```

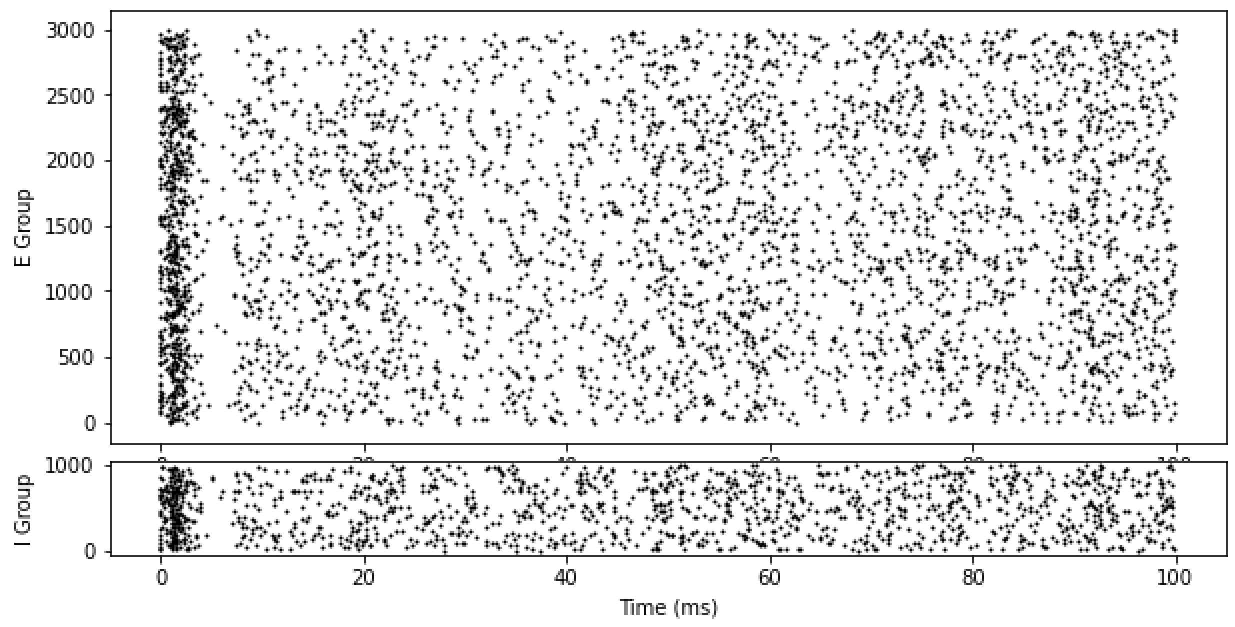
(continues on next page)

(continued from previous page)

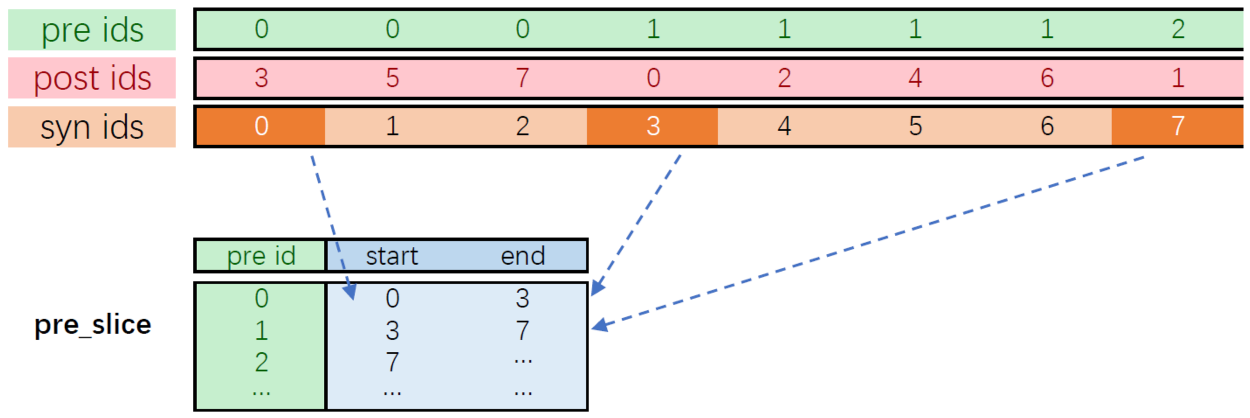
```
                self.post.input[post_i] += self.g[syn_i] * (self.E - self.post.V[post_i])
```

[20]:
```
t_syn_vec4 = run_net(neu_model=LIF, syn_model=SynVec4)
```

```
Compilation used 2.5473 s.
Start running ...
Run 10.0% used 0.190 s.
Run 20.0% used 0.372 s.
Run 30.0% used 0.552 s.
Run 40.0% used 0.744 s.
Run 50.0% used 0.934 s.
Run 60.0% used 1.158 s.
Run 70.0% used 1.351 s.
Run 80.0% used 1.538 s.
Run 90.0% used 1.718 s.
Run 100.0% used 1.915 s.
Simulation is done in 1.915 s.
```



Similarly, a connection mapping `pre_slice` can also be implemented, in which for each pre-synaptic neuron i, `start, end = pre_slice[i]` retrieves the start/end position of the connected synapse states.

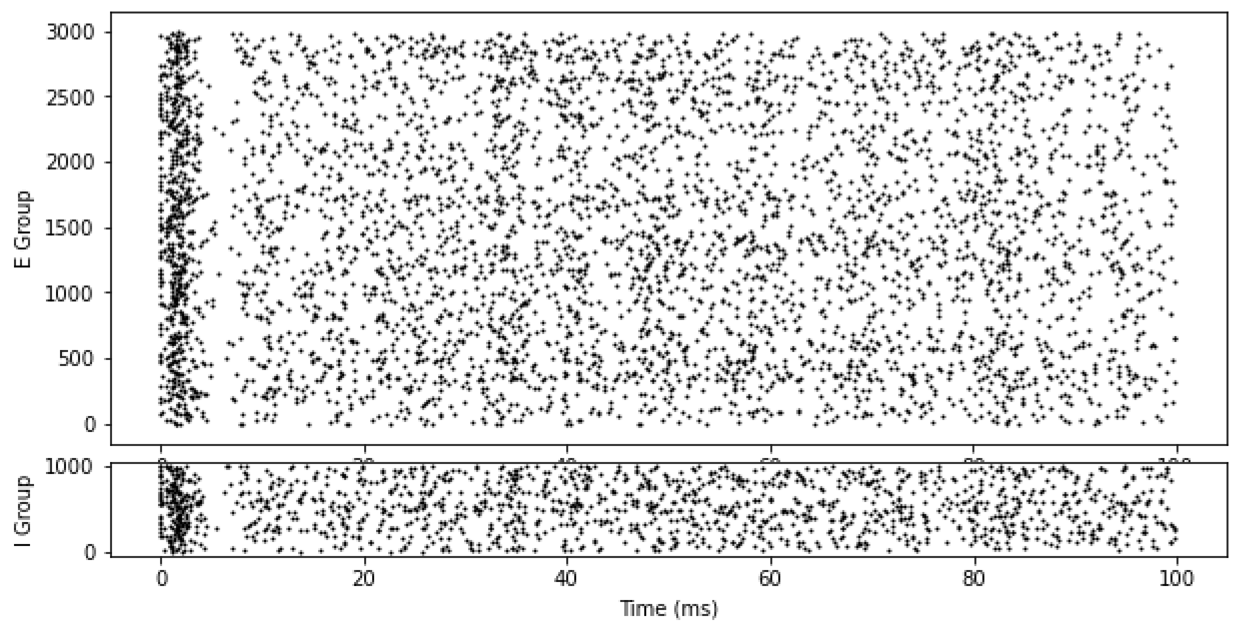


Moreover, an alternative updating logic of `pre2post` (in `SynVec3` class) can also be replaced by `pre_slice` and `post_ids`:

[21]:
```
class SynVec5(bp.TwoEndConn):
    target_backend = ['numpy', 'numba', 'numba-cuda']

    @staticmethod
    def dev_g(g, t, tau):
        dg = - g / tau
        return dg

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

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

        # variables
        self.g = bp.ops.zeros(post.num)

        # initialize
        self.int_g = bp.odeint(self.dev_g)
        super(SynVec5, self).__init__(pre=pre, post=post, **kwargs)

    def update(self, _t):
        self.g = self.int_g(self.g, _t, self.tau)
        # p1: update
        for pre_i in range(self.pre.num):
            if self.pre.spike[pre_i]:
                start, end = self.pre_slice[pre_i]
                for post_i in self.post_ids[start: end]:
                    self.g[post_i] += self.weight
        # p2: output
        self.post.input += self.g * (self.E - self.post.V)
```

[22]:
```
t_syn_vec5 = run_net(neu_model=LIF, syn_model=SynVec5)
```

```
Compilation used 2.9655 s.
Start running ...
Run 10.0% used 0.000 s.
Run 20.0% used 0.010 s.
Run 30.0% used 0.020 s.
Run 40.0% used 0.020 s.
Run 50.0% used 0.030 s.
Run 60.0% used 0.040 s.
Run 70.0% used 0.040 s.
Run 80.0% used 0.050 s.
Run 90.0% used 0.060 s.
Run 100.0% used 0.060 s.
Simulation is done in 0.060 s.
```



6.3 Speed comparison

In this tutorial, we introduce nine different synaptic connection structures:

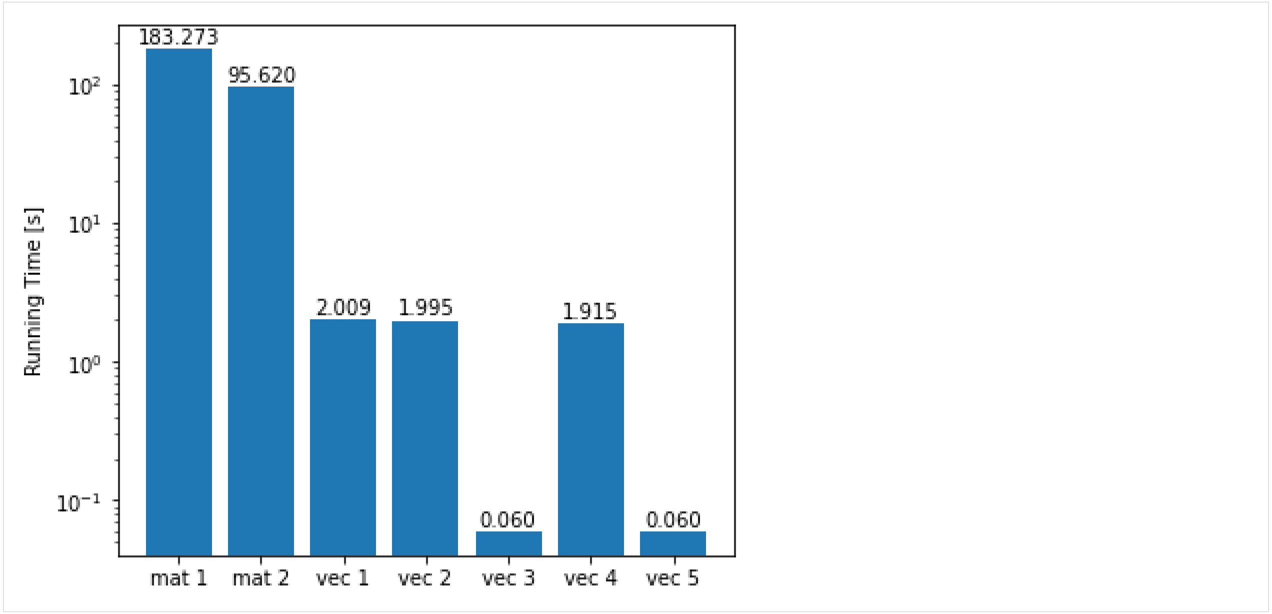
1. **conn_mat** : The connection matrix with the shape of `(pre_num, post_num)`.
2. **pre_ids**: The connected pre-synaptic neuron indexes, a vector with the shape pf `syn_num`.
3. **post_ids**: The connected post-synaptic neuron indexes, a vector with the shape pf `syn_num`.
4. **pre2syn**: A list (with the length of `pre_num`) contains the synaptic indexes connected by each pre-synaptic neuron. `pre2syn[i]` denotes the synapse ids connected by the pre-synaptic neuron i.
5. **post2syn**: A list (with the length of `post_num`) contains the synaptic indexes connected by each post-synaptic neuron. `post2syn[j]` denotes the synapse ids connected by the post-synaptic neuron j.

6. **pre2post**: A list (with the length of `pre_num`) contains the post-synaptic indexes connected by each pre-synaptic neuron. `pre2post[i]` retrieves the post neurons connected by the pre neuron i.
7. **post2pre**: A list (with the length of `post_num`) contains the pre-synaptic indexes connected by each post-synaptic neuron. `post2pre[j]` retrieves the pre neurons connected by the post neuron j.
8. **pre_slice**: A two dimensional matrix with the shape of `(pre_num, 2)` stores the *start* and *end* positions on the synapse state for each connected pre-synaptic neuron i .
9. **post_slice**: A two dimensional matrix with the shape of `(post_num, 2)` stores the *start* and *end* positions on the synapse state for each connected post-synaptic neuron j .

We illustrate their efficiency by a spare randomly connected E/I balance network COBA [1]. We summarize their speed in the following comparison figure:

[23]:
```
names = ['mat 1',    'mat 2',    'vec 1',    'vec 2',    'vec 3',    'vec 4',    'vec 5']
times = [t_syn_mat1, t_syn_mat2, t_syn_vec1, t_syn_vec2, t_syn_vec3, t_syn_vec4, t_syn_
↪vec5]
xs = list(range(len(times)))
```

[27]:
```
import matplotlib.pyplot as plt

def autolabel(rects):
    """Attach a text label above each bar in *rects*, displaying its height."""
    for rect in rects:
        height = rect.get_height()
        ax.annotate(f'{height:.3f}',
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 0.5),  # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom')

fig, gs = bp.visualize.get_figure(1, 1, 4, 5)

ax = fig.add_subplot(gs[0, 0])
rects = ax.bar(xs, times)
ax.set_xticks(xs)
ax.set_xticklabels(names)
ax.set_yscale('log')
plt.ylabel('Running Time [s]')
autolabel(rects)
```



However, the speed comparison presented here does not mean that the vector-based connection is always better than the matrix-based connection. Vector-based synaptic model is well suitable to run on the JIT compilers like Numba. Whereas the matrix-based synaptic model is best to run on the array- or tensor-oriented backend such like NumPy, PyTorch, TensorFlow, and is highly suitable to solve problems for dense connections, such like all-to-all connection.

References:

[1] Vogels, T. P. and Abbott, L. F. (2005), Signal propagation and logic gating in networks of integrate-and-fire neurons., J. Neurosci., 25, 46, 10786–95

Author:

- Chaoming Wang
- Email: adaduo@outlook.com
- Date: 2021.04.18

CHAPTER

SEVEN

RUNNING ORDER SCHEDULING

In this section, we will detail the running order scheduling in `BrainPy`, and tell you how to customize the running order of objects in a network.

[1]:
```
import brainpy as bp

import warnings
warnings.filterwarnings("ignore")
```

7.1 Step function

In BrainPy, the basic concept for neurodynamics simulation is the `step function`. For example, in a customized `brainpy.NeuGroup`,

[2]:
```
class HH(bp.NeuGroup):
    target_backend = 'numpy'

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

        # parameters
        self.ENa = ENa
        self.EK = EK
        self.EL = EL
        self.C = C
        self.gNa = gNa
        self.gK = gK
        self.gL = gL
        self.V_th = V_th

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

(continues on next page)

(continued from previous page)

```
    @bp.odeint(method='exponential_euler')
    def integral(self, V, m, h, n, t, Iext):
        alpha = 0.1 * (V + 40) / (1 - bp.ops.exp(-(V + 40) / 10))
        beta = 4.0 * bp.ops.exp(-(V + 65) / 18)
        dmdt = alpha * (1 - m) - beta * m

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

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

        I_Na = (self.gNa * m ** 3 * h) * (V - self.ENa)
        I_K = (self.gK * n ** 4) * (V - self.EK)
        I_leak = self.gL * (V - self.EL)
        dVdt = (- I_Na - I_K - I_leak + Iext) / self.C

        return dVdt, dmdt, dhdt, dndt

    def update(self, _t, _i, _dt):
        V, m, h, n = self.integral(self.V, self.m, self.h, self.n, _t, self.input)
        self.spike = (self.V < self.V_th) * (V >= self.V_th)
        self.V = V
        self.m = m
        self.h = h
        self.n = n
        self.input[:] = 0
```

three step functions are included:

- `update` step function (explicitly defined by users)
- `monitor` step function (implicitly generated by BrainPy if users require monitors when intializing `HH` model: `group = HH(..., monitors=['V', ...])`)
- `input` step function (implicitly generated by BrainPy if users give inputs: `group.run(..., inputs=('input', 10.))`)

We can inspect this by:

[3]:
```
group1 = HH(1, monitors=['V', 'input'])

hh_schedule = tuple(group1.schedule())
hh_schedule
```

[3]:
```
('input', 'update', 'monitor')
```

Later, BrainPy will update the step functions by the running order of `hh_schedule`:

[4]:
```
group1.build(duration=100, inputs=('input', 10.), show_code=True)
```

```
The input to a new key "input" in <__main__.HH object at 0x0000022CD3844580>.
def input_step(_i):
  NG1.input += NG1._input_data_of_input
{'NG1': <__main__.HH object at 0x0000022CD3844580>}

def monitor_step(_i, _t):
  NG1.mon.V[_i] = NG1.V
  NG1.mon.V_t[_i] = _t
  NG1.mon.input[_i] = NG1.input
  NG1.mon.input_t[_i] = _t
{'NG1': <__main__.HH object at 0x0000022CD3844580>,
 'ops': <module 'brainpy.backend.ops' from '../..\\brainpy\\backend\\ops\\__init__.py'>}

def run_func(_t, _i, _dt):
  NG1.input_step(_i)
  NG1.update(_t, _i, _dt)
  NG1.monitor_step(_i, _t)
{'NG1': <__main__.HH object at 0x0000022CD3844580>}
```

[4]:
```
<function run_func(_t, _i, _dt)>
```

As you can see, in the final `run_func(_t, _i, _dt)`, the running order of step functions is in line with the `hh_schedule`.

7.2 Customize `schedule()`

Fortunately, BrainPy allows users to customize the running schedule.

7.2.1 Customize `schedule()` in a brain object

Let's take the above HH class as the example to illustrate how to customize the running order in a single brain object:

[5]:
```
class HH2(bp.NeuGroup):
    target_backend = 'numpy'

    def __init__(self, size, ENa=50., EK=-77., EL=-54.387,
                 C=1.0, gNa=120., gK=36., gL=0.03, V_th=20.,
                 **kwargs):
        super(HH2, self).__init__(size=size, **kwargs)

        # parameters
        self.ENa = ENa
        self.EK = EK
        self.EL = EL
        self.C = C
        self.gNa = gNa
        self.gK = gK
        self.gL = gL
        self.V_th = V_th
```

(continues on next page)

(continued from previous page)

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

    @bp.odeint(method='exponential_euler')
    def integral(self, V, m, h, n, t, Iext):
        alpha = 0.1 * (V + 40) / (1 - bp.ops.exp(-(V + 40) / 10))
        beta = 4.0 * bp.ops.exp(-(V + 65) / 18)
        dmdt = alpha * (1 - m) - beta * m

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

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

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

        return dVdt, dmdt, dhdt, dndt

    def update(self, _t, _i, _dt):
        V, m, h, n = self.integral(self.V, self.m, self.h, self.n, _t, self.input)
        self.spike = (self.V < self.V_th) * (V >= self.V_th)
        self.V = V
        self.m = m
        self.h = h
        self.n = n
        self.input[:] = 0

    def schedule(self):
        return ('monitor',) + tuple(self.steps.keys()) + ('input',)
```

Here, we define HH2 class. What's different from the above HH model is that the customized `schedule()` function makes the `monitor` function and the user-defined `steps` functions run first, then run the `input` function.

[6]:
```
group2 = HH2(1, monitors=['V', 'input'])

group2.schedule()
```

[6]:
```
('monitor', 'update', 'input')
```

The advantage of such scheduling arrangement is that users can monitor the total synaptic inputs at each time step:

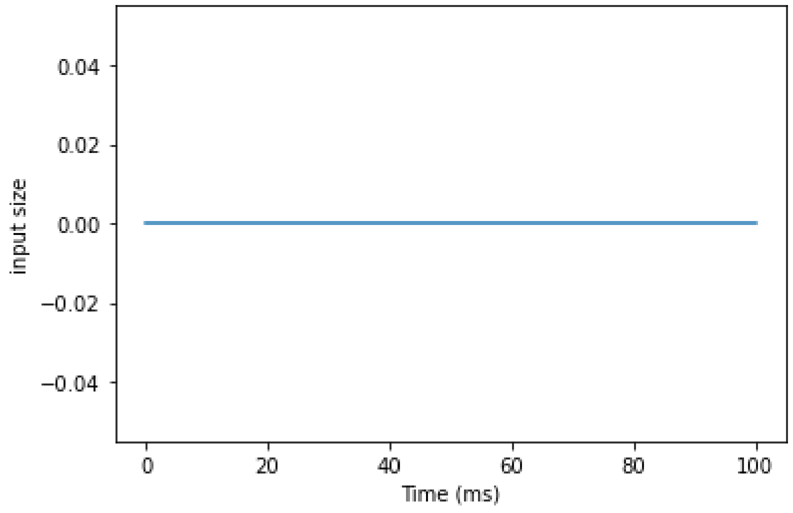
[7]:
```
import numpy as np
```

(continues on next page)

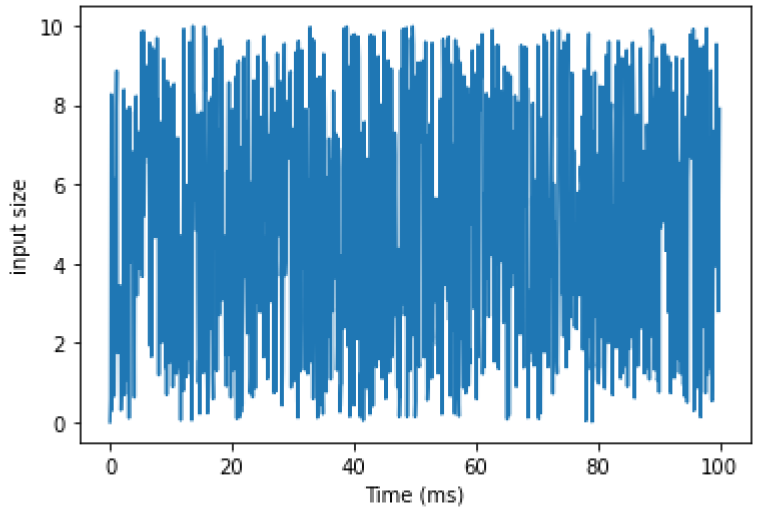
(continued from previous page)

```
duration = 100
random_inputs = np.random.random(int(duration / bp.backend.get_dt())) * 10.
```

[8]:
```
group1.run(duration, inputs=('input', random_inputs))

bp.visualize.line_plot(group1.mon.input_t, group1.mon.input,
                       ylabel='input size', show=True)
```



[9]:
```
group2.run(duration, inputs=('input', random_inputs))

bp.visualize.line_plot(group2.mon.input_t, group2.mon.input,
                       ylabel='input size', show=True)
```



This is because in the original HH class, the `self.input` variable is reset to 0 in the `update` function before calling the `monitor_step` function.

7.2.2 Customize `schedule()` in a network

Similarly, the running order scheduling for multiple brain objects in a network can also be customized. Let's illustrate this by a COBA E/I balance network.

[10]:
```
bp.backend.set('numba')

# Parameters for the neurons
num_exc, num_inh = 3200, 800
tau = 20  # ms
Vt = -50  # mV
Vr = -60  # mV
El = -60  # mV
ref_time = 5.0  # refractory time, ms
```

[11]:
```
class LIF(bp.NeuGroup):
    target_backend = ['numpy', 'numba']

    @staticmethod
    @bp.odeint(method='exponential_euler')
    def int_V(V, t, Iexc):
        dV = (Iexc + El - V) / tau
        return dV

    def __init__(self, size, **kwargs):
        super(LIF, self).__init__(
            size=size, steps=[self.update, self.threshold, self.reset], **kwargs)

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

    def update(self, _t):
        # update the membrane potential
        for i in range(self.num):
            if (_t - self.t_last_spike[i]) > ref_time:
                self.V[i] = self.int_V(self.V[i], _t, self.input[i])

    def threshold(self):
        # judge whether the neuron potential reach the spike threshold
        for i in range(self.num):
            self.spike[i] = self.V[i] >= Vt

    def reset(self, _t):
        # reset the neuron potential.
        for i in range(self.num):
            if self.spike[i]:
                self.V[i] = Vr
                self.t_last_spike[i] = _t
            self.input[i] = 20.
```

[12]:
```
# Parameters for the synapses
tau_exc = 5  # ms
tau_inh = 10  # ms
E_exc = 0.  # mV
E_inh = -80.  # mV
delta_exc = 0.6  # excitatory synaptic weight
delta_inh = 6.7  # inhibitory synaptic weight
```

[13]:
```
class Syn(bp.TwoEndConn):
    target_backend = ['numpy', 'numba']

    @staticmethod
    @bp.odeint(method='exponential_euler')
    def int_g(g, t, tau):
        dg = - g / tau
        return dg

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

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

        # variables
        self.g = bp.ops.zeros(post.num)

        super(Syn, self).__init__(pre=pre, post=post,
                                  steps=[self.update, self.output],
                                  **kwargs)

    def update(self, _t):
        self.g = self.int_g(self.g, _t, self.tau)
        # p1: update
        for pre_i in range(self.pre.num):
            if self.pre.spike[pre_i]:
                start, end = self.pre_slice[pre_i]
                for post_i in self.post_ids[start: end]:
                    self.g[post_i] += self.weight

    def output(self):
        # p2: output
        self.post.input += self.g * (self.E - self.post.V)
```

[14]:
```
def run_network(net_model):
    from pprint import pprint

    E = LIF(num_exc, monitors=['spike'], name='E')
    E.V = np.random.randn(num_exc) * 5. + Vr
```

(continues on next page)

(continued from previous page)

```
    I = LIF(num_inh, monitors=['spike'], name='I')
    I.V = np.random.randn(num_inh) * 5. + Vr

    E2E = Syn(pre=E, post=E, conn=bp.connect.FixedProb(0.02),
              tau=tau_exc, weight=delta_exc, E=E_exc, name='E2E')
    E2I = Syn(pre=E, post=I, conn=bp.connect.FixedProb(0.02),
              tau=tau_exc, weight=delta_exc, E=E_exc, name='E2I')
    I2E = Syn(pre=I, post=E, conn=bp.connect.FixedProb(0.02),
              tau=tau_inh, weight=delta_inh, E=E_inh, name='I2E')
    I2I = Syn(pre=I, post=I, conn=bp.connect.FixedProb(0.02),
              tau=tau_inh, weight=delta_inh, E=E_inh, name='I2I')

    net = net_model(E, I, E2E, E2I, I2E, I2I)

    print(f'The network schedule is: ')
    pprint(list(net.schedule()))

    t = net.run(100.)

    print('\n\nThe running result is:')
    fig, gs = bp.visualize.get_figure(row_num=5, col_num=1, row_len=1, col_len=10)
    fig.add_subplot(gs[:4, 0])
    bp.visualize.raster_plot(E.mon.spike_t, E.mon.spike, ylabel='E Group', xlabel='')
    fig.add_subplot(gs[4, 0])
    bp.visualize.raster_plot(I.mon.spike_t, I.mon.spike, ylabel='I Group', show=True)
```

The default scheduling in a network is the serial running of the brain objects. The code for schedule generation is like this:

[15]:
```
class DefaultNetwork(bp.Network):
    def schedule(self):
        for node in self.all_nodes.values():
            for key in node.schedule():
                yield f'{node.name}.{key}'
```

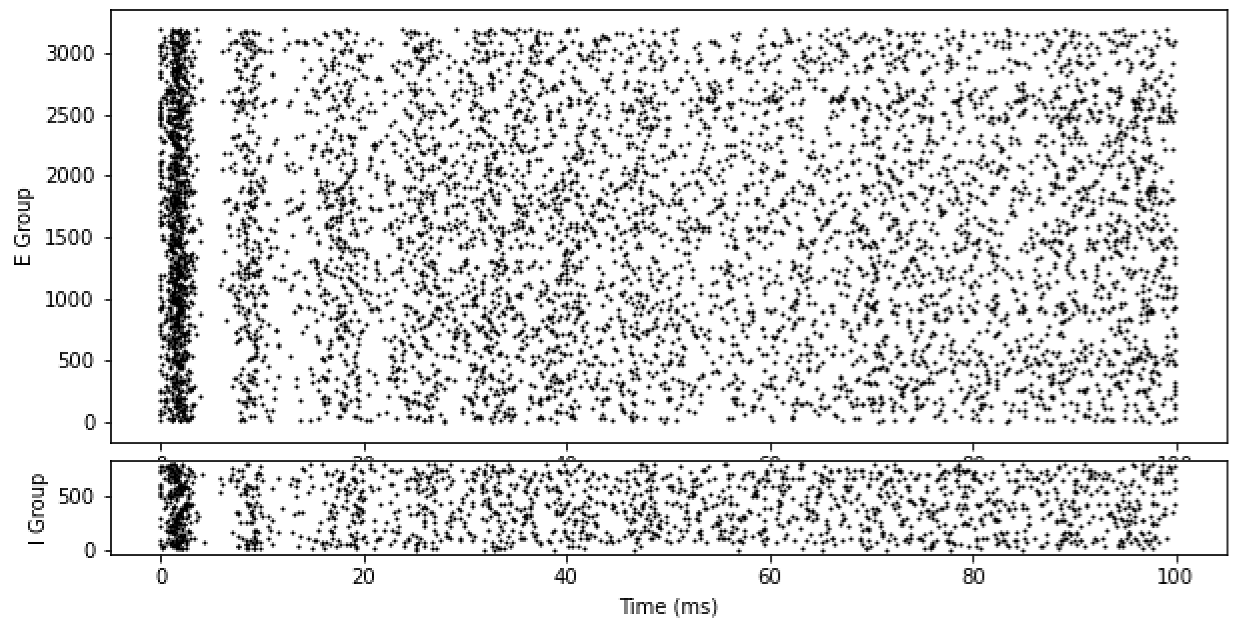
[16]:
```
run_network(DefaultNetwork)
```

```
The network schedule is:
['E.input',
 'E.update',
 'E.threshold',
 'E.reset',
 'E.monitor',
 'I.input',
 'I.update',
 'I.threshold',
 'I.reset',
 'I.monitor',
 'E2E.input',
 'E2E.update',
 'E2E.output',
```

(continues on next page)

(continued from previous page)

```
 'E2E.monitor',
 'E2I.input',
 'E2I.update',
 'E2I.output',
 'E2I.monitor',
 'I2E.input',
 'I2E.update',
 'I2E.output',
 'I2E.monitor',
 'I2I.input',
 'I2I.update',
 'I2I.output',
 'I2I.monitor']


The running result is:
```



In the next, we will define a network model in which all the `input` functions run first, then run the synaptic step functions (because the synapses will output the values to neuron groups), next we run the neuronal step functions, finally we monitor the neural and synaptic variables.

[17]:
```
class CustomizedNetwork(bp.Network):
    def schedule(self):

        # run input functions
        for node in self.all_nodes.values():  # self.all_nodes is a dict with the
                                              # format of {"node_name": node}
            yield f'{node.name}.input'

        # run synpatic step functions
        for node in self.all_nodes.values():
            if isinstance(node, bp.SynConn):
```

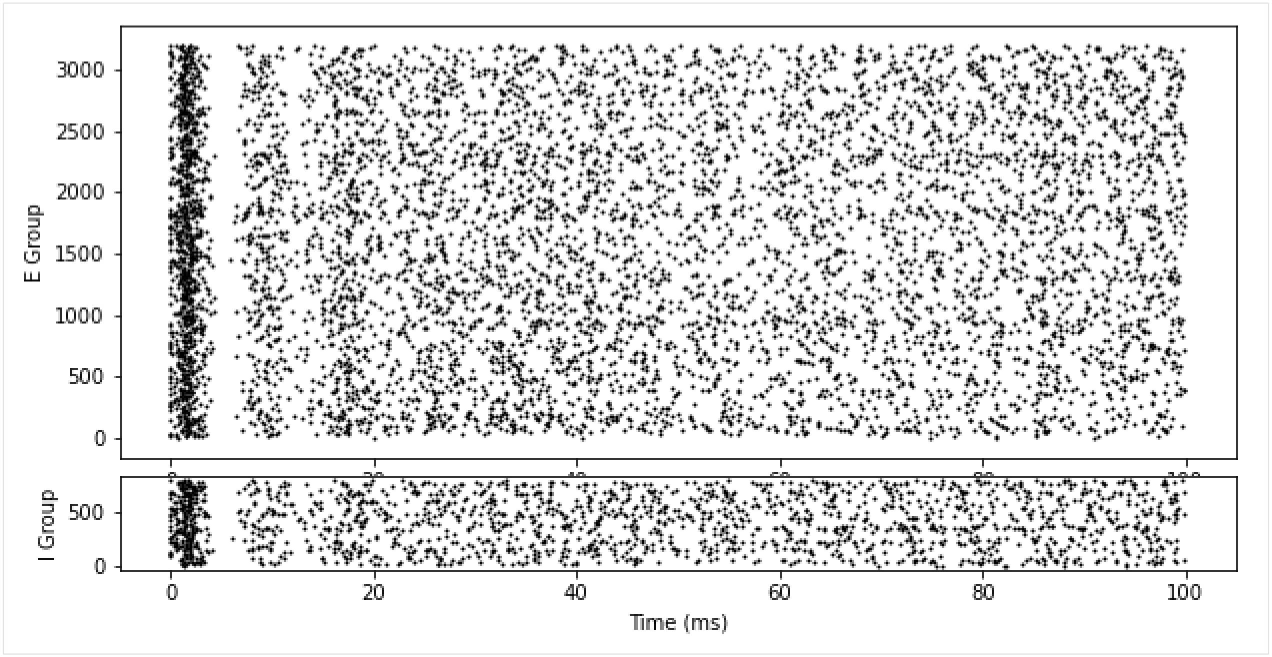
(continues on next page)

(continued from previous page)

```
                for key in node.steps.keys(): # node.steps is a dict with the
                                              # format of {"step_name": step}
                    yield f'{node.name}.{key}'

        # run neural model step functions
        for node in self.all_nodes.values():
            if isinstance(node, bp.NeuGroup):
                for key in node.steps.keys(): # node.steps is a dict with the
                                              # format of {"step_name": step}
                    yield f'{node.name}.{key}'

        # run the monitor functions
        for node in self.all_nodes.values():
            yield f'{node.name}.monitor'
```

[18]:
```
run_network(CustomizedNetwork)
```

```
The network schedule is:
['E.input',
 'I.input',
 'E2E.input',
 'E2I.input',
 'I2E.input',
 'I2I.input',
 'E2E.update',
 'E2E.output',
 'E2I.update',
 'E2I.output',
 'I2E.update',
 'I2E.output',
 'I2I.update',
 'I2I.output',
 'E.update',
 'E.threshold',
 'E.reset',
 'I.update',
 'I.threshold',
 'I.reset',
 'E.monitor',
 'I.monitor',
 'E2E.monitor',
 'E2I.monitor',
 'I2E.monitor',
 'I2I.monitor']


The running result is:
```



7.3 Decorator `@every`

In default, step functions in BrainPy will be updated at every `dt`. However, in a real scenario, some step functions may have a updating period different from `dt`, for example, rest a variable at every `10 ms`. Therefore, BrainPy provides another useful scheduling decorator `@every(time)`, where `time` can be an int/float (denoting to update with a constant period), or can be a bool function (denoting to update with a varied period).

We illustrate this by also using the HH neuron model:

[19]:
```
bp.backend.set('numpy')
```

[20]:
```
class HH3(bp.NeuGroup):
    target_backend = 'numpy'

    def __init__(self, size, ENa=50., EK=-77., EL=-54.387,
                 C=1.0, gNa=120., gK=36., gL=0.03, V_th=20.,
                 **kwargs):
        super(HH3, self).__init__(size=size, **kwargs)

        # parameters
        self.ENa = ENa
        self.EK = EK
        self.EL = EL
        self.C = C
        self.gNa = gNa
        self.gK = gK
        self.gL = gL
        self.V_th = V_th

        # variables
```

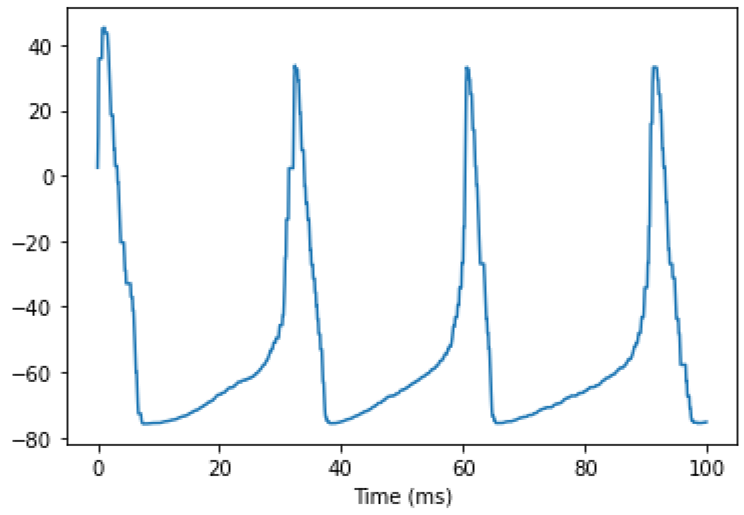
(continues on next page)

(continued from previous page)

```
        self.V = bp.ops.ones(self.num) * -65.
        self.m = bp.ops.ones(self.num) * 0.5
        self.h = bp.ops.ones(self.num) * 0.6
        self.n = bp.ops.ones(self.num) * 0.32
        self.spike = bp.ops.zeros(self.num)
        self.input = bp.ops.ones(self.num) * 10.

    @bp.odeint(method='exponential_euler')
    def integral(self, V, m, h, n, t, Iext):
        alpha = 0.1 * (V + 40) / (1 - bp.ops.exp(-(V + 40) / 10))
        beta = 4.0 * bp.ops.exp(-(V + 65) / 18)
        dmdt = alpha * (1 - m) - beta * m

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

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

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

        return dVdt, dmdt, dhdt, dndt

    @bp.every(time=lambda: np.random.random() < 0.5)
    def update(self, _t):
        V, m, h, n = self.integral(self.V, self.m, self.h, self.n, _t, self.input)
        self.spike = (self.V < self.V_th) * (V >= self.V_th)
        self.V = V
        self.m = m
        self.h = h
        self.n = n
```

In the HH3 model, at the top of `update()` function, the decorator `every()` receives a bool function. It represents at each `dt`, if the bool function returns "True", the corresponding step function will be updated, if "False", the step function will not be called. Therefore, `@bp.every(time=lambda: np.random.random() < 0.5)` denotes there is a 50% probability to run the `update()` function at each time step `dt`.

[21]:
```
group3 = HH3(1, monitors=['V'])

group3.run(100.)

bp.visualize.line_plot(group3.mon.V_t, group3.mon.V, show=True)
```



As you will see, the monitors will get the same values (variables not be updated) at the nearest neighborhood time points.

[22]:
```
group3.mon.V[:20]
```

[22]:
```
array([[ 2.44966378],
       [12.6364925 ],
       [35.82151319],
       [35.82151319],
       [35.82151319],
       [35.82151319],
       [35.82151319],
       [35.82151319],
       [45.01057186],
       [45.01057186],
       [45.01057186],
       [45.32911768],
       [43.67693542],
       [43.67693542],
       [43.67693542],
       [43.67693542],
       [43.67693542],
       [41.04038136],
       [37.57836768],
       [33.4323443 ]])
```

Author:

- Chaoming Wang
- Email: adaduo@outlook.com
- Date: 2021.05.25

CHAPTER

EIGHT

MORE ABOUT MONITOR

In BrainPy, each object (any instance of `brainpy.DynamicalSystem`) has the inherent monitor. Users can set up the monitor when initializing the brain objects. For example, if you have the following `HH` neuron model,

[1]:
```
import brainpy as bp

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

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

        # parameters
        self.ENa = ENa
        self.EK = EK
        self.EL = EL
        self.C = C
        self.gNa = gNa
        self.gK = gK
        self.gL = gL
        self.V_th = V_th

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

    @bp.odeint(method='exponential_euler')
    def integral(self, V, m, h, n, t, Iext):
        alpha = 0.1 * (V + 40) / (1 - bp.ops.exp(-(V + 40) / 10))
        beta = 4.0 * bp.ops.exp(-(V + 65) / 18)
        dmdt = alpha * (1 - m) - beta * m

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

(continues on next page)

(continued from previous page)

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

        I_Na = (self.gNa * m ** 3 * h) * (V - self.ENa)
        I_K = (self.gK * n ** 4) * (V - self.EK)
        I_leak = self.gL * (V - self.EL)
        dVdt = (- I_Na - I_K - I_leak + Iext) / self.C

        return dVdt, dmdt, dhdt, dndt

    def update(self, _t, _i, _dt):
        V, m, h, n = self.integral(self.V, self.m, self.h, self.n, _t, self.input)
        self.spike = (self.V < self.V_th) * (V >= self.V_th)
        self.V = V
        self.m = m
        self.h = h
        self.n = n
        self.input[:] = 0
```

the monitor can be set up when users create a HH neuron group:

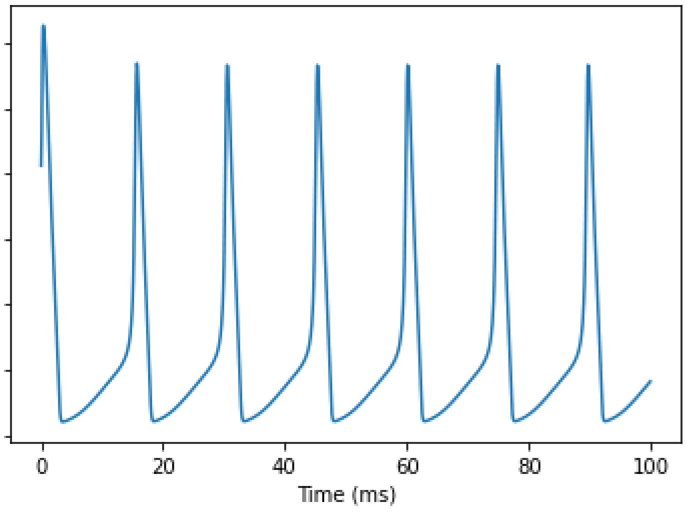
[2]:
```
# set up a monitor using a list/tuple of strings
group1 = HH(size=10, monitors=['V', 'spike'])

type(group1.mon)
```

[2]:
```
brainpy.simulation.monitors.Monitor
```

[3]:
```
# set up a monitor using the Monitor class
group2 = HH(size=10, monitors=bp.simulation.Monitor(variables=['V', 'spike']))
```

Once we run the given model/network, the monitors will record the evolution of variables in the corresponding neural or synaptic models.

[4]:
```
group1.run(100., inputs=('input', 10))

bp.visualize.line_plot(group1.mon.V_t, group1.mon.V, show=True)
```



8.1 Monitor variables at the selected index

However, we do not always take care of the all the content in a variable. We may be only interested in the values at the selected index. Moreover, for huge networks and long simulations, monitors will be a big part to consume RAM. Monitoring variables only at the selected index is a good solution. For these scenarios, we can initialize the monitors with the format of tuple/dict like this:

```
[5]: group3 = HH(
         size=10,
         monitors=['V', ('spike', [1, 2, 3])]  # use a tuple to specify the (key, index)
     )

     group3.run(100., inputs=('input', 10.))

     print(f'The monitor shape of "V" is (run length, variable size) = {group3.mon.V.shape}')
     print(f'The monitor shape of "spike" is (run length, index size) = {group3.mon.spike.
     ↪shape}')
```

```
The monitor shape of "V" is (run length, variable size) = (1000, 10)
The monitor shape of "spike" is (run length, index size) = (1000, 3)
```

Or, use a dictionary to specify the interested index of the variable:

```
[6]: group4 = HH(
         size=10,
         monitors={'V': None, 'spike': [1, 2, 3]}  # use a dict to specify the {key: index}
     )

     group4.run(100., inputs=('input', 10.))

     print(f'The monitor shape of "V" is (run length, variable size) = {group4.mon.V.shape}')
     print(f'The monitor shape of "spike" is (run length, index size) = {group4.mon.spike.
     ↪shape}')
```

```
The monitor shape of "V" is (run length, variable size) = (1000, 10)
The monitor shape of "spike" is (run length, index size) = (1000, 3)
```

Also, an instance of `Monitor` class can also be used:

[7]:
```
group5 = HH(
    size=10,
    # monitors=bp.simulation.Monitor(variables=['V', ('spike', [1, 2, 3])])
    monitors=bp.simulation.Monitor(variables={'V': None, 'spike': [1, 2, 3]})  #␣
↪initialize a Monitor
                                                                              # to␣
↪specify the key-index
)

group5.run(100., inputs=('input', 10.))

print(f'The monitor shape of "V" is (run length, variable size) = {group5.mon.V.shape}')
print(f'The monitor shape of "spike" is (run length, index size) = {group5.mon.spike.
↪shape}')
```

```
The monitor shape of "V" is (run length, variable size) = (1000, 10)
The monitor shape of "spike" is (run length, index size) = (1000, 3)
```

8.2 Monitor variables with customized period

In long simulations with small `dt` time step, what we take care about is the trend of the variable evolution, not the exact values at each time point (especially when `dt` is very small). For this scenario, we can initializing the monitors with the `every` item specification (similar to the decorator `@brainpy.every(time=...)`):

[8]:
```
group6 = HH(
    size=10,
    monitors=bp.simulation.Monitor(variables={'V': None, 'spike': [1, 2, 3]},
                                   every={'V': None, 'spike': 1.})
)
```

In this example, we monitor “spike” variables at the index of [1, 2, 3] for each `1 ms`.

[9]:
```
group6.run(100., inputs=('input', 10.))

print(f'The monitor shape of "V" = {group6.mon.V.shape}')
print(f'The monitor shape of "spike" = {group6.mon.spike.shape}')
```

```
The monitor shape of "V" = (1000, 10)
The monitor shape of "spike" = (100, 3)
```

But what is different from the decorator `@brainpy.every(time=...)` is that the `time` can not receive a bool function. This is because the monitors will allocate the data need to record in advance. But the bool function makes the beforehand allocation more difficult.

Author:

- Chaoming Wang

- Email: adaduo@outlook.com
- Date: 2021.05.24

CHAPTER

NINE

REPEAT RUNNING MODE

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 repetitively. Specifically, each brainpy.Network is instantiated in the repeat mode by default. The parameters and the variables in the step functions can be arbitrarily changed without the need to recompile the network models.

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

9.1 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 brainpy as bp

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

[2]:
```
bp.backend.set('numba', dt=0.04)
bp.integrators.set_default_odeint('exponential_euler')
```

We first define the HH neuron model:

[3]:
```
# Neuron Group #
# ------------ #

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

    def __init__(self, size, ENa=55., EK=-90., EL=-65,
                 C=1.0, gNa=35., gK=9., gL=0.1, V_th=20.,
                 phi=5.0, **kwargs):
```

(continues on next page)

(continued from previous page)

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

        # parameters
        self.ENa = ENa
        self.EK = EK
        self.EL = EL
        self.C = C
        self.gNa = gNa
        self.gK = gK
        self.gL = gL
        self.V_th = V_th
        self.phi = phi

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

    def reset(self):
        self.V[:] = -70. + np.random.random(self.num) * 20

        h_alpha = 0.07 * np.exp(-(self.V + 58) / 20)
        h_beta = 1 / (np.exp(-0.1 * (self.V + 28)) + 1)
        self.h[:] = h_alpha / (h_alpha + h_beta)

        n_alpha = -0.01 * (self.V + 34) / (np.exp(-0.1 * (self.V + 34)) - 1)
        n_beta = 0.125 * np.exp(-(self.V + 44) / 80)
        self.n[:] = n_alpha / (n_alpha + n_beta)

        self.spike[:] = False
        self.input[:] = 0.

    @staticmethod
    @bp.odeint
    def integral(V, h, n, t, Iext, gNa, ENa, gK, EK, gL, EL, C, phi):
        alpha = 0.07 * bp.ops.exp(-(V + 58) / 20)
        beta = 1 / (bp.ops.exp(-0.1 * (V + 28)) + 1)
        dhdt = phi * (alpha * (1 - h) - beta * h)

        alpha = -0.01 * (V + 34) / (bp.ops.exp(-0.1 * (V + 34)) - 1)
        beta = 0.125 * bp.ops.exp(-(V + 44) / 80)
        dndt = phi * (alpha * (1 - n) - beta * n)

        m_alpha = -0.1 * (V + 35) / (bp.ops.exp(-0.1 * (V + 35)) - 1)
        m_beta = 4 * bp.ops.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 = gL * (V - EL)
        dVdt = (- INa - IK - IL + Iext) / C
```

(continues on next page)

(continued from previous page)

```
        return dVdt, dhdt, dndt

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

Then, the inter-connected synapse GABAA can be defined as:

[4]:
```
# GABAa Synapse #
# ------------- #

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

    def __init__(self, pre, post, conn, delay=0., E=-75., g_max=0.1,
                 alpha=12., beta=0.1, T=1.0, T_duration=1.0, **kwargs):
        # parameters
        self.E = E
        self.alpha = alpha
        self.beta = beta
        self.T = T
        self.T_duration = T_duration
        self.delay = delay
        self.g_max = g_max

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

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

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

    def reset(self):
        self.s[:] = 0.
        self.t_last_pre_spike[:] = -1e7
        self.g.reset()

    @staticmethod
    @bp.odeint
    def int_s(s, t, TT, alpha, beta):
```

(continues on next page)

(continued from previous page)

```
        dsdt = alpha * TT * (1 - s) - beta * s
        return dsdt

    def update(self, _t):
        for i in range(self.pre.size[0]):
            if self.pre.spike[i] > 0:
                self.t_last_pre_spike[i] = _t
        TT = ((_t - self.t_last_pre_spike) < self.T_duration) * self.T
        self.s = self.int_s(self.s, _t, TT, self.alpha, self.beta)
        self.g.push(self.g_max * self.s)
        g = self.g.pull()
        self.post.input -= bp.ops.sum(g, axis=0) * (self.post.V - self.E)
```

Putting the HH neuron and the GABAA syanpse together, let's define the network in which HH neurons are interconnected with the GABAA syanpses.

[5]:
```
# Network #
# ------- #

num = 100

group = HH(num, monitors=['spike', 'V'])
conn = GABAa(pre=group, post=group, g_max=0.1/num,
             conn=bp.connect.All2All(include_self=False))
net = bp.Network(group, conn)
```

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

[6]:
```
# 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.reset()
    conn.reset()

    net.run(duration=500., inputs=[group, 'input', 1.2], report=True, report_percent=1.)

    cc = bp.measure.cross_correlation(group.mon.spike, bin=0.5)
    all_cc.append(cc)
```

```
When g_max = 0.00050 ...
Compilation used 5.6063 s.
Start running ...
Run 100.0% used 1.334 s.
Simulation is done in 1.334 s.

When g_max = 0.00060 ...
```

(continues on next page)

(continued from previous page)

```
Compilation used 0.0000 s.
Start running ...
Run 100.0% used 1.332 s.
Simulation is done in 1.332 s.

When g_max = 0.00070 ...
Compilation used 0.0000 s.
Start running ...
Run 100.0% used 1.353 s.
Simulation is done in 1.353 s.

When g_max = 0.00080 ...
Compilation used 0.0000 s.
Start running ...
Run 100.0% used 1.373 s.
Simulation is done in 1.373 s.

When g_max = 0.00090 ...
Compilation used 0.0000 s.
Start running ...
Run 100.0% used 1.354 s.
Simulation is done in 1.354 s.

When g_max = 0.00100 ...
Compilation used 0.0000 s.
Start running ...
Run 100.0% used 1.358 s.
Simulation is done in 1.358 s.

When g_max = 0.00110 ...
Compilation used 0.0010 s.
Start running ...
Run 100.0% used 1.364 s.
Simulation is done in 1.364 s.

When g_max = 0.00120 ...
Compilation used 0.0000 s.
Start running ...
Run 100.0% used 1.352 s.
Simulation is done in 1.352 s.

When g_max = 0.00130 ...
Compilation used 0.0000 s.
Start running ...
Run 100.0% used 1.348 s.
Simulation is done in 1.348 s.

When g_max = 0.00140 ...
Compilation used 0.0000 s.
Start running ...
Run 100.0% used 1.350 s.
Simulation is done in 1.350 s.
```

(continues on next page)

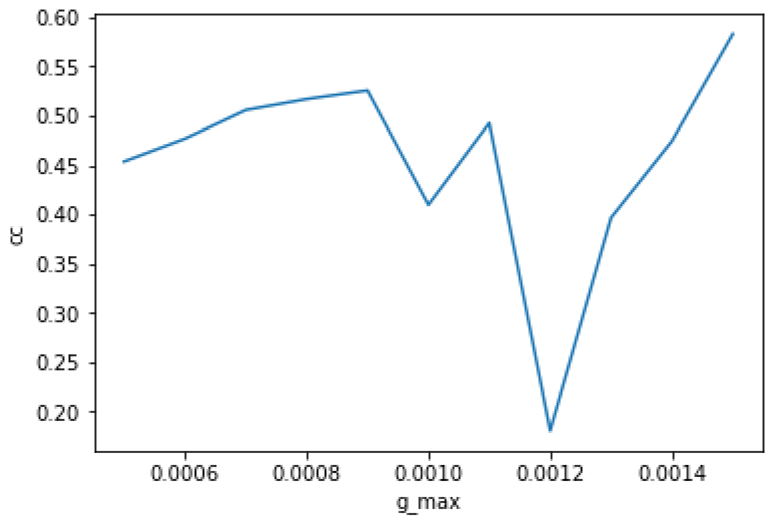
(continued from previous page)

```
When g_max = 0.00150 ...
Compilation used 0.0000 s.
Start running ...
Run 100.0% used 1.353 s.
Simulation is done in 1.353 s.
```

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`.

[7]:
```
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 variable state of the neuron group and the synaptic 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 variable state should not be reset.

9.2 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()`. In this section, we illustrate this function by using the above defined gamma oscillation network.

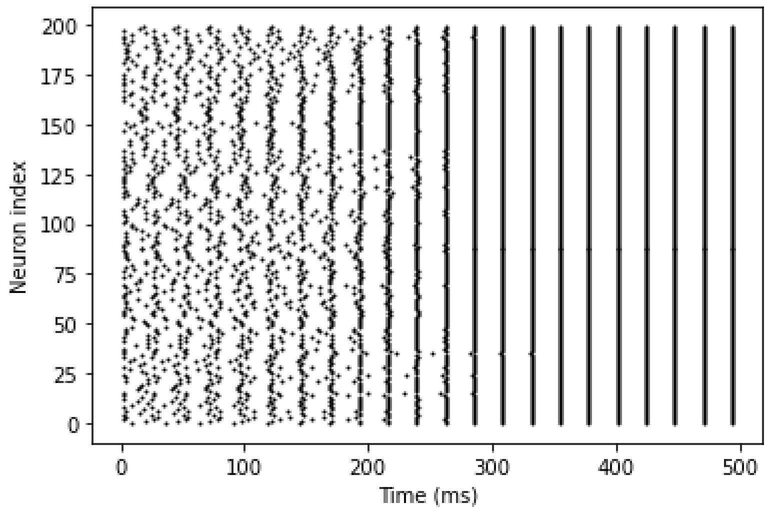
We define a network with the size of 200 HH neurons, and try to run this network in 2 seconds.

[8]:
```
group2 = HH(200, monitors=['spike', 'V'])
group2.reset()

conn2 = GABAa(pre=group2, post=group2, g_max=0.1/200,
              conn=bp.connect.All2All(include_self=False))
net2 = bp.Network(group2, conn2)
```

Here, we do not run the total 1.5 second at one time. On the contrary, we run the model with four steps, with each step of 0.5 second running duration.

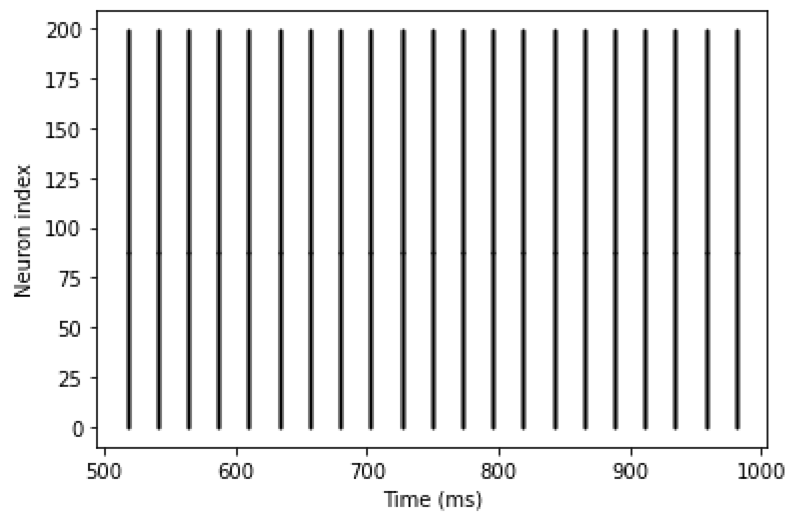
[9]:
```
# run 1: 0 - 500 ms

net2.run(duration=(0., 500.), inputs=[group2, 'input', 1.2], report=True, report_
↪percent=0.5)
bp.visualize.raster_plot(group2.mon.spike_t, group2.mon.spike, show=True)
```

```
Compilation used 2.1291 s.
Start running ...
Run 50.0% used 2.095 s.
Run 100.0% used 4.187 s.
Simulation is done in 4.187 s.
```



[10]:
```
# run 2: 500 - 1000 ms

net2.run(duration=(500., 1000.), inputs=[group2, 'input', 1.2], report=True, report_
↪percent=0.5)
bp.visualize.raster_plot(group2.mon.spike_t, group2.mon.spike, show=True)
```

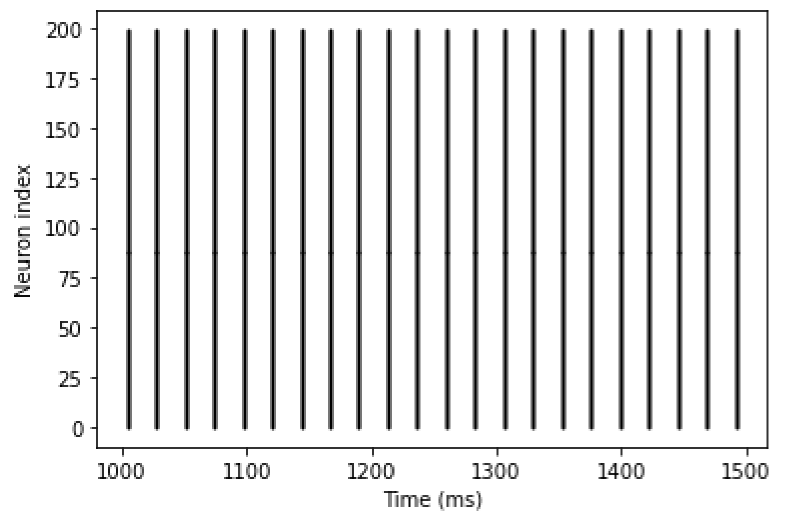
```
Compilation used 0.0000 s.
Start running ...
Run 50.0% used 2.128 s.
Run 100.0% used 4.249 s.
Simulation is done in 4.249 s.
```



Set a different inputs structure (the previous is a "float", now it's a vector), the network will rebuild the input function and run the network continuously.

[11]:
```
# run 1000 - 1500 ms

Iext = np.ones(group2.num) * 1.2
net2.run(duration=(1000., 1500.), inputs=[group2, 'input', Iext],
        report=True, report_percent=0.5)
bp.visualize.raster_plot(group2.mon.spike_t, group2.mon.spike, show=True)
```

```
Compilation used 0.0010 s.
Start running ...
Run 50.0% used 2.134 s.
Run 100.0% used 4.284 s.
Simulation is done in 4.284 s.
```



NOTE

Another thing worthy noting is that if the model variable states rely on the time (for example, the LIF neuron model `self.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`.

Author:

- Chaoming Wang
- Email: adaduo@outlook.com
- Date: 2021.05.25

CHAPTER

TEN

UNIFIED OPERATIONS

BrainPy is targeted on multiple backends. Flexible switch between various backends needs to solve the problem of how to unify the different operations between them.

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

Intrinsically, BrainPy needs several necessary operations for numerical solvers, dynamics simulation and data construction.

[2]:
```
# necessary operations for numerical solvers

bp.ops.OPS_FOR_SOLVER
```

[2]:
```
['normal', 'sum', 'exp', 'shape']
```

[3]:
```
# necessary operations for neurodynamics simulation

bp.ops.OPS_FOR_SIMULATION
```

[3]:
```
['as_tensor', 'zeros', 'ones', 'arange', 'concatenate', 'where', 'reshape']
```

[4]:
```
# necessary data types

bp.ops.OPS_OF_DTYPE
```

[4]:
```
['bool', 'int', 'int32', 'int64', 'float', 'float32', 'float64']
```

However, if want to unify more commonly used operations, users can use `brainpy.ops.set_buffer(backend, **operations)` to set operation buffers.

For example, if users want to implement unified `clip` and `square` operations across different backends, ones can define like this:

[5]:
```
# NumPy

import numpy as np

bp.ops.set_buffer('numpy', clip=np.clip, sqrt=np.sqrt)
```

[6]:
```
# PyTorch

try:
    import torch
```

(continues on next page)

(continued from previous page)

```
    bp.ops.set_buffer('pytorch', clip=torch.clamp, sqrt=torch.sqrt)

except ModuleNotFoundError:
    pass
```

[7]:
```
# TensorFlow

try:
    import tensorflow as tf

    bp.ops.set_buffer('tensorflow', clip=tf.clip_by_value, sqrt=tf.math.sqrt)

except ModuleNotFoundError:
    pass
```

[8]:
```
# Numba

try:
    import numba as nb

    @nb.njit
    def nb_clip(x, x_min, x_max):
        x = np.maximum(x, x_min)
        x = np.minimum(x, x_max)
        return x

    bp.ops.set_buffer('numba', clip=nb_clip, sqrt=np.sqrt)
    bp.ops.set_buffer('numba-parallel', clip=nb_clip, sqrt=np.sqrt)

except ModuleNotFoundError:
    pass
```

[9]:
```
# Numba-CUDA

try:
    import math
    import numba as nb
    from numba import cuda

    @cuda.jit(devicde=True)
    def cuda_clip(x, x_min, x_max):
        if x < x_min: return x_min
        elif x > x_max: return x_max
        else: return x

    bp.ops.set_buffer('numba-cuda', clip=nb_clip, sqrt=math.sqrt)

except ModuleNotFoundError:
    pass
```

After the buffer setting, users can use the unified operation to define models which will automatically works natively with buffered backends.

```
[10]: def test(arr):
          return bp.ops.sqrt(bp.ops.clip(arr, 0., 1.))
```

```
[11]: bp.backend.set('numpy')

      test(bp.ops.as_tensor([-1, 0.5, 2.]))
```

```
[11]: array([0.        , 0.70710678, 1.        ])
```

```
[12]: bp.backend.set('pytorch')

      test(bp.ops.as_tensor([-1, 0.5, 2.]))
```

```
[12]: tensor([0.0000, 0.7071, 1.0000])
```

```
[13]: bp.backend.set('tensorflow')

      test(bp.ops.as_tensor([-1, 0.5, 2.]))
```

```
[13]: <tf.Tensor: shape=(3,), dtype=float32, numpy=array([0.        , 0.70710677, 1.        ],␣
      ↪dtype=float32)>
```

```
[14]: bp.backend.set('numba')

      test(bp.ops.as_tensor([-1, 0.5, 2.]))
```

```
[14]: array([0.        , 0.70710678, 1.        ])
```

Author:

- Chaoming Wang
- Email: adaduo@outlook.com
- Date: 2021.05.26

CHAPTER

ELEVEN

NUMERICAL SOLVERS FOR ODES

BrainPy provides several numerical methods for ordinary differential equations (ODEs). Specifically, we provide explicit Runge-Kutta methods, adaptive Runge-Kutta methods, and Exponential Euler method for ODE numerical integration. For the introductionary tutorial for how to use BrainPy provided numerical solvers for ODEs, please see the document in *Quickstart/Numerical Solvers*.

[1]:
```
import brainpy as bp

bp.__version__
```

[1]:
```
'1.0.3'
```

11.1 Explicit Runge-Kutta methods

The first category of ODE numerical integration support is the explicit Runge-Kutta (RK) methods. RK methods are a huge family of numerical methods with a wide variety of trade-offs: efficiency, accuracy, stability, etc. The supported RK methods are listed in the following table:

Methods	Keywords
Euler	euler
Midpoint	midpoint
Heun’s second-order method	heun2
Ralston’s second-order method	ralston2
RK2	rk2
RK3	rk3
RK4	rk4
Heun’s third-order method	heun3
Ralston’s third-order method	ralston3
Third-order Strong Stability Preserving Runge-Kutta	ssprk3
Ralston’s fourth-order method	ralston4
Runge-Kutta 3/8-rule fourth-order method	rk4_38rule

Users can utilize these methods by specify the `method` option in `brainpy.odeint()` with their corresponding keyword. For example:

[2]:
```
@bp.odeint(method='rk4')
def int_v(v, t, p):
    # do something
    return v
```

11.2 Adaptive Runge-Kutta methods

The second category of ODE numerical support is the adaptive RK methods. What's different from the explicit RK methods is that adaptive methods are designed to produce an estimate of the local truncation error in a single Runge-Kutta step, then such error can be used to adaptively control the numerical step size. Specifically, if $error > tol$, then replace dt with dt_{new} and repeat the step. Therefore, adaptive RK methods allow the varied step size. In BrainPy, the following adaptive RK methods are provided:

Methods	keywords
Runge–Kutta–Fehlberg 4(5)	rkf45
Runge–Kutta–Fehlberg 1(2)	rkf12
Dormand–Prince method	rkdp
Cash–Karp method	ck
Bogacki–Shampine method	bs
Heun–Euler method	heun_euler

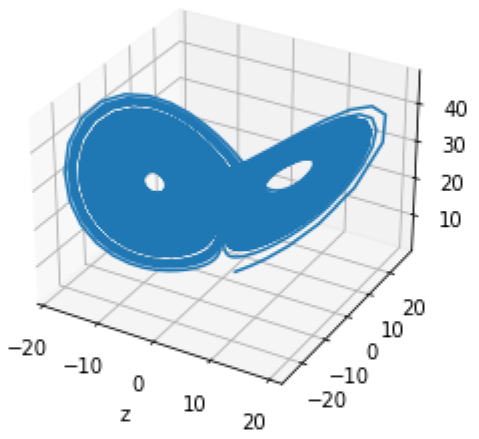
In default, the above methods are not adaptive, unless users provide a keyword `adaptive=True` in `brainpy.odeint()`. When users use the adaptive RK methods for numerical integration, the instantaneously adjusted stepsize `dt` will be appended in the functional arguments. Moreover, the tolerance `tol` for stepsize adjustment can also be controlled by users. Let's take the Lorenz system as the example:

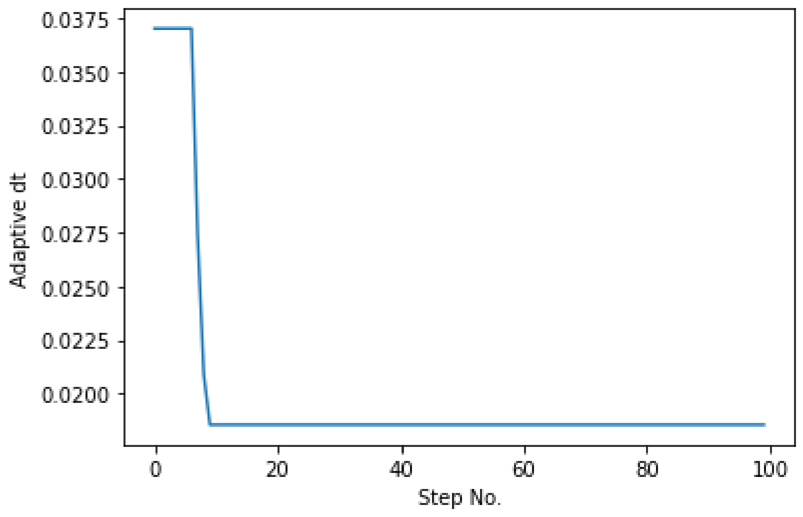
[3]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[4]:
```
# adaptively adjust stepsize

@bp.odeint(method='rkf45',
           adaptive=True, # active the "adaptive" option
           tol=0.001) # set the tolerance
def lorenz(x, y, z, t, sigma, beta, rho):
    dx = sigma * (y - x)
    dy = x * (rho - z) - y
    dz = x * y - beta * z
    return dx, dy, dz
```

[5]:
```
times = np.arange(0, 100, 0.01)
hist_x, hist_y, hist_z, hist_dt = [], [], [], []
x, y, z, dt = 1, 1, 1, 0.05
for t in times:
    # should provide one more argument "dt" when using the adaptive rk method
    x, y, z, dt = lorenz(x, y, z, t, sigma=10, beta=8/3, rho=28, dt=dt)
    hist_x.append(x)
    hist_y.append(y)
    hist_z.append(z)
    hist_dt.append(dt)
hist_x = np.array(hist_x)
hist_y = np.array(hist_y)
hist_z = np.array(hist_z)
hist_dt = np.array(hist_dt)
```

[6]:
```
fig = plt.figure()
ax = fig.gca(projection='3d')
plt.plot(hist_x, hist_y, hist_z)
ax.set_xlabel('x')
ax.set_xlabel('y')
ax.set_xlabel('z')

fig = plt.figure()
plt.plot(hist_dt[:100])
plt.xlabel('Step No.')
plt.ylabel('Adaptive dt')
plt.show()
```





However, when `adaptive=True` is not set, users cannot call numerical function with the adaptively changed `dt`.

[7]:
```
# not adaptive

@bp.odeint(method='rkf45')
def lorenz_non_adaptive(x, y, z, t, sigma, beta, rho):
    dx = sigma * (y - x)
```

(continues on next page)

(continued from previous page)

```
    dy = x * (rho - z) - y
    dz = x * y - beta * z
    return dx, dy, dz

lorenz_non_adaptive(x=1., y=1., z=1., t=0., sigma=10, beta=8/3, rho=28, dt=dt)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-32a7a847de20> in <module>
      8     return dx, dy, dz
      9
---> 10 lorenz_non_adaptive(x=1., y=1., z=1., t=0., sigma=10, beta=8/3, rho=28, dt=dt)

TypeError: ode_brainpy_intg_of_lorenz_non_adaptive() got an unexpected keyword argument
↪'dt'
```

11.3 Exponential Euler methods

Finally, BrainPy provides Exponential Euler method for ODEs. For you linear ODE systems, we highly recommend you to to use Exponential Euler methods.

Methods	keywords
Exponential Euler	exponential_euler

For a linear system,

$$\frac{dy}{dt} = A - By$$

the exponential Euler schema is given by:

$$y(t+dt) = y(t)e^{-B*dt} + \frac{A}{B}(1 - e^{-B*dt})$$

As you can see, for such linear systems, the exponential Euler schema is nearly the exact solution.

In BrainPy, in order to automatically find out the linear part, we will utilize the SymPy to parse user defined functions. Therefore, ones need install `sympy` first when using exponential Euler method.

What's interesting, the computational expensive neuron model Hodgkin–Huxley model is a linear ODE system. In the next, you will find that by using Exponential Euler method, the numerical step can be enlarged much to save the computation time.

$$\begin{aligned}
C_m\frac{dV}{dt} &= -\left[\bar{g}_{\text{K}}n^4 + \bar{g}_{\text{Na}}m^3h + \bar{g}_l\right]V + \bar{g}_{\text{K}}n^4V_K + \bar{g}_{\text{Na}}m^3hV_{Na} + \bar{g}_lV_l + I_{syn} \\
\frac{dm}{dt} &= \left[-\alpha_m(V) - \beta_m(V)\right]m \qquad + \alpha_m(V) \\
\frac{dh}{dt} &= \left[-\alpha_h(V) - \beta_h(V)\right]h \qquad + \alpha_h(V) \\
\frac{dn}{dt} &= \left[-\alpha_n(V) - \beta_n(V)\right]n \qquad + \alpha_n(V)
\end{aligned}$$

[8]:
```
Iext=10.;   ENa=50.;   EK=-77.;   EL=-54.387
C=1.0;      gNa=120.;  gK=36.;    gL=0.03
```

[9]:
```
def derivative(V, m, h, n, t, Iext, gNa, ENa, gK, EK, gL, EL, C):
    alpha = 0.1 * (V + 40) / (1 - bp.ops.exp(-(V + 40) / 10))
    beta = 4.0 * bp.ops.exp(-(V + 65) / 18)
    dmdt = alpha * (1 - m) - beta * m

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

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

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

    return dVdt, dmdt, dhdt, dndt
```

[10]:
```
def run(method, Iext=10.):
    hist_times = np.arange(0, 100, method.dt)
    hist_V, hist_m, hist_h, hist_n = [], [], [], []
    V, m, h, n = 0., 0., 0., 0.
    for t in hist_times:
        V, m, h, n = method(V, m, h, n, t, Iext, gNa, ENa, gK, EK, gL, EL, C)
        hist_V.append(V)
        hist_m.append(m)
        hist_h.append(h)
        hist_n.append(n)

    plt.subplot(211)
    plt.plot(hist_times, hist_V, label='V')
    plt.legend()
    plt.subplot(212)
    plt.plot(hist_times, hist_m, label='m')
    plt.plot(hist_times, hist_h, label='h')
    plt.plot(hist_times, hist_n, label='n')
    plt.legend()
```

Euler Method

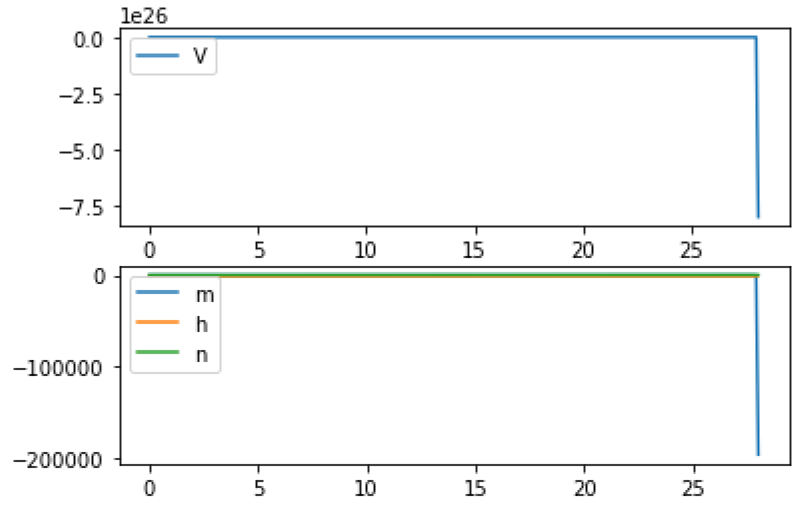
[11]:
```
int1 = bp.odeint(f=derivative, method='euler', dt=0.1)

run(int1, Iext=10)
```

```
<ipython-input-9-e2231649300f>:2: RuntimeWarning: overflow encountered in exp
  alpha = 0.1 * (V + 40) / (1 - bp.ops.exp(-(V + 40) / 10))
<ipython-input-9-e2231649300f>:3: RuntimeWarning: overflow encountered in exp
  beta = 4.0 * bp.ops.exp(-(V + 65) / 18)
```

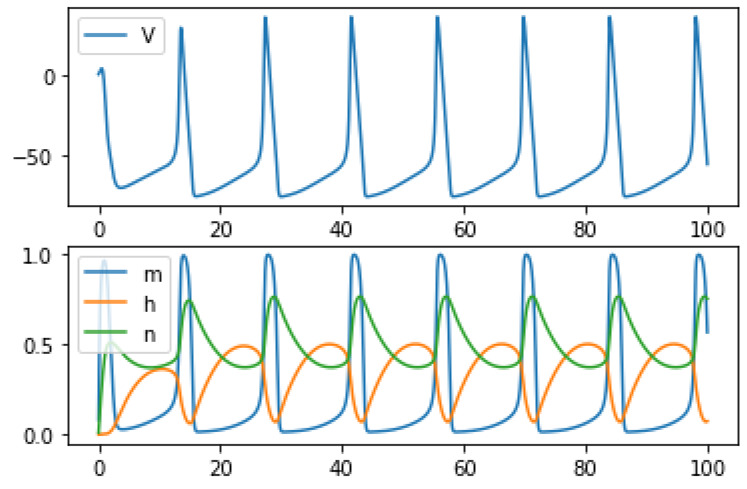
(continues on next page)

(continued from previous page)

```
<ipython-input-9-e2231649300f>:6: RuntimeWarning: overflow encountered in exp
  alpha = 0.07 * bp.ops.exp(-(V + 65) / 20.)
<ipython-input-9-e2231649300f>:7: RuntimeWarning: overflow encountered in exp
  beta = 1 / (1 + bp.ops.exp(-(V + 35) / 10))
<ipython-input-9-e2231649300f>:10: RuntimeWarning: overflow encountered in exp
  alpha = 0.01 * (V + 55) / (1 - bp.ops.exp(-(V + 55) / 10))
<ipython-input-9-e2231649300f>:11: RuntimeWarning: overflow encountered in exp
  beta = 0.125 * bp.ops.exp(-(V + 65) / 80)
<ipython-input-9-e2231649300f>:4: RuntimeWarning: invalid value encountered in double_
↪scalars
  dmdt = alpha * (1 - m) - beta * m
<ipython-input-9-e2231649300f>:8: RuntimeWarning: invalid value encountered in double_
↪scalars
  dhdt = alpha * (1 - h) - beta * h
<ipython-input-9-e2231649300f>:12: RuntimeWarning: invalid value encountered in double_
↪scalars
  dndt = alpha * (1 - n) - beta * n
```

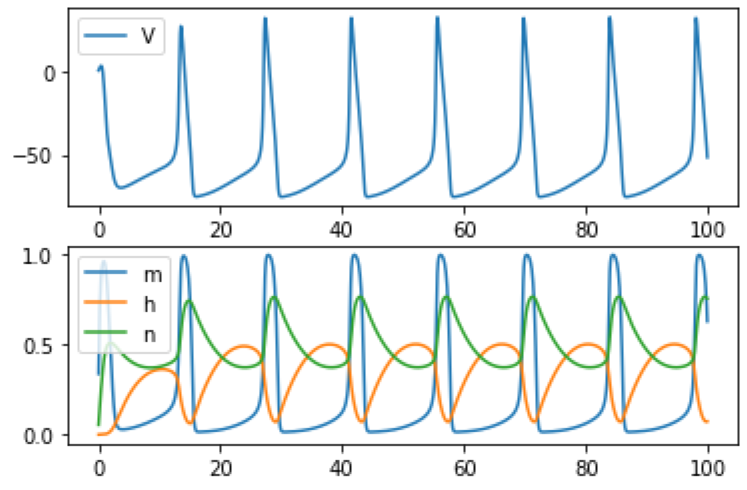


[12]:
```
int2 = bp.odeint(f=derivative, method='euler', dt=0.02)

run(int2, Iext=10)
```



RK4 Method

```
[13]: int3 = bp.odeint(f=derivative, method='rk4', dt=0.1)

      run(int3, Iext=10)
```



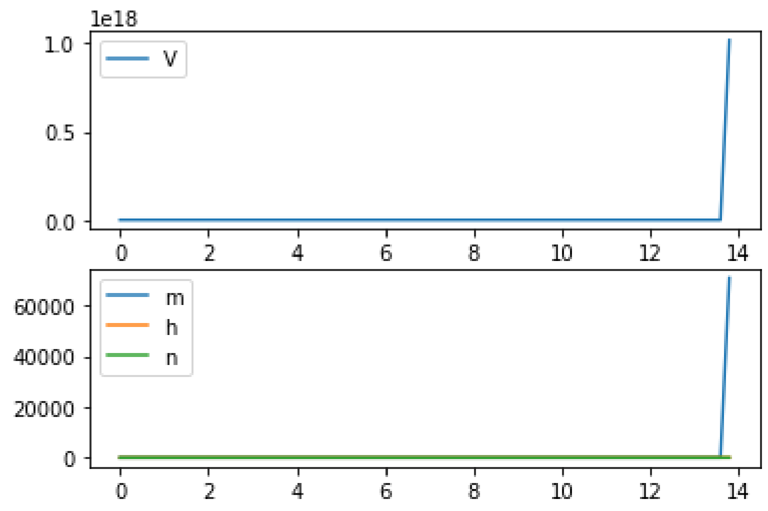
```
[14]: int4 = bp.odeint(f=derivative, method='rk4', dt=0.2)

      run(int4, Iext=10)
```

```
<ipython-input-9-e2231649300f>:2: RuntimeWarning: overflow encountered in exp
  alpha = 0.1 * (V + 40) / (1 - bp.ops.exp(-(V + 40) / 10))
<ipython-input-9-e2231649300f>:3: RuntimeWarning: overflow encountered in exp
  beta = 4.0 * bp.ops.exp(-(V + 65) / 18)
<ipython-input-9-e2231649300f>:6: RuntimeWarning: overflow encountered in exp
  alpha = 0.07 * bp.ops.exp(-(V + 65) / 20.)
<ipython-input-9-e2231649300f>:7: RuntimeWarning: overflow encountered in exp
  beta = 1 / (1 + bp.ops.exp(-(V + 35) / 10))
<ipython-input-9-e2231649300f>:10: RuntimeWarning: overflow encountered in exp
  alpha = 0.01 * (V + 55) / (1 - bp.ops.exp(-(V + 55) / 10))
```

(continues on next page)

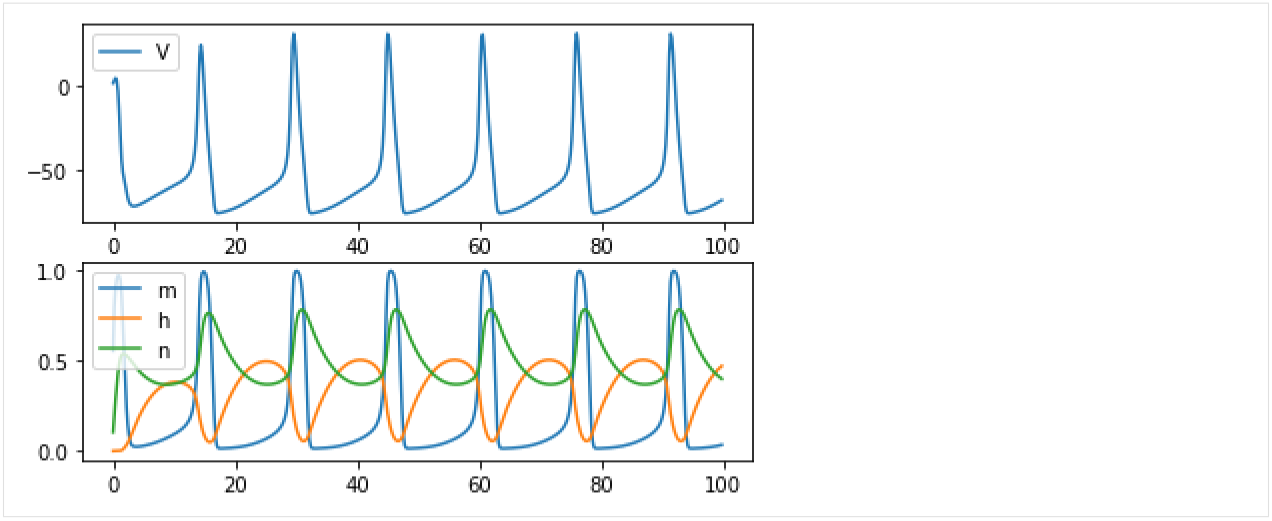
(continued from previous page)

```
<ipython-input-9-e2231649300f>:11: RuntimeWarning: overflow encountered in exp
  beta = 0.125 * bp.ops.exp(-(V + 65) / 80)
<ipython-input-9-e2231649300f>:4: RuntimeWarning: invalid value encountered in double_
↪scalars
  dmdt = alpha * (1 - m) - beta * m
<ipython-input-9-e2231649300f>:8: RuntimeWarning: invalid value encountered in double_
↪scalars
  dhdt = alpha * (1 - h) - beta * h
<ipython-input-9-e2231649300f>:12: RuntimeWarning: invalid value encountered in double_
↪scalars
  dndt = alpha * (1 - n) - beta * n
<ipython-input-9-e2231649300f>:17: RuntimeWarning: invalid value encountered in double_
↪scalars
  dVdt = (- I_Na - I_K - I_leak + Iext) / C
```



Exponential Euler Method

```
[15]: int5 = bp.odeint(f=derivative, method='exponential_euler', dt=0.2)

run(int5, Iext=10)
```



Author:

- Chaoming Wang
- Email: adaduo@outlook.com
- Date: 2021.05.29

CHAPTER

TWELVE

NUMERICAL SOLVERS FOR SDES

BrainPy provides several numerical methods for stochastic differential equations (SDEs). Specifically, we provide explicit Runge-Kutta methods, derivative-free Milstein methods, and exponential Euler method for SDE numerical integration. For the introductionary tutorial for how to use BrainPy provided numerical solversfor SDEs, please see the document in *Quickstart/Numerical Solvers*.

```
[1]: import brainpy as bp

     bp.__version__
```

```
[1]: '1.0.3'
```

Methods	Keywords	Ito SDE support	Stratonovich SDE support	Scalar Wiener support	Vector Wiener support
Strong SRK scheme: SRI1W1	srk1w1_scalar	Yes		Yes	
Strong SRK scheme: SRI2W1	srk2w1_scalar	Yes		Yes	
Strong SRK scheme: KlPl	KlPl_scalar	Yes		Yes	
Euler method	euler	Yes	Yes	Yes	Yes
Heun method	heun		Yes	Yes	Yes
Derivative-free Milstein	milstein	Yes	Yes	Yes	Yes
Exponential Euler	exponential_euler	Yes		Yes	Yes

Author:

- Chaoming Wang
- Email: adaduo@outlook.com
- Date: 2021.05.29

CHAPTER

THIRTEEN

HOW BRAINPY WORKS

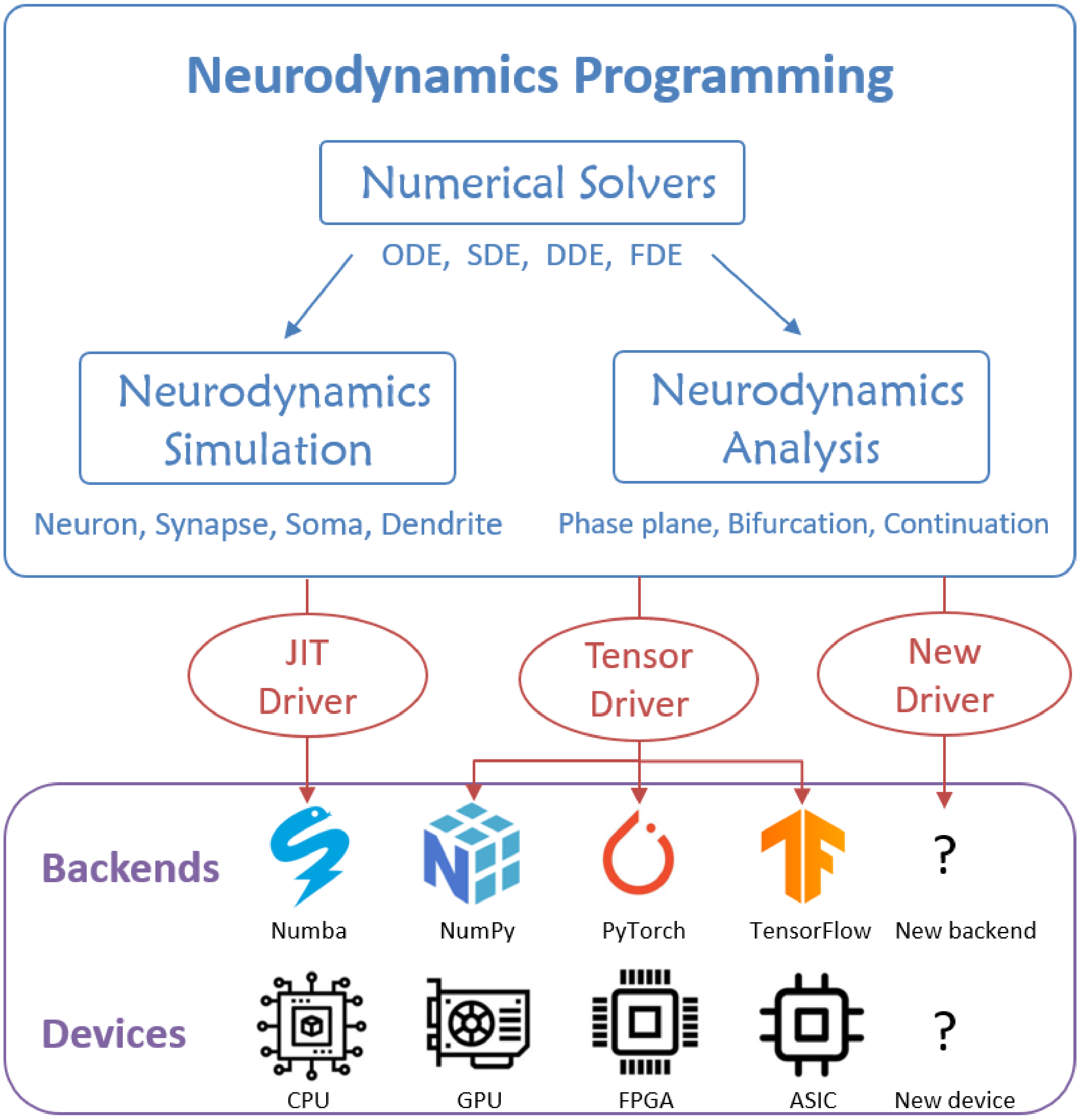
13.1 Design philosophy

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 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 did 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.
- **Simulation with the guidance of the analysis**. Simulation, although very important in neurodynamics, has to be guided by theory. Blind simulation of nonlinear systems is likely to produce few results or misleading results. A "brute force" simulation approach is hardly effective and accurate in practice. For example, *attractors* and *repellors* can be easily obtained through simulation, forward and backward in time, while *saddles* can be hard to find. Moreover, some complex dynamical equations are hard to analyze by hand, and can only be analyzed by computer optimization. However, current popular neural simulators (including NEURON, NEST, BRIAN, etc.) cannot give any insights for the defined dynamical models. Traditionally, users need define the models twice by two different ways, one for simulation in neural simulators, and one for model analysis in other programming language. There, BrainPy provides an integrated simulation and analysis environment for neuronal dynamics. The same codes defined by users can not only be used for simulation, but also for neurodyanmics analysis.
- **Running 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 (such as Numba) into BrainPy.

13.2 Framework Architecture

BrainPy is designed to support general computing backends on multiple devices. The overall architecture is shown in the following figure:



BrainPy is aimed to provide a general operating system for neuronal dynamics programming. At the first step, BrainPy provides general numerical solvers for ordinary differential equations (ODEs), stochastic differential equations (SDEs), delay differential equations (DDEs), fractional differential equations (FDEs). Then, based on the numerical solvers of various DEs, at one hand, BrainPy supports neuronal dynamics simulation, at the other hand, such general coded differential equations can be used for neuronal dynamics analysis.

Behind the programming interface, BrainPy provides various drivers for different backends. For example, when users run models on NumPy or PyTorch backend (`brainpy.backend.set('numpy')` or `brainpy.backend.`

`set('pytorch')`), the tensor-based drivers will be automatically utilized. Moreover, BrainPy support JIT compilers (such as Numba, JAX) because of the powerful JIT drivers. For the other accessible devices and corresponding backends in the future, BrainPy can have a seamless support because of the new driver can be defined.

Author:

- Chaoming Wang
- Email: adaduo@outlook.com
- Date: 2020.10.01, updated at 2021.05.29

CHAPTER

FOURTEEN

BRAINPY.ANALYSIS MODULE

Contents

14.1 Summary

`PhasePlane`(integrals, target_vars[, . . .])	A tool class for phase plane analyzer.
`Bifurcation`(integrals, target_pars, target_vars)	A tool class for bifurcation analysis.
`FastSlowBifurcation`(integrals, fast_vars, . . .)	Fast slow analysis analysis proposed by John Rinzel[1][2][3].
`get_1d_stability_types`()	Get the stability types of 1D system.
`get_2d_stability_types`()	Get the stability types of 2D system.
`get_3d_stability_types`()	Get the stability types of 3D system.
`stability_analysis`(derivatives)	Stability analysis for fixed points.

[1] Rinzel, John. "Bursting oscillations in an excitable membrane model." In Ordinary and partial differential equations, pp. 304-316. Springer, Berlin, Heidelberg, 1985.

[2] Rinzel, John , and Y. S. Lee . On Different Mechanisms for Membrane Potential Bursting. Nonlinear Oscillations in Biology and Chemistry. Springer Berlin Heidelberg, 1986.

[3] Rinzel, John. "A formal classification of bursting mechanisms in excitable systems." In Mathematical topics in population biology, morphogenesis and neurosciences, pp. 267-281. Springer, Berlin, Heidelberg, 1987.

14.1.1 brainpy.analysis.PhasePlane

class `brainpy.analysis.`**`PhasePlane`**(*integrals*, *target_vars*, *fixed_vars=None*, *pars_update=None*, *numerical_resolution=0.1*, *options=None*)

A tool class for phase plane analyzer.

PhasePlaneAnalyzer is used to analyze the phase portrait of 1D or 2D dynamical systems. It can also be used to analyze the phase portrait of high-dimensional system but with the fixation of other variables to preserve only one/two dynamical variables.

Parameters

- **`integrals`** (*`NeuType`*) – The neuron model which defines the differential equations by using *brainpy.integrate*.
- **`target_vars`** (*`dict`*) – The target variables to analyze, with the format of *{'var1': [var_min, var_max], 'var2': [var_min, var_max]}*.
- **`fixed_vars`** (*`dict, optional`*) – The fixed variables, which means the variables will not be updated.
- **`pars_update`** (*`dict, optional`*) – The parameters in the differential equations to update.
- **`numerical_resolution`** (*`float, dict`*) – The variable resolution for numerical iterative solvers. This variable will be useful in the solving of nullcline and fixed points by using the iterative optimization method. It can be a float, which will be used as `numpy.arange(var_min, var_max, resolution)`. Or, it can be a dict, with the format of `{'var1': resolution1, 'var2': resolution2}`. Or, it can be a dict with the format of `{'var1': np.arange(x, x, x), 'var2': np.arange(x, x, x)}`.
- **`options`** (*`dict, optional`*) – The other setting parameters, which includes:

  lim_scale float. The axis limit scale factor. Default is 1.05. The setting means the axes will be clipped to `[var_min * (1-lim_scale)/2, var_max * (var_max-1)/2]`.

  sympy_solver_timeout float, with the unit of second. The maximum time allowed to use sympy solver to get the variable relationship.

  escape_sympy_solver bool. Whether escape to use sympy solver, and directly use numerical optimization method to solve the nullcline and fixed points.

  shgo_args dict. Arguments of *shgo* optimization method, which can be used to set the fields of: constraints, n, iters, callback, minimizer_kwargs, options, sampling_method.

  show_shgo bool. whether print the shgo's value.

  perturbation float. The small perturbation used to solve the function derivative.

  fl_tol float. The tolerance of the function value to recognize it as a condidate of function root point.

  xl_tol float. The tolerance of the l2 norm distances between this point and previous points. If the norm distances are all bigger than *xl_tol* means this point belong to a new function root point.

`__init__`(*integrals*, *target_vars*, *fixed_vars=None*, *pars_update=None*, *numerical_resolution=0.1*, *options=None*)

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

Methods

`__init__`(integrals, target_vars[, . . .])	Initialize self.
`plot_fixed_point`(*args, **kwargs)	Plot fixed points.
`plot_limit_cycle_by_sim`(initials, duration)	Plot limit cycles according to the settings.
`plot_nullcline`(*args, **kwargs)	Plot nullcline (only supported in 2D system).
`plot_trajectory`(initials, duration[, . . .])	Plot trajectories according to the settings.
`plot_vector_field`(*args, **kwargs)	Plot vector filed of a 2D/1D system.

14.1.2 brainpy.analysis.Bifurcation

`class brainpy.analysis.`**`Bifurcation`**(*integrals*, *target_pars*, *target_vars*, *fixed_vars=None*, *pars_update=None*, *numerical_resolution=0.1*, *options=None*)

A tool class for bifurcation analysis.

The bifurcation analyzer is restricted to analyze the bifurcation relation between membrane potential and a given model parameter (co-dimension-1 case) or two model parameters (co-dimension-2 case).

Externally injected current is also treated as a model parameter in this class, instead of a model state.

Parameters

- **`integrals`** (*`function, functions`*) – The integral functions defined with *brainpy.odeint* or *brainpy.sdeint* or *brainpy.ddeint*, or *brainpy.fdeint*.
- **`target_vars`** (*`dict`*) – The target dynamical variables. It must a dictionary which specifies the boundary of the variables: *{'var1': [min, max]}*.
- **`fixed_vars`** (*`dict`*) – The fixed variables. It must a fixed value with the format of *{'var1': value}*.
- **`target_pars`** (*`dict, optional`*) – The parameters which can be dynamical varied. It must be a dictionary which specifies the boundary of the variables: *{'par1': [min, max]}*
- **`pars_update`** (*`dict, optional`*) – The parameters to update. Or, they can be treated as staitic parameters. Same with the *fixed_vars*, they are must fixed values with the format of *{'par1': value}*.
- **`numerical_resolution`** (*`float, dict`*) – The resolution for numerical iterative solvers. Default is 0.1. It can set the numerical resolution of dynamical variables or dynamical parameters. For example, set `numerical_resolution=0.1` will generalize it to all variables and parameters; set `numerical_resolution={var1: 0.1, var2: 0.2, par1: 0.1, par2: 0.05}` will specify the particular resolutions to variables and parameters. Moreover, you can also set `numerical_resolution={var1: np.array([...]), var2: 0.1}` to specify the search points need to explore for variable *var1*. This will be useful to set sense search points at some inflection points.
- **`options`** (*`dict, optional`*) – The other setting parameters, which includes:

  perturbation float. The small perturbation used to solve the function derivatives.

  sympy_solver_timeout float, with the unit of second. The maximum time allowed to use sympy solver to get the variable relationship.

  escape_sympy_solver bool. Whether escape to use sympy solver, and directly use numerical optimization method to solve the nullcline and fixed points.

lim_scale float. The axis limit scale factor. Default is 1.05. The setting means the axes will be clipped to `[var_min * (1-lim_scale)/2, var_max * (var_max-1)/2]`.

The parameters which are usefull for two-dimensional bifurcation analysis:

shgo_args dict. Arguments of *shgo* optimization method, which can be used to set the fields of: constraints, n, iters, callback, minimizer_kwargs, options, sampling_method.

show_shgo bool. whether print the shgo's value.

fl_tol float. The tolerance of the function value to recognize it as a candidate of function root point.

xl_tol float. The tolerance of the l2 norm distances between this point and previous points. If the norm distances are all bigger than *xl_tol* means this point belong to a new function root point.

`__init__`(*integrals*, *target_pars*, *target_vars*, *fixed_vars=None*, *pars_update=None*, *numerical_resolution=0.1*, *options=None*)

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

Methods

`__init__`(integrals, target_pars, target_vars)	Initialize self.
`plot_bifurcation`(*args, **kwargs)	Plot bifurcation, which support bifurcation analysis of co-dimension 1 and co-dimension 2.
`plot_limit_cycle_by_sim`(var[, duration, …])	Plot limit cycles by the simulation results.

14.1.3 brainpy.analysis.FastSlowBifurcation

`class brainpy.analysis.`**`FastSlowBifurcation`**(*integrals*, *fast_vars*, *slow_vars*, *fixed_vars=None*, *pars_update=None*, *numerical_resolution=0.1*, *options=None*)

Fast slow analysis analysis proposed by John Rinzel[1][2][3].

(J Rinzel, 1985, 1986, 1987) proposed that in a fast-slow dynamical system, we can treat the slow variables as the bifurcation parameters, and then study how the different value of slow variables affect the bifurcation of the fast sub-system.

Parameters

- **`integrals`** (*`function, functions`*) – The integral functions defined with *brainpy.odeint* or *brainpy.sdeint* or *brainpy.ddeint*, or *brainpy.fdeint*.
- **`fast_vars`** (*`dict`*) – The fast dynamical variables. It must a dictionary which specifies the boundary of the variables: *{'var1': [min, max]}*.
- **`slow_vars`** (*`dict`*) – The slow dynamical variables. It must a dictionary which specifies the boundary of the variables: *{'var1': [min, max]}*.

[1] Rinzel, John. "Bursting oscillations in an excitable membrane model." In Ordinary and partial differential equations, pp. 304-316. Springer, Berlin, Heidelberg, 1985.

[2] Rinzel, John , and Y. S. Lee . On Different Mechanisms for Membrane Potential Bursting. Nonlinear Oscillations in Biology and Chemistry. Springer Berlin Heidelberg, 1986.

[3] Rinzel, John. "A formal classification of bursting mechanisms in excitable systems." In Mathematical topics in population biology, morphogenesis and neurosciences, pp. 267-281. Springer, Berlin, Heidelberg, 1987.

- **fixed_vars** (*dict*) – The fixed variables. It must a fixed value with the format of *{'var1': value}*.
- **pars_update** (*dict, optional*) – The parameters to update. Or, they can be treated as staitic parameters. Same with the *fixed_vars*, they are must fixed values with the format of *{'par1': value}*.
- **numerical_resolution** (*float, dict*) – The resolution for numerical iterative solvers. Default is 0.1. It can set the numerical resolution of dynamical variables or dynamical parameters. For example, set `numerical_resolution=0.1` will generalize it to all variables and parameters; set `numerical_resolution={var1: 0.1, var2: 0.2, par1: 0.1, par2: 0.05}` will specify the particular resolutions to variables and parameters. Moreover, you can also set `numerical_resolution={var1: np.array([...]), var2: 0.1}` to specify the search points need to explore for variable *var1*. This will be useful to set sense search points at some inflection points.
- **options** (*dict, optional*) – The other setting parameters, which includes:

  perturbation float. The small perturbation used to solve the function derivatives.

  sympy_solver_timeout float, with the unit of second. The maximum time allowed to use sympy solver to get the variable relationship.

  escape_sympy_solver bool. Whether escape to use sympy solver, and directly use numerical optimization method to solve the nullcline and fixed points.

  lim_scale float. The axis limit scale factor. Default is 1.05. The setting means the axes will be clipped to `[var_min * (1-lim_scale)/2, var_max * (var_max-1)/2]`.

References

`__init__`(*integrals*, *fast_vars*, *slow_vars*, *fixed_vars=None*, *pars_update=None*, *numerical_resolution=0.1*, *options=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(integrals, fast_vars, slow_vars[, ...])	Initialize self.
`plot_bifurcation`(*args, **kwargs)	Plot bifurcation.
`plot_limit_cycle_by_sim`(*args, **kwargs)	Plot limit cycles by the simulation results.
`plot_trajectory`(*args, **kwargs)	Plot trajectory.

14.1.4 brainpy.analysis.get_1d_stability_types

`brainpy.analysis.`**`get_1d_stability_types`**`()`

Get the stability types of 1D system.

14.1.5 brainpy.analysis.get_2d_stability_types

`brainpy.analysis.`**`get_2d_stability_types`**`()`

Get the stability types of 2D system.

14.1.6 brainpy.analysis.get_3d_stability_types

`brainpy.analysis.`**`get_3d_stability_types`**`()`

Get the stability types of 3D system.

14.1.7 brainpy.analysis.stability_analysis

`brainpy.analysis.`**`stability_analysis`**`(`*derivatives*`)`

Stability analysis for fixed points.

The analysis is referred to[1].

Parameters **`derivatives`** (*`float, tuple, list, np.ndarray`*) – The derivative of the f.

Returns **fp_type** – The type of the fixed point.

Return type str

References

14.2 Phase Plane Analysis

We provide a fundamental class *PhasePlane* to help users make phase plane analysis for 1D/2D dynamical systems. Five methods are provided, which can help you to plot:

- Fixed points
- Nullcline (zero-growth isoclines)
- Vector filed
- Limit cycles
- Trajectory

`class` `brainpy.analysis.`**`PhasePlane`**`(`*integrals*, *target_vars*, *fixed_vars=None*, *pars_update=None*, *numerical_resolution=0.1*, *options=None*`)`

A tool class for phase plane analyzer.

PhasePlaneAnalyzer is used to analyze the phase portrait of 1D or 2D dynamical systems. It can also be used to analyze the phase portrait of high-dimensional system but with the fixation of other variables to preserve only one/two dynamical variables.

Parameters

[1] http://www.egwald.ca/nonlineardynamics/twodimensionaldynamics.php

- **integrals** (*NeuType*) – The neuron model which defines the differential equations by using *brainpy.integrate*.
- **target_vars** (*dict*) – The target variables to analyze, with the format of *{'var1': [var_min, var_max], 'var2': [var_min, var_max]}*.
- **fixed_vars** (*dict, optional*) – The fixed variables, which means the variables will not be updated.
- **pars_update** (*dict, optional*) – The parameters in the differential equations to update.
- **numerical_resolution** (*float, dict*) – The variable resolution for numerical iterative solvers. This variable will be useful in the solving of nullcline and fixed points by using the iterative optimization method. It can be a float, which will be used as `numpy.arange(var_min, var_max, resolution)`. Or, it can be a dict, with the format of `{'var1': resolution1, 'var2': resolution2}`. Or, it can be a dict with the format of `{'var1': np.arange(x, x, x), 'var2': np.arange(x, x, x)}`.
- **options** (*dict, optional*) – The other setting parameters, which includes:

  lim_scale float. The axis limit scale factor. Default is 1.05. The setting means the axes will be clipped to `[var_min * (1-lim_scale)/2, var_max * (var_max-1)/2]`.

  sympy_solver_timeout float, with the unit of second. The maximum time allowed to use sympy solver to get the variable relationship.

  escape_sympy_solver bool. Whether escape to use sympy solver, and directly use numerical optimization method to solve the nullcline and fixed points.

  shgo_args dict. Arguments of *shgo* optimization method, which can be used to set the fields of: constraints, n, iters, callback, minimizer_kwargs, options, sampling_method.

  show_shgo bool. whether print the shgo's value.

  perturbation float. The small perturbation used to solve the function derivative.

  fl_tol float. The tolerance of the function value to recognize it as a condidate of function root point.

  xl_tol float. The tolerance of the l2 norm distances between this point and previous points. If the norm distances are all bigger than *xl_tol* means this point belong to a new function root point.

plot_fixed_point(**args*, ***kwargs*)
Plot fixed points.

Parameters **show** (*bool*) – Whether show the figure.

plot_limit_cycle_by_sim(*initials*, *duration*, *tol=0.001*, *show=False*)
Plot limit cycles according to the settings.

Parameters

- **initials** (*list, tuple*) – The initial value setting of the targets. It can be a tuple/list of floats to specify each value of dynamical variables (for example, `(a, b)`). It can also be a tuple/list of tuple to specify multiple initial values (for example, `[(a1, b1), (a2, b2)]`).
- **duration** (*int, float, tuple, list*) – The running duration. Same with the `duration` in `NeuGroup.run()`. It can be a int/float (`t_end`) to specify the same running end time, or it can be a tuple/list of int/float (`(t_start, t_end)`) to specify the start and

end simulation time. Or, it can be a list of tuple (`[(t1_start, t1_end), (t2_start, t2_end)]`) to specify the specific start and end simulation time for each initial value.

- **show** (*bool*) – Whether show or not.

plot_nullcline(**args*, ***kwargs*)
Plot nullcline (only supported in 2D system).

Parameters **show** (*bool*) – Whether show the figure.

plot_trajectory(*initials*, *duration*, *plot_duration=None*, *axes='v-v'*, *show=False*)
Plot trajectories according to the settings.

Parameters

- **initials** (*list, tuple, dict*) – The initial value setting of the targets. It can be a tuple/list of floats to specify each value of dynamical variables (for example, `(a, b)`). It can also be a tuple/list of tuple to specify multiple initial values (for example, `[(a1, b1), (a2, b2)]`).
- **duration** (*int, float, tuple, list*) – The running duration. Same with the `duration` in `NeuGroup.run()`. It can be a int/float (`t_end`) to specify the same running end time, or it can be a tuple/list of int/float (`(t_start, t_end)`) to specify the start and end simulation time. Or, it can be a list of tuple (`[(t1_start, t1_end), (t2_start, t2_end)]`) to specify the specific start and end simulation time for each initial value.
- **plot_duration** (*tuple, list, optional*) – The duration to plot. It can be a tuple with `(start, end)`. It can also be a list of tuple `[(start1, end1), (start2, end2)]` to specify the plot duration for each initial value running.
- **axes** (*str*) – The axes to plot. It can be:
  - **'v-v'** Plot the trajectory in the 'x_var'-'y_var' axis.
  - **'t-v'** Plot the trajectory in the 'time'-'var' axis.
- **show** (*bool*) – Whether show or not.

plot_vector_field(**args*, ***kwargs*)
Plot vector filed of a 2D/1D system.

Parameters **show** (*bool*) – Whether show the figure.

14.3 Bifurcation Analysis

We also provide basic bifurcation analysis for 1D/2D dynamical systems.

class brainpy.analysis.**Bifurcation**(*integrals*, *target_pars*, *target_vars*, *fixed_vars=None*, *pars_update=None*, *numerical_resolution=0.1*, *options=None*)
A tool class for bifurcation analysis.

The bifurcation analyzer is restricted to analyze the bifurcation relation between membrane potential and a given model parameter (co-dimension-1 case) or two model parameters (co-dimension-2 case).

Externally injected current is also treated as a model parameter in this class, instead of a model state.

Parameters

- **integrals** (*function, functions*) – The integral functions defined with *brainpy.odeint* or *brainpy.sdeint* or *brainpy.ddeint*, or *brainpy.fdeint*.

- `target_vars` (*dict*) – The target dynamical variables. It must a dictionary which specifies the boundary of the variables: *{'var1': [min, max]}*.
- `fixed_vars` (*dict*) – The fixed variables. It must a fixed value with the format of *{'var1': value}*.
- `target_pars` (*dict, optional*) – The parameters which can be dynamical varied. It must be a dictionary which specifies the boundary of the variables: *{'par1': [min, max]}*
- `pars_update` (*dict, optional*) – The parameters to update. Or, they can be treated as staitic parameters. Same with the *fixed_vars*, they are must fixed values with the format of *{'par1': value}*.
- `numerical_resolution` (*float, dict*) – The resolution for numerical iterative solvers. Default is 0.1. It can set the numerical resolution of dynamical variables or dynamical parameters. For example, set `numerical_resolution=0.1` will generalize it to all variables and parameters; set `numerical_resolution={var1: 0.1, var2: 0.2, par1: 0.1, par2: 0.05}` will specify the particular resolutions to variables and parameters. Moreover, you can also set `numerical_resolution={var1: np.array([...]), var2: 0.1}` to specify the search points need to explore for variable *var1*. This will be useful to set sense search points at some inflection points.
- `options` (*dict, optional*) – The other setting parameters, which includes:

  perturbation float. The small perturbation used to solve the function derivatives.

  sympy_solver_timeout float, with the unit of second. The maximum time allowed to use sympy solver to get the variable relationship.

  escape_sympy_solver bool. Whether escape to use sympy solver, and directly use numerical optimization method to solve the nullcline and fixed points.

  lim_scale float. The axis limit scale factor. Default is 1.05. The setting means the axes will be clipped to `[var_min * (1-lim_scale)/2, var_max * (var_max-1)/2]`.

  The parameters which are usefull for two-dimensional bifurcation analysis:

  shgo_args dict. Arguments of *shgo* optimization method, which can be used to set the fields of: constraints, n, iters, callback, minimizer_kwargs, options, sampling_method.

  show_shgo bool. whether print the shgo's value.

  fl_tol float. The tolerance of the function value to recognize it as a candidate of function root point.

  xl_tol float. The tolerance of the l2 norm distances between this point and previous points. If the norm distances are all bigger than *xl_tol* means this point belong to a new function root point.

`plot_bifurcation`(**args*, ***kwargs*)

Plot bifurcation, which support bifurcation analysis of co-dimension 1 and co-dimension 2.

Parameters `show` (*bool*) – Whether show the bifurcation figure.

Returns **points** – The bifurcation points which specifies their fixed points and corresponding stability.

Return type dict

plot_limit_cycle_by_sim(*var*, *duration=100*, *inputs=()*, *plot_style=None*, *tol=0.001*, *show=False*)
Plot limit cycles by the simulation results.

This function help users plot the limit cycles through the simulation results, in which the periodic signals will be automatically found and then treated them as the candidate of limit cycles.

Parameters

- **var** (*str*) – The target variable to found its limit cycles.
- **duration** (*int, float, tuple, list*) – The simulation duration.
- **inputs** (*tuple, list*) – The simulation inputs.
- **plot_style** (*dict*) – The limit cycle plotting style settings.
- **tol** (*float*) – The tolerance to found periodic signals.
- **show** (*bool*) – Whether show the figure.

14.4 Fast-slow System Analysis

For some 3D dynamical system, which can be treated as a fast-slow system, they can be easily analyzed through our provided *FastSlowBifurcation*.

class brainpy.analysis.**FastSlowBifurcation**(*integrals*, *fast_vars*, *slow_vars*, *fixed_vars=None*, *pars_update=None*, *numerical_resolution=0.1*, *options=None*)
Fast slow analysis analysis proposed by John Rinzel[Page 122, 1??].

(J Rinzel, 1985, 1986, 1987) proposed that in a fast-slow dynamical system, we can treat the slow variables as the bifurcation parameters, and then study how the different value of slow variables affect the bifurcation of the fast sub-system.

Parameters

- **integrals** (*function, functions*) – The integral functions defined with *brainpy.odeint* or *brainpy.sdeint* or *brainpy.ddeint*, or *brainpy.fdeint*.
- **fast_vars** (*dict*) – The fast dynamical variables. It must a dictionary which specifies the boundary of the variables: *{'var1': [min, max]}*.
- **slow_vars** (*dict*) – The slow dynamical variables. It must a dictionary which specifies the boundary of the variables: *{'var1': [min, max]}*.
- **fixed_vars** (*dict*) – The fixed variables. It must a fixed value with the format of *{'var1': value}*.
- **pars_update** (*dict, optional*) – The parameters to update. Or, they can be treated as staitic parameters. Same with the *fixed_vars*, they are must fixed values with the format of *{'par1': value}*.
- **numerical_resolution** (*float, dict*) – The resolution for numerical iterative solvers. Default is 0.1. It can set the numerical resolution of dynamical variables or dynamical parameters. For example, set `numerical_resolution=0.1` will generalize it to all variables and parameters; set `numerical_resolution={var1: 0.1, var2: 0.2, par1: 0.1, par2: 0.05}` will specify the particular resolutions to variables and parameters. Moreover, you can also set `numerical_resolution={var1: np.array([...]), var2: 0.1}` to specify the search points need to explore for variable *var1*. This will be useful to set sense search points at some inflection points.

- **options** (*dict, optional*) – The other setting parameters, which includes:

  perturbation float. The small perturbation used to solve the function derivatives.

  sympy_solver_timeout float, with the unit of second. The maximum time allowed to use sympy solver to get the variable relationship.

  escape_sympy_solver bool. Whether escape to use sympy solver, and directly use numerical optimization method to solve the nullcline and fixed points.

  lim_scale float. The axis limit scale factor. Default is 1.05. The setting means the axes will be clipped to `[var_min * (1-lim_scale)/2, var_max * (var_max-1)/2]`.

References

plot_bifurcation(**args*, ***kwargs*)
Plot bifurcation.

Parameters **show** (*bool*) – Whether show the bifurcation figure.

Returns **points** – The bifurcation points which specifies their fixed points and corresponding stability.

Return type dict

14.5 Useful Functions

In *brainpy.analysis* module, we also provide several useful functions which may help your dynamical system analysis.

```
>>> get_1d_stability_types()
['saddle node', 'stable point', 'unstable point']
```

```
>>> get_2d_stability_types()
['saddle node',
 'center',
 'stable node',
 'stable focus',
 'stable star',
 'center manifold',
 'unstable node',
 'unstable focus',
 'unstable star',
 'unstable line',
 'stable degenerate',
 'unstable degenerate']
```

CHAPTER

FIFTEEN

BRAINPY.INTEGRATORS MODULE

15.1 Numerical Methods for ODEs

Numerical methods for ordinary differential equations.

15.1.1 Explicit Runge-Kutta methods

euler([f, show_code, dt, var_type])	The Euler method is first order.
midpoint([f, show_code, dt, var_type])	midpoint method for ordinary differential equations.
heun2([f, show_code, dt, var_type])	Heun's method for ordinary differential equations.
ralston2([f, show_code, dt, var_type])	Ralston's method for ordinary differential equations.
rk2([f, show_code, dt, beta, var_type])	Runge–Kutta methods for ordinary differential equations.
rk3([f, show_code, dt, var_type])	Classical third-order Runge-Kutta method for ordinary differential equations.
heun3([f, show_code, dt, var_type])	Heun's third-order method for ordinary differential equations.
ralston3([f, show_code, dt, var_type])	Ralston's third-order method for ordinary differential equations.
ssprk3([f, show_code, dt, var_type])	Third-order Strong Stability Preserving Runge-Kutta (SSPRK3).
rk4([f, show_code, dt, var_type])	Classical fourth-order Runge-Kutta method for ordinary differential equations.
ralston4([f, show_code, dt, var_type])	Ralston's fourth-order method for ordinary differential equations.
rk4_38rule([f, show_code, dt, var_type])	3/8-rule fourth-order method for ordinary differential equations.

brainpy.integrators.ode.euler

`brainpy.integrators.ode.euler`(*f=None*, *show_code=None*, *dt=None*, *var_type=None*)

The Euler method is first order. The lack of stability and accuracy limits its popularity mainly to use as a simple introductory example of a numeric solution method.

brainpy.integrators.ode.midpoint

`brainpy.integrators.ode.midpoint`(*f=None*, *show_code=None*, *dt=None*, *var_type=None*)

midpoint method for ordinary differential equations.

The (explicit) midpoint method is a second-order method with two stages.

It has the characteristics of:

- method stage = 2
- method order = 2
- Butcher Tables:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1/2 & 1/2 & 0 \\ \hline & 0 & 1 \end{array}$$

brainpy.integrators.ode.heun2

`brainpy.integrators.ode.heun2`(*f=None*, *show_code=None*, *dt=None*, *var_type=None*)

Heun's method for ordinary differential equations.

Heun's method is a second-order method with two stages. It is also known as the explicit trapezoid rule, improved Euler's method, or modified Euler's method.

It has the characteristics of:

- method stage = 2
- method order = 2
- Butcher Tables:

$$\begin{array}{c|cc} 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 0.0 \\ \hline & 0.5 & 0.5 \end{array}$$

brainpy.integrators.ode.ralston2

`brainpy.integrators.ode.ralston2`(*f=None*, *show_code=None*, *dt=None*, *var_type=None*)

Ralston's method for ordinary differential equations.

Ralston's method is a second-order method with two stages and a minimum local error bound.

It has the characteristics of:

- method stage = 2
- method order = 2
- Butcher Tables:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 2/3 & 2/3 & 0 \\ \hline & 1/4 & 3/4 \end{array}$$

brainpy.integrators.ode.rk2

`brainpy.integrators.ode.rk2`(*f=None*, *show_code=None*, *dt=None*, *beta=None*, *var_type=None*)

Runge–Kutta methods for ordinary differential equations.

Generic second-order method.

It has the characteristics of:

- method stage = 2
- method order = 2
- Butcher Tables:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \beta & \beta & 0 \\ \hline & 1 - \frac{1}{2*\beta} & \frac{1}{2*\beta} \end{array}$$

brainpy.integrators.ode.rk3

`brainpy.integrators.ode.rk3`(*f=None*, *show_code=None*, *dt=None*, *var_type=None*)

Classical third-order Runge-Kutta method for ordinary differential equations.

It has the characteristics of:

- method stage = 3
- method order = 3
- Butcher Tables:

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \\ 1 & -1 & 2 & 0 \\ \hline & 1/6 & 2/3 & 1/6 \end{array}$$

brainpy.integrators.ode.heun3

`brainpy.integrators.ode.heun3`(*f=None*, *show_code=None*, *dt=None*, *var_type=None*)

Heun's third-order method for ordinary differential equations.

It has the characteristics of:

- method stage = 3
- method order = 3
- Butcher Tables:

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1/3 & 1/3 & 0 & 0 \\ 2/3 & 0 & 2/3 & 0 \\ \hline & 1/4 & 0 & 3/4 \end{array}$$

brainpy.integrators.ode.ralston3

`brainpy.integrators.ode.`**`ralston3`**(*f=None*, *show_code=None*, *dt=None*, *var_type=None*)

Ralston’s third-order method for ordinary differential equations.

It has the characteristics of:

- method stage = 3
- method order = 3
- Butcher Tables:

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \\ 3/4 & 0 & 3/4 & 0 \\ \hline & 2/9 & 1/3 & 4/9 \end{array}$$

References

brainpy.integrators.ode.ssprk3

`brainpy.integrators.ode.`**`ssprk3`**(*f=None*, *show_code=None*, *dt=None*, *var_type=None*)

Third-order Strong Stability Preserving Runge-Kutta (SSPRK3).

It has the characteristics of:

- method stage = 3
- method order = 3
- Butcher Tables:

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1/2 & 1/4 & 1/4 & 0 \\ \hline & 1/6 & 1/6 & 2/3 \end{array}$$

brainpy.integrators.ode.rk4

`brainpy.integrators.ode.`**`rk4`**(*f=None*, *show_code=None*, *dt=None*, *var_type=None*)

Classical fourth-order Runge-Kutta method for ordinary differential equations.

It has the characteristics of:

- method stage = 4
- method order = 4
- Butcher Tables:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

brainpy.integrators.ode.ralston4

`brainpy.integrators.ode.ralston4`(*f=None*, *show_code=None*, *dt=None*, *var_type=None*)

Ralston's fourth-order method for ordinary differential equations.

It has the characteristics of:

- method stage = 4
- method order = 4
- Butcher Tables:

$$\begin{array}{c|cccc}
0 & 0 & 0 & 0 & 0 \\
.4 & .4 & 0 & 0 & 0 \\
.45573725 & .29697761 & .15875964 & 0 & 0 \\
1 & .21810040 & -3.05096516 & 3.83286476 & 0 \\
\hline
 & .17476028 & -.55148066 & 1.20553560 & .17118478
\end{array}$$

References

[1] **Ralston, Anthony (1962). "Runge-Kutta Methods with Minimum Error Bounds".** Math. Comput. 16 (80): 431–437. doi:10.1090/S0025-5718-1962-0150954-0

brainpy.integrators.ode.rk4_38rule

`brainpy.integrators.ode.rk4_38rule`(*f=None*, *show_code=None*, *dt=None*, *var_type=None*)

3/8-rule fourth-order method for ordinary differential equations.

This method doesn't have as much notoriety as the "classical" method, but is just as classical because it was proposed in the same paper (Kutta, 1901).

It has the characteristics of:

- method stage = 4
- method order = 4
- Butcher Tables:

$$\begin{array}{c|cccc}
0 & 0 & 0 & 0 & 0 \\
1/3 & 1/3 & 0 & 0 & 0 \\
2/3 & -1/3 & 1 & 0 & 0 \\
1 & 1 & -1 & 1 & 0 \\
\hline
 & 1/8 & 3/8 & 3/8 & 1/8
\end{array}$$

15.1.2 Adaptive Runge-Kutta methods

rkf45([f, tol, adaptive, dt, show_code, ...])	The Runge–Kutta–Fehlberg method for ordinary differential equations.
rkf12([f, tol, adaptive, dt, show_code, ...])	The Fehlberg RK1(2) method for ordinary differential equations.
rkdp([f, tol, adaptive, dt, show_code, var_type])	The Dormand–Prince method for ordinary differential equations.

continues on next page

Table 2 – continued from previous page

ck([f, tol, adaptive, dt, show_code, var_type])	The Cash–Karp method for ordinary differential equations.
bs([f, tol, adaptive, dt, show_code, var_type])	The Bogacki–Shampine method for ordinary differential equations.
heun_euler([f, tol, adaptive, dt, ...])	The Heun–Euler method for ordinary differential equations.

brainpy.integrators.ode.rkf45

`brainpy.integrators.ode.`**`rkf45`**(*f=None*, *tol=None*, *adaptive=None*, *dt=None*, *show_code=None*, *var_type=None*)

The Runge–Kutta–Fehlberg method for ordinary differential equations.

The method presented in Fehlberg's 1969 paper has been dubbed the RKF45 method, and is a method of order $O(h^4)$ with an error estimator of order $O(h^5)$. The novelty of Fehlberg's method is that it is an embedded method from the Runge–Kutta family, meaning that identical function evaluations are used in conjunction with each other to create methods of varying order and similar error constants.

It has the characteristics of:

- method stage = 6
- method order = 5
- Butcher Tables:

$$
\begin{array}{l|lllll}
0 & & & & & \\
 & & & & & \\
1/4 & 1/4 & & & & \\
3/8 & 3/32 & 9/32 & & & \\
12/13 & 1932/2197 & -7200/2197 & 7296/2197 & & \\
1 & 439/216 & -8 & 3680/513 & -845/4104 & \\
 & & & & & \\
1/2 & -8/27 & 2 & -3544/2565 & 1859/4104 & -11/40 \\
\hline
 & 16/135 & 0 & 6656/12825 & 28561/56430 & -9/50 \\
2/55 & & & & & \\
 & 25/216 & 0 & 1408/2565 & 2197/4104 & -1/5 \\
0 & & & & &
\end{array}
$$

References

[1] https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta%E2%80%93Fehlberg_method [2] Erwin Fehlberg (1969). Low-order classical Runge-Kutta formulas with step

size control and their application to some heat transfer problems . NASA Technical Report 315. https://ntrs.nasa.gov/api/citations/19690021375/downloads/19690021375.pdf

brainpy.integrators.ode.rkf12

`brainpy.integrators.ode.rkf12`(*f=None*, *tol=None*, *adaptive=None*, *dt=None*, *show_code=None*, *var_type=None*)

The Fehlberg RK1(2) method for ordinary differential equations.

The Fehlberg method has two methods of orders 1 and 2.

It has the characteristics of:

- method stage = 2
- method order = 1
- Butcher Tables:

$$
\begin{array}{l|ll}
0 & & \\
1/2 & 1/2 & \\
1 & 1/256 & 255/256 \\
\hline
 & 1/512 & 255/256 \\
1/512 & & \\
 & 1/256 & 255/256 \\
0 & &
\end{array}
$$

References

brainpy.integrators.ode.rkdp

`brainpy.integrators.ode.rkdp`(*f=None*, *tol=None*, *adaptive=None*, *dt=None*, *show_code=None*, *var_type=None*)

The Dormand–Prince method for ordinary differential equations.

The DOPRI method, is an explicit method for solving ordinary differential equations (Dormand & Prince 1980). The Dormand–Prince method has seven stages, but it uses only six function evaluations per step because it has the FSAL (First Same As Last) property: the last stage is evaluated at the same point as the first stage of the next step. Dormand and Prince chose the coefficients of their method to minimize the error of the fifth-order solution. This is the main difference with the Fehlberg method, which was constructed so that the fourth-order solution has a small error. For this reason, the Dormand–Prince method is more suitable when the higher-order solution is used to continue the integration, a practice known as local extrapolation (Shampine 1986; Hairer, Nørsett & Wanner 2008, pp. 178–179).

It has the characteristics of:

- method stage = 7
- method order = 5
- Butcher Tables:

0						
1/5	1/5					
3/10	3/40	9/40				
4/5	44/45	−56/15	32/9			
8/9	19372/6561	−25360/2187	64448/6561	−212/729		
1	9017/3168	−355/33	46732/5247	49/176	−5103/18656	
1	35/384	0	500/1113	125/192	−2187/6784	11/84
	35/384	0	500/1113	125/192	−2187/6784	11/84
0						
	5179/57600	0	7571/16695	393/640	−92097/339200	187/2100
1/40						

References

[1] https://en.wikipedia.org/wiki/Dormand%E2%80%93Prince_method [2] Dormand, J. R.; Prince, P. J. (1980), “A family of embedded Runge-Kutta formulae”,

Journal of Computational and Applied Mathematics, 6 (1): 19–26, doi:10.1016/0771-050X(80)90013-3.

brainpy.integrators.ode.ck

`brainpy.integrators.ode.ck`(*f=None*, *tol=None*, *adaptive=None*, *dt=None*, *show_code=None*, *var_type=None*)

The Cash–Karp method for ordinary differential equations.

The Cash–Karp method was proposed by Professor Jeff R. Cash from Imperial College London and Alan H. Karp from IBM Scientific Center. it uses six function evaluations to calculate fourth- and fifth-order accurate solutions. The difference between these solutions is then taken to be the error of the (fourth order) solution. This error estimate is very convenient for adaptive stepsize integration algorithms.

It has the characteristics of:

- method stage = 6
- method order = 4
- Butcher Tables:

0					
1/5	1/5				
3/10	3/40	9/40			
3/5	3/10	−9/10	6/5		
1	−11/54	5/2	−70/27	35/27	
7/8	1631/55296	175/512	575/13824	44275/110592	253/4096
	37/378	0	250/621	125/594	0
512/1771					
	2825/27648	0	18575/48384	13525/55296	277/14336
1/4					

References

[1] https://en.wikipedia.org/wiki/Cash%E2%80%93Karp_method [2] J. R. Cash, A. H. Karp. "A variable order Runge-Kutta method for initial value

problems with rapidly varying right-hand sides", ACM Transactions on Mathematical Software 16: 201-222, 1990. doi:10.1145/79505.79507

brainpy.integrators.ode.bs

`brainpy.integrators.ode.bs`(*f=None*, *tol=None*, *adaptive=None*, *dt=None*, *show_code=None*, *var_type=None*)

The Bogacki–Shampine method for ordinary differential equations.

The Bogacki–Shampine method was proposed by Przemysław Bogacki and Lawrence F. Shampine in 1989 (Bogacki & Shampine 1989). The Bogacki–Shampine method is a Runge–Kutta method of order three with four stages with the First Same As Last (FSAL) property, so that it uses approximately three function evaluations per step. It has an embedded second-order method which can be used to implement adaptive step size.

It has the characteristics of:

- method stage = 4
- method order = 3
- Butcher Tables:

0			
$1/2$	$1/2$		
$3/4$	0	$3/4$	
1	$2/9$	$1/3$	$4/9$
	$2/9$	$1/3$	$4/90$
	$7/24$	$1/4$	$1/3$
$1/8$			

References

[1] https://en.wikipedia.org/wiki/Bogacki%E2%80%93Shampine_method [2] Bogacki, Przemysław; Shampine, Lawrence F. (1989), "A 3(2) pair of Runge–Kutta

formulas", Applied Mathematics Letters, 2 (4): 321–325, doi:10.1016/0893-9659(89)90079-7

brainpy.integrators.ode.heun_euler

`brainpy.integrators.ode.heun_euler`(*f=None*, *tol=None*, *adaptive=None*, *dt=None*, *show_code=None*, *var_type=None*)

The Heun–Euler method for ordinary differential equations.

The simplest adaptive Runge–Kutta method involves combining Heun's method, which is order 2, with the Euler method, which is order 1.

It has the characteristics of:

- method stage = 2
- method order = 1
- Butcher Tables:

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \\ & 1 & 0 \end{array}$$

15.1.3 Other methods

`exponential_euler`([f, show_code, dt, var_type])	First order, explicit exponential Euler method.

brainpy.integrators.ode.exponential_euler

`brainpy.integrators.ode.`**`exponential_euler`**(*f=None*, *show_code=None*, *dt=None*, *var_type=None*)

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

$$y_{n+1} = y_n e^{hA} - B/A(1 - e^{hA})$$

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

Return type callable

15.2 Numerical Methods for SDEs

Numerical methods for stochastic differential equations.

`euler`([f, g, dt, sde_type, var_type, …])	
`heun`([f, g, dt, sde_type, var_type, …])	
`milstein`([f, g, dt, sde_type, var_type, …])	
`exponential_euler`([f, g, dt, sde_type, …])	First order, explicit exponential Euler method.
`srk1w1_scalar`([f, g, dt, sde_type, …])	Order 2.0 weak SRK methods for SDEs with scalar Wiener process.
`srk2w1_scalar`([f, g, dt, sde_type, …])	Order 1.5 Strong SRK Methods for SDEs witdt Scalar Noise.
`KlPl_scalar`([f, g, dt, sde_type, var_type, …])	Order 1.0 Strong SRK Methods for SDEs with Scalar Noise.

15.2.1 brainpy.integrators.sde.euler

`brainpy.integrators.sde.`**`euler`**(*f=None*, *g=None*, *dt=None*, *sde_type=None*, *var_type=None*, *wiener_type=None*, *show_code=None*)

15.2.2 brainpy.integrators.sde.heun

`brainpy.integrators.sde.`**`heun`**(*f=None*, *g=None*, *dt=None*, *sde_type=None*, *var_type=None*, *wiener_type=None*, *show_code=None*)

15.2.3 brainpy.integrators.sde.milstein

`brainpy.integrators.sde.`**`milstein`**(*f=None*, *g=None*, *dt=None*, *sde_type=None*, *var_type=None*, *wiener_type=None*, *show_code=None*)

15.2.4 brainpy.integrators.sde.exponential_euler

`brainpy.integrators.sde.`**`exponential_euler`**(*f=None*, *g=None*, *dt=None*, *sde_type=None*, *var_type=None*, *wiener_type=None*, *show_code=None*)

First order, explicit exponential Euler method.

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[1]

$$\begin{aligned} y_{n+1} &= e^{\Delta tA}(y_n + g(y_n)\Delta W_n) + \varphi(\Delta tA)F(y_n)\Delta t \\ &= y_n + \Delta t\varphi(\Delta tA)f(y) + e^{\Delta tA}g(y_n)\Delta W_n \end{aligned}$$

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

15.2.5 brainpy.integrators.sde.srk1w1_scalar

`brainpy.integrators.sde.`**`srk1w1_scalar`**(*f=None*, *g=None*, *dt=None*, *sde_type=None*, *var_type=None*, *wiener_type=None*, *show_code=None*)

Order 2.0 weak SRK methods for SDEs with scalar Wiener process.

This method has have strong orders $(p_d, p_s) = (2.0, 1.5)$.

[1] 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).

The Butcher table is:

$$
\begin{array}{c|cccc|cccc|cccc}
0 & & & & & & & & & & & & \\
 & & & & & & & & & & & & \\
3/4 & 3/4 & & & & 3/2 & & & & & & & \\
0 & 0 & 0 & 0 & & 0 & 0 & 0 & & & & & \\
\hline
0 & & & & & & & & & & & & \\
1/4 & 1/4 & & & & 1/2 & & & & & & & \\
1 & 1 & 0 & & & -1 & 0 & & & & & & \\
1/4 & 0 & 0 & 1/4 & & -5 & 3 & 1/2 & & & & & \\
\hline
 & 1/3 & 2/3 & 0 & 0 & -1 & 4/3 & 2/3 & 0 & -1 & 4/3 & -1/3 & 0 \\
\hline
 & & & & & 2 & -4/3 & -2/3 & 0 & -2 & 5/3 & -2/3 & 1
\end{array}
$$

References

15.2.6 brainpy.integrators.sde.srk2w1_scalar

`brainpy.integrators.sde.`**`srk2w1_scalar`**(*f=None*, *g=None*, *dt=None*, *sde_type=None*, *var_type=None*, *wiener_type=None*, *show_code=None*)

Order 1.5 Strong SRK Methods for SDEs witdt Scalar Noise.

This method has have strong orders $(p_d, p_s) = (3.0, 1.5)$.

The Butcher table is:

$$
\begin{array}{c|cccc|cccc|ccc|}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & \\
1/2 & 1/4 & 1/4 & 0 & 0 & 1 & 1/2 & 0 & 0 & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & \\
1/4 & 1/4 & 0 & 0 & 0 & -1/2 & 0 & 0 & 0 & & & \\
1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & & & \\
1/4 & 0 & 0 & 1/4 & 0 & 2 & -1 & 1/2 & 0 & & & \\
\hline
0 & 1/6 & 1/6 & 2/3 & 0 & -1 & 4/3 & 2/3 & 0 & -1 & -4/3 & 1/3 \\
\hline
1 & & & & & 2 & -4/3 & -2/3 & 0 & -2 & 5/3 & -2/3
\end{array}
$$

References

[1] **Rößler, Andreas. "Strong and weak approximation methods for stochastic differential** equations—some recent developments." Recent developments in applied probability and statistics. Physica-Verlag HD, 2010. 127-153.

[2] **Rößler, Andreas. "Runge–Kutta methods for the strong approximation of solutions of** stochastic differential equations." SIAM Journal on Numerical Analysis 48.3 (2010): 922-952.

15.2.7 brainpy.integrators.sde.KlPl_scalar

`brainpy.integrators.sde.`**`KlPl_scalar`**(*f=None*, *g=None*, *dt=None*, *sde_type=None*, *var_type=None*, *wiener_type=None*, *show_code=None*)

Order 1.0 Strong SRK Methods for SDEs with Scalar Noise.

This method has have orders $p_s = 1.0$.

The Butcher table is:

$$\begin{array}{c|cc|cc|cc|}
0 & 0 & 0 & 0 & 0 & & \\
0 & 0 & 0 & 0 & 0 & & \\
\hline
0 & 0 & 0 & 0 & 0 & & \\
0 & 1 & 0 & 1 & 0 & & \\
\hline
0 & 1 & 0 & 1 & 0 & -1 & 1 \\
\hline
 & & & 1 & 0 & 0 & 0
\end{array}$$

References

[1] **P. E. Kloeden, E. Platen, Numerical Solution of Stochastic Differential** Equations, 2nd Edition, Springer, Berlin Heidelberg New York, 1995.

15.3 General functions

`odeint`([f, method])	Numerical integration for ODE.
`sdeint`([f, method])	
`set_default_odeint`(method)	Set the default ODE numerical integrator method for differential equations.
`get_default_odeint`()	Get the default ODE numerical integrator method.
`set_default_sdeint`(method)	Set the default SDE numerical integrator method for differential equations.
`get_default_sdeint`()	Get the default ODE numerical integrator method.

15.3.1 brainpy.integrators.odeint

`brainpy.integrators.odeint`(*f=None*, *method=None*, ***kwargs*)

Numerical integration for ODE.

Parameters

- **f** (*callable*) –
- **method** (*str*) –
- **kwargs** –

Returns **int_f** – The numerical solver of *f*.

Return type callable

15.3.2 brainpy.integrators.sdeint

`brainpy.integrators.sdeint`(*f=None*, *method=None*, ***kwargs*)

15.3.3 brainpy.integrators.set_default_odeint

`brainpy.integrators.set_default_odeint`(*method*)

Set the default ODE numerical integrator method for differential equations.

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

15.3.4 brainpy.integrators.get_default_odeint

`brainpy.integrators.get_default_odeint()`

Get the default ODE numerical integrator method.

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

Return type str

15.3.5 brainpy.integrators.set_default_sdeint

`brainpy.integrators.set_default_sdeint`(*method*)

Set the default SDE numerical integrator method for differential equations.

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

15.3.6 brainpy.integrators.get_default_sdeint

`brainpy.integrators.get_default_sdeint()`

Get the default ODE numerical integrator method.

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

Return type str

Numerical Methods for ODEs

ode.euler	The Euler method is first order.
ode.midpoint	midpoint method for ordinary differential equations.
ode.heun2	Heun's method for ordinary differential equations.
ode.ralston2	Ralston's method for ordinary differential equations.
ode.rk2	Runge–Kutta methods for ordinary differential equations.
ode.rk3	Classical third-order Runge-Kutta method for ordinary differential equations.
ode.heun3	Heun's third-order method for ordinary differential equations.
ode.ralston3	Ralston's third-order method for ordinary differential equations.
ode.ssprk3	Third-order Strong Stability Preserving Runge-Kutta (SSPRK3).
ode.rk4	Classical fourth-order Runge-Kutta method for ordinary differential equations.
ode.ralston4	Ralston's fourth-order method for ordinary differential equations.
ode.rk4_38rule	3/8-rule fourth-order method for ordinary differential equations.
ode.rkf45	The Runge–Kutta–Fehlberg method for ordinary differential equations.
ode.rkf12	The Fehlberg RK1(2) method for ordinary differential equations.
ode.rkdp	The Dormand–Prince method for ordinary differential equations.
ode.ck	The Cash–Karp method for ordinary differential equations.
ode.bs	The Bogacki–Shampine method for ordinary differential equations.
ode.heun_euler	The Heun–Euler method for ordinary differential equations.
ode.exponential_euler	First order, explicit exponential Euler method.

Numerical Methods for SDEs

sde.euler	
sde.heun	
sde.milstein	
sde.exponential_euler	First order, explicit exponential Euler method.
sde.srk1w1_scalar	Order 2.0 weak SRK methods for SDEs with scalar Wiener process.
sde.srk2w1_scalar	Order 1.5 Strong SRK Methods for SDEs witdt Scalar Noise.

continues on next page

Table 7 – continued from previous page

sde.KlPl_scalar	Order 1.0 Strong SRK Methods for SDEs with Scalar Noise.

CHAPTER

SIXTEEN

BRAINPY.CONNECT MODULE

16.1 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`(i, j[, num_pre])	Get post slicing connections by pre-synaptic ids.
`post_slice`(i, j[, num_post])	Get pre slicing connections by post-synaptic ids.

16.1.1 brainpy.simulation.connectivity.ij2mat

`brainpy.simulation.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

16.1.2 brainpy.simulation.connectivity.mat2ij

`brainpy.simulation.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

16.1.3 brainpy.simulation.connectivity.pre2post

`brainpy.simulation.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

16.1.4 brainpy.simulation.connectivity.post2pre

`brainpy.simulation.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

16.1.5 brainpy.simulation.connectivity.pre2syn

`brainpy.simulation.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

16.1.6 brainpy.simulation.connectivity.post2syn

brainpy.simulation.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

16.1.7 brainpy.simulation.connectivity.pre_slice

brainpy.simulation.connectivity.**pre_slice**(*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

16.1.8 brainpy.simulation.connectivity.post_slice

brainpy.simulation.connectivity.**post_slice**(*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

16.2 Regular Connections

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.
GridN([N, include_self])	The nearest (2*N+1) * (2*N+1) neighbors conn method.

16.2.1 brainpy.simulation.connectivity.One2One

class `brainpy.simulation.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`()	
`make_pre2post`()	
`make_pre2syn`()	
`make_pre_slice`()	
`requires`(*syn_requires)	

16.2.2 brainpy.simulation.connectivity.All2All

class `brainpy.simulation.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`()	

continues on next page

Table 4 – continued from previous page

make_post2syn()	
make_post_slice()	
make_pre2post()	
make_pre2syn()	
make_pre_slice()	
requires(*syn_requires)	

16.2.3 brainpy.simulation.connectivity.GridFour

class brainpy.simulation.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()	
make_pre2post()	
make_pre2syn()	
make_pre_slice()	
requires(*syn_requires)	

16.2.4 brainpy.simulation.connectivity.GridEight

class brainpy.simulation.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`()	
`make_pre2post`()	
`make_pre2syn`()	
`make_pre_slice`()	
`requires`(*syn_requires)	

16.2.5 brainpy.simulation.connectivity.GridN

class brainpy.simulation.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`()	
`make_pre2post`()	
`make_pre2syn`()	
`make_pre_slice`()	
`requires`(*syn_requires)	

class brainpy.simulation.connectivity.**One2One**

Connect two neuron groups one by one. This means The two neuron groups should have the same size.

class brainpy.simulation.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.simulation.connectivity.**GridFour**(*include_self=False*)

The nearest four neighbors conn method.

class brainpy.simulation.connectivity.**GridEight**(*include_self=False*)

The nearest eight neighbors conn method.

class brainpy.simulation.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 ?

16.3 Random Connections

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, method])	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(num_neighbor, prob[, directed, . . .])	Build a Watts–Strogatz small-world graph.
ScaleFreeBA(m[, directed, seed])	Build a random graph according to the Barabási–Albert preferential attachment model.
ScaleFreeBADual(m1, m2, p[, directed, seed])	Build a random graph according to the dual Barabási–Albert preferential attachment model.
PowerLaw(m, p[, directed, seed])	Holme and Kim algorithm for growing graphs with powerlaw degree distribution and approximate average clustering.

16.3.1 brainpy.simulation.connectivity.FixedPostNum

class brainpy.simulation.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()	

continues on next page

Table 9 – continued from previous page

make_post2syn()	
make_post_slice()	
make_pre2post()	
make_pre2syn()	
make_pre_slice()	
requires(*syn_requires)	

16.3.2 brainpy.simulation.connectivity.FixedPreNum

`class brainpy.simulation.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()	
make_pre2post()	
make_pre2syn()	
make_pre_slice()	
requires(*syn_requires)	

16.3.3 brainpy.simulation.connectivity.FixedProb

class brainpy.simulation.connectivity.**FixedProb**(*prob*, *include_self=True*, *seed=None*, *method='matrix'*)

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*, *method='matrix'*)

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

Methods

__init__(prob[, include_self, seed, method])	Initialize self.
make_conn_mat()	
make_mat2ij()	
make_post2pre()	
make_post2syn()	
make_post_slice()	
make_pre2post()	
make_pre2syn()	
make_pre_slice()	
requires(*syn_requires)	

16.3.4 brainpy.simulation.connectivity.GaussianProb

class brainpy.simulation.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`()	
`make_pre2post`()	
`make_pre2syn`()	
`make_pre_slice`()	
`requires`(*syn_requires)	

16.3.5 brainpy.simulation.connectivity.GaussianWeight

class brainpy.simulation.connectivity.**GaussianWeight**(*sigma*, *w_max*, *w_min=None*, *normalize=True*, *include_self=True*, *seed=None*)

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*, *seed=None*)
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`()	
`make_post_slice`()	
`make_pre2post`()	
`make_pre2syn`()	
`make_pre_slice`()	
`requires`(*syn_requires)	

16.3.6 brainpy.simulation.connectivity.DOG

class `brainpy.simulation.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`()	
`make_pre2post`()	
`make_pre2syn`()	
`make_pre_slice`()	
`requires`(*syn_requires)	

16.3.7 brainpy.simulation.connectivity.SmallWorld

class brainpy.simulation.connectivity.**SmallWorld**(*num_neighbor*, *prob*, *directed=False*, *include_self=False*)

Build a Watts–Strogatz small-world graph.

Parameters

- **num_neighbor** (*int*) – Each node is joined with its *k* nearest neighbors in a ring topology.
- **prob** (*float*) – The probability of rewiring each edge
- **directed** (*bool*) – Whether the graph is a directed graph.
- **include_self** (*bool*) – Whether include the node self.

Notes

First create a ring over num_node nodes[1]. Then each node in the ring is joined to its $num_neighbor$ nearest neighbors (or $num_neighbor - 1$ neighbors if $num_neighbor$ is odd). Then shortcuts are created by replacing some edges as follows: for each edge (u, v) in the underlying "num_node-ring with $num_neighbor$ nearest neighbors" with probability $prob$ replace it with a new edge (u, w) with uniformly random choice of existing node w.

[1] Duncan J. Watts and Steven H. Strogatz, Collective dynamics of small-world networks, Nature, 393, pp. 440–442, 1998.

References

`__init__`(*num_neighbor*, *prob*, *directed=False*, *include_self=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(num_neighbor, prob[, directed, . . .])	Initialize self.
`make_conn_mat`()	
`make_mat2ij`()	
`make_post2pre`()	
`make_post2syn`()	
`make_post_slice`()	
`make_pre2post`()	
`make_pre2syn`()	
`make_pre_slice`()	
`requires`(*syn_requires)	

16.3.8 brainpy.simulation.connectivity.ScaleFreeBA

`class brainpy.simulation.connectivity.`**`ScaleFreeBA`**(*m*, *directed=False*, *seed=None*)
Build a random graph according to the Barabási–Albert preferential attachment model.

A graph of num_node nodes is grown by attaching new nodes each with m edges that are preferentially attached to existing nodes with high degree.

Parameters

- **`m`** (*`int`*) – Number of edges to attach from a new node to existing nodes
- **`seed`** (*`integer, random_state, or None (default)`*) – Indicator of random number generation state.

Raises **`ValueError`** – If *m* does not satisfy `1 <= m < n`.

References

`__init__`(*m*, *directed=False*, *seed=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(m[, directed, seed])	Initialize self.
`make_conn_mat`()	
`make_mat2ij`()	
`make_post2pre`()	
`make_post2syn`()	
`make_post_slice`()	
`make_pre2post`()	
`make_pre2syn`()	
`make_pre_slice`()	
`requires`(*syn_requires)	

16.3.9 brainpy.simulation.connectivity.ScaleFreeBADual

class `brainpy.simulation.connectivity.`**`ScaleFreeBADual`**(*m1*, *m2*, *p*, *directed=False*, *seed=None*)
Build a random graph according to the dual Barabási–Albert preferential attachment model.

A graph of num_node nodes is grown by attaching new nodes each with either m_1 edges (with probability p) or m_2 edges (with probability $1-p$) that are preferentially attached to existing nodes with high degree.

Parameters

- **m1** (*int*) – Number of edges to attach from a new node to existing nodes with probability p
- **m2** (*int*) – Number of edges to attach from a new node to existing nodes with probability $1-p$
- **p** (*float*) – The probability of attaching m_1 edges (as opposed to m_2 edges)
- **seed** (*integer, random_state, or None (default)*) – Indicator of random number generation state.

Raises **ValueError** – If *m1* and *m2* do not satisfy `1 <= m1,m2 < n` or *p* does not satisfy `0 <= p <= 1`.

References

`__init__`(*m1*, *m2*, *p*, *directed=False*, *seed=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(m1, m2, p[, directed, seed])	Initialize self.
`make_conn_mat`()	
`make_mat2ij`()	
`make_post2pre`()	
`make_post2syn`()	
`make_post_slice`()	
`make_pre2post`()	
`make_pre2syn`()	
`make_pre_slice`()	
`requires`(*syn_requires)	

16.3.10 brainpy.simulation.connectivity.PowerLaw

class `brainpy.simulation.connectivity.`**`PowerLaw`**(*m*, *p*, *directed=False*, *seed=None*)
Holme and Kim algorithm for growing graphs with powerlaw degree distribution and approximate average clustering.

Parameters

- **m** (*int*) – the number of random edges to add for each new node
- **p** (*float,*) – Probability of adding a triangle after adding a random edge
- **seed** (*integer, random_state, or None (default)*) – Indicator of random number generation state.

Notes

The average clustering has a hard time getting above a certain cutoff that depends on *m*. This cutoff is often quite low. The transitivity (fraction of triangles to possible triangles) seems to decrease with network size.

It is essentially the Barabási–Albert (BA) growth model with an extra step that each random edge is followed by a chance of making an edge to one of its neighbors too (and thus a triangle).

This algorithm improves on BA in the sense that it enables a higher average clustering to be attained if desired.

It seems possible to have a disconnected graph with this algorithm since the initial *m* nodes may not be all linked to a new node on the first iteration like the BA model.

Raises `ValueError` – If *m* does not satisfy `1 <= m <= n` or *p* does not satisfy `0 <= p <= 1`.

References

`__init__`(*m*, *p*, *directed=False*, *seed=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(m, p[, directed, seed])	Initialize self.
`make_conn_mat`()	
`make_mat2ij`()	
`make_post2pre`()	
`make_post2syn`()	
`make_post_slice`()	
`make_pre2post`()	
`make_pre2syn`()	
`make_pre_slice`()	
`requires`(*syn_requires)	

`class brainpy.simulation.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.simulation.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.simulation.connectivity.`**`FixedProb`**(*prob*, *include_self=True*, *seed=None*, *method='matrix'*)
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.simulation.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.simulation.connectivity.GaussianWeight(`*sigma*, *w_max*, *w_min=None*, *normalize=True*, *include_self=True*, *seed=None*`)`

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.simulation.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.simulation.connectivity.**SmallWorld**(*num_neighbor*, *prob*, *directed=False*, *include_self=False*)

Build a Watts–Strogatz small-world graph.

Parameters

- **num_neighbor** (*int*) – Each node is joined with its *k* nearest neighbors in a ring topology.
- **prob** (*float*) – The probability of rewiring each edge
- **directed** (*bool*) – Whether the graph is a directed graph.
- **include_self** (*bool*) – Whether include the node self.

Notes

First create a ring over num_node nodes [1]_. Then each node in the ring is joined to its $num_neighbor$ nearest neighbors (or $num_neighbor - 1$ neighbors if $num_neighbor$ is odd). Then shortcuts are created by replacing some edges as follows: for each edge (u, v) in the underlying "num_node-ring with $num_neighbor$ nearest neighbors" with probability $prob$ replace it with a new edge (u, w) with uniformly random choice of existing node w.

References

class brainpy.simulation.connectivity.**ScaleFreeBA**(*m*, *directed=False*, *seed=None*)

Build a random graph according to the Barabási–Albert preferential attachment model.

A graph of num_node nodes is grown by attaching new nodes each with m edges that are preferentially attached to existing nodes with high degree.

Parameters

- **m** (*int*) – Number of edges to attach from a new node to existing nodes
- **seed** (*integer, random_state, or None (default)*) – Indicator of random number generation state.

Raises `ValueError` – If *m* does not satisfy `1 <= m < n`.

References

class `brainpy.simulation.connectivity.`**`ScaleFreeBADual`**(*m1*, *m2*, *p*, *directed=False*, *seed=None*)

Build a random graph according to the dual Barabási–Albert preferential attachment model.

A graph of num_node nodes is grown by attaching new nodes each with either m_1 edges (with probability p) or m_2 edges (with probability $1-p$) that are preferentially attached to existing nodes with high degree.

Parameters

- **`m1`** (*`int`*) – Number of edges to attach from a new node to existing nodes with probability p
- **`m2`** (*`int`*) – Number of edges to attach from a new node to existing nodes with probability $1-p$
- **`p`** (*`float`*) – The probability of attaching m_1 edges (as opposed to m_2 edges)
- **`seed`** (*`integer, random_state, or None (default)`*) – Indicator of random number generation state.

Raises **`ValueError`** – If *m1* and *m2* do not satisfy `1 <= m1,m2 < n` or *p* does not satisfy `0 <= p <= 1`.

References

class `brainpy.simulation.connectivity.`**`PowerLaw`**(*m*, *p*, *directed=False*, *seed=None*)

Holme and Kim algorithm for growing graphs with powerlaw degree distribution and approximate average clustering.

Parameters

- **`m`** (*`int`*) – the number of random edges to add for each new node
- **`p`** (*`float,`*) – Probability of adding a triangle after adding a random edge
- **`seed`** (*`integer, random_state, or None (default)`*) – Indicator of random number generation state.

Notes

The average clustering has a hard time getting above a certain cutoff that depends on *m*. This cutoff is often quite low. The transitivity (fraction of triangles to possible triangles) seems to decrease with network size.

It is essentially the Barabási–Albert (BA) growth model with an extra step that each random edge is followed by a chance of making an edge to one of its neighbors too (and thus a triangle).

This algorithm improves on BA in the sense that it enables a higher average clustering to be attained if desired.

It seems possible to have a disconnected graph with this algorithm since the initial *m* nodes may not be all linked to a new node on the first iteration like the BA model.

Raises **`ValueError`** – If *m* does not satisfy `1 <= m <= n` or *p* does not satisfy `0 <= p <= 1`.

References

CHAPTER SEVENTEEN

BRAINPY.INPUTS MODULE

brainpy.inputs module aims to provide several commonly used helper functions to help construct input currents.

```
[1]: import brainpy as bp

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

```
[2]: def show(current, duration, title):
    ts = np.arange(0, duration, 0.1)
    plt.plot(ts, current)
    plt.title(title)
    plt.xlabel('Time [ms]')
    plt.ylabel('Current Value')
    plt.show()
```

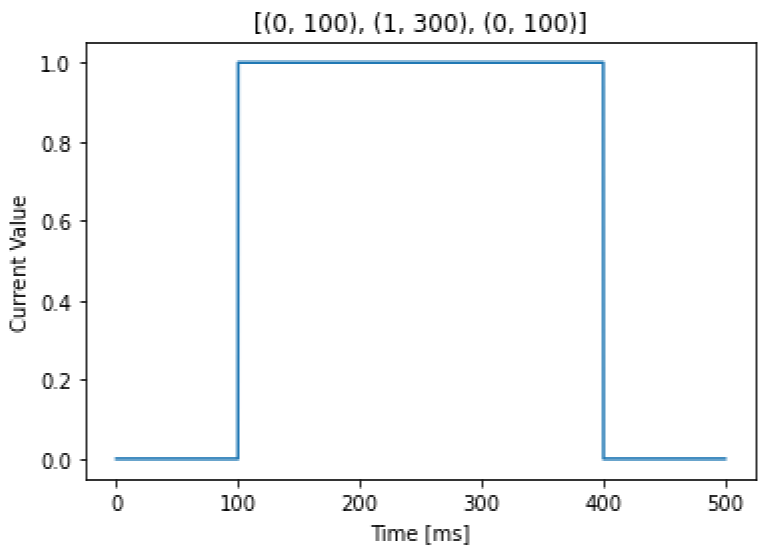
17.1 brainpy.inputs.period_input

brainpy.inputs.period_input() is an updated function of previous brainpy.inputs.constant_input() (see below).

Sometimes, we need input currents with different values in different 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:

```
[3]: current, duration = bp.inputs.period_input(values=[0, 1., 0.],
                                           durations=[100, 300, 100],
                                           return_length=True)
```

```
[4]: show(current, duration, '[(0, 100), (1, 300), (0, 100)]')
```



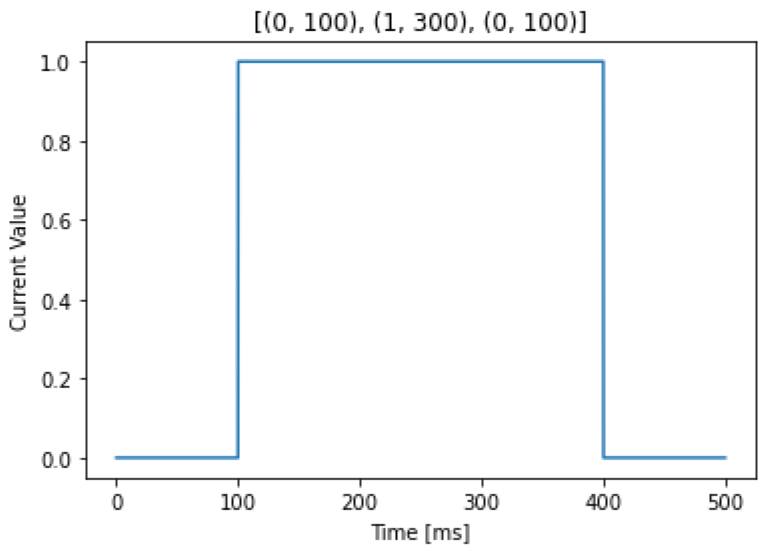
17.2 brainpy.inputs.constant_input

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

For the input created above, we can define it again with `constant_current()` by:

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

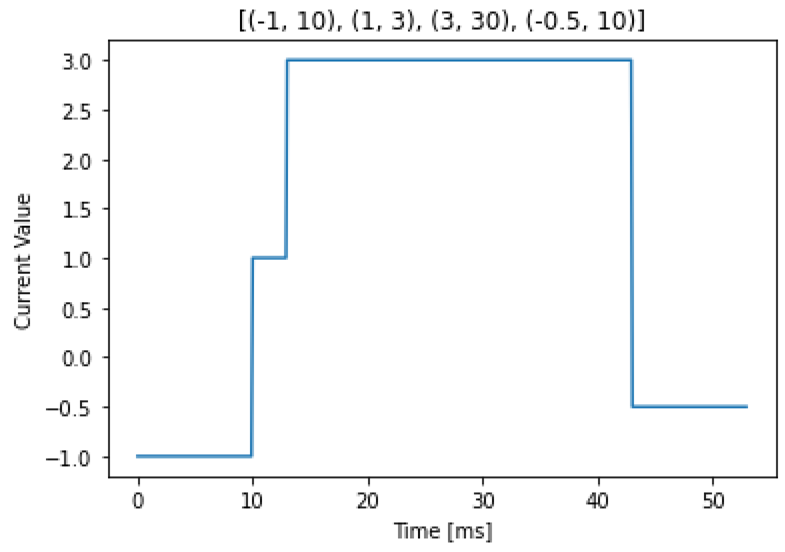
```
[6]: show(current, duration, '[(0, 100), (1, 300), (0, 100)]')
```



Another example is this:

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

[8]:
```
show(current, duration, '[(-1, 10), (1, 3), (3, 30), (-0.5, 10)]')
```



17.3 brainpy.inputs.spike_input

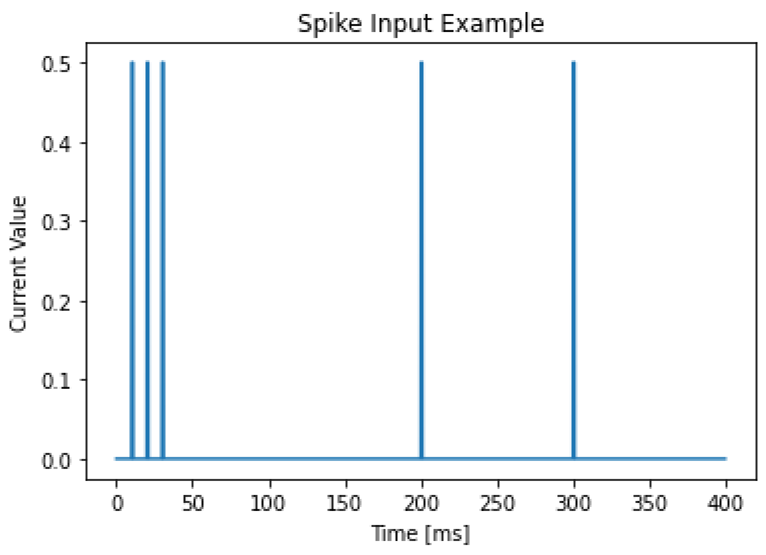
`brainpy.inputs.spike_input()` helps you to construct an input like a series of short-time spikes. It receives the following settings:

- `points` : The spike time-points. Must be an iterable object. For example, list, tuple, or arrays.
- `lengths` : The length of each point-current, mimicking the spike durations. It can be a scalar float to specify the unified duration. Or, it can be list/tuple/array of time lengths with the length same with `points`.
- `sizes` : The current sizes. It can be a scalar value. Or, it can be a list/tuple/array of spike current sizes with the length same with `points`.
- `duration` : The total current duration.
- `dt` : The time step precision. The default is None (will be initialized as the default `dt` step).

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:

[9]:
```
current = bp.inputs.spike_input(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 spike current size at each point
            duration=400.)
```

[10]:
```
show(current, 400, 'Spike Input Example')
```



17.4 brainpy.inputs.ramp_input

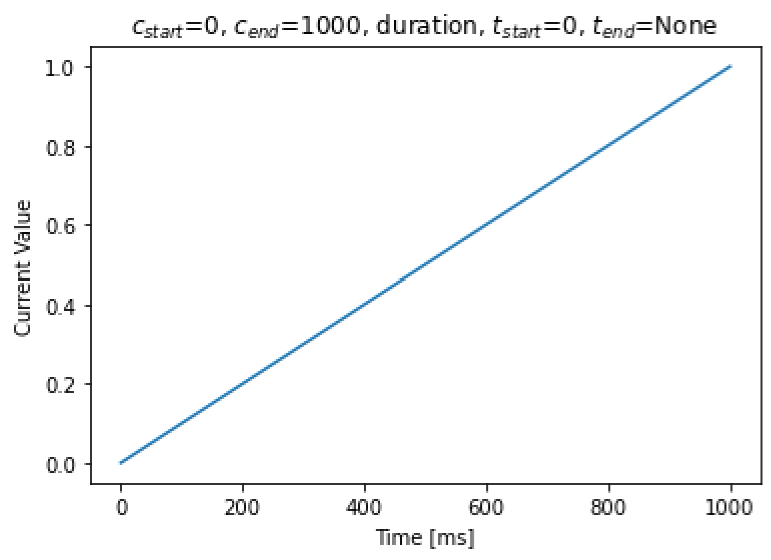
`brainpy.inputs.ramp_input()` mimics a ramp or a step current to the input of the circuit. It receives the following settings:

- `c_start` : The minimum (or maximum) current size.
- `c_end` : The maximum (or minimum) current size.
- `duration` : The total duration.
- `t_start` : The ramped current start time-point.
- `t_end` : The ramped current end time-point. Default is the None.
- `dt` : The current precision.

We illustrate the usage of `brainpy.inputs.ramp_input()` by two examples.

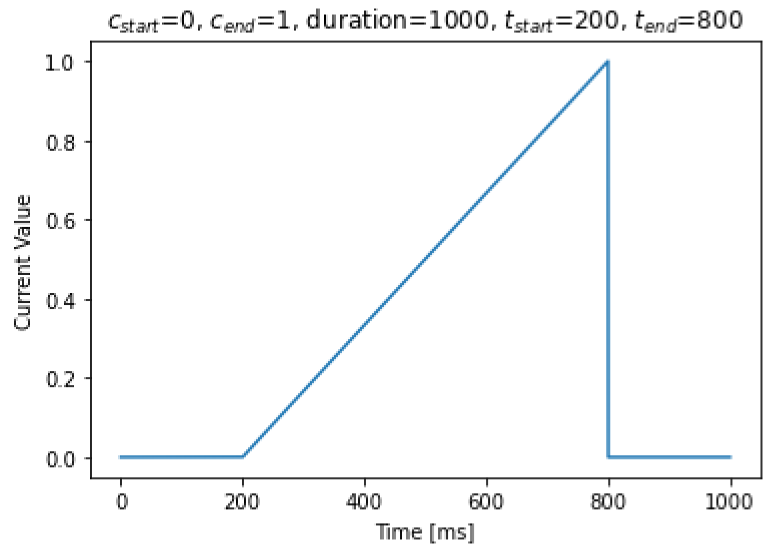
In the first example, we increase the current size from 0. to 1. between the start time (0 ms) and the end time (1000 ms).

[11]:
```
duration = 1000
current = bp.inputs.ramp_input(0, 1, duration)

show(current, duration, r'$c_{start}$=0, $c_{end}$=%d, duration, '
                        r'$t_{start}$=0, $t_{end}$=None' % (duration))
```



In the second example, we increase the current size from 0. to 1. from the 200 ms to 800 ms.

```
[12]: duration, t_start, t_end = 1000, 200, 800
current = bp.inputs.ramp_input(0, 1, duration, t_start, t_end)

show(current, duration, r'$c_{start}$=0, $c_{end}$=1, duration=%d, '
                        r'$t_{start}$=%d, $t_{end}$=%d' % (duration, t_start, t_end))
```



CHAPTER

EIGHTEEN

RELEASE NOTES

18.1 BrainPy 1.0.2

This release continues to improve the user-friendliness.

Highlights of core changes:

- Remove support for Numba-CUDA backend
- Super initialization *super(XXX, self).__init__()* can be done at anywhere (not required to add at the bottom of the *__init__()* function).
- Add the output message of the step function running error.
- More powerful support for Monitoring
- More powerful support for running order scheduling
- Remove *unsqueeze()* and *squeeze()* operations in `brainpy.ops`
- Add *reshape()* operation in `brainpy.ops`
- Improve docs for numerical solvers
- Improve tests for numerical solvers
- Add keywords checking in ODE numerical solvers
- Add more unified operations in brainpy.ops
- Support "@every" in steps and monitor functions
- Fix ODE solver bugs for class bounded function
- Add build phase in Monitor

18.2 BrainPy 1.0.1

- Fix bugs

18.3 BrainPy 1.0.0

- **NEW VERSION OF BRAINPY**
- Change the coding style into the object-oriented programming
- Systematically improve the documentation

18.4 BrainPy 0.3.5

- Add 'timeout' in sympy solver in neuron dynamics analysis
- Reconstruct and generalize phase plane analysis
- Generalize the repeat mode of `Network` to different running duration between two runs
- Update benchmarks
- Update detailed documentation

18.5 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

18.6 BrainPy 0.3.0

18.6.1 Computation API

- Rename "brainpy.numpy" to "brainpy.backend"
- Delete "pytorch", "tensorflow" backends
- Add "numba" requirement
- Add GPU support

18.6.2 Profile setting

- Delete "backend" profile setting, add "jit"

18.6.3 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

18.6.4 Package installation

- Remove "pypi" installation, installation now only rely on "conda"

18.7 BrainPy 0.2.4

18.7.1 API changes

- Fix bugs

18.8 BrainPy 0.2.3

18.8.1 API changes

- Add "animate_1D" in `visualization` module
- Add "PoissonInput", "SpikeTimeInput" and "FreqInput" in `inputs` module
- Update phase_portrait_analyzer.py

18.8.2 Models and examples

- Add CANN examples

18.9 BrainPy 0.2.2

18.9.1 API changes

- Redesign visualization
- Redesign connectivity
- Update docs

18.10 BrainPy 0.2.1

18.10.1 API changes

- Fix bugs in *numba import*
- Fix bugs in *numpy* mode with *scalar* model

18.11 BrainPy 0.2.0

18.11.1 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`.

18.11.2 Models and examples

- Neuron models: `HH model`, `LIF model`, `Izhikevich model`
- Synapse models: `AMPA`, `GABA`, `NMDA`, `STP`, `GapJunction`
- Network models: `gamma oscillation`

CHAPTER

NINETEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX