

目录

0. 简介	1.1
1. 神经元模型	1.2
1.1 生物背景	1.2.1
1.2 生理模型	1.2.2
1.3 简化模型	1.2.3
1.4 发放率模型	1.2.4
2. 突触模型	1.3
2.1 突触动力学模型	1.3.1
2.2 突触可塑性模型	1.3.2
3. 网络模型	1.4
3.1 兴奋-抑制平衡网络	1.4.1
3.2 抢择网络	1.4.2
3.3 连续吸引子神经网络	1.4.3
附录：模型代码	1.5
神经元模型	1.5.1
突触模型	1.5.2
网络模型	1.5.3

前言

在本手册中，我们将介绍一系列经典的计算神经科学模型，包括神经元模型、突触模型和网络模型，并提供它们的BrainPy——基于Python的计算神经科学及类脑计算平台——实现。

我们希望，本手册不仅能列出模型的定义、功能，也能对计算神经科学这一学科的脉络和思想有所涉及。通过阅读本手册，读者若能建立对计算神经科学的基本认识，知道如何在学术或应用场景中选择合适的模型、或对现象进行适当的建模，那就是我们在编辑本书时所期望的。

此外，模型后附BrainPy实现代码，帮助初学者快速上手，完成第一次仿真。对于熟悉计算神经科学的读者，我们也希望书中的例子能帮助大家了解BrainPy的优势、学习BrainPy的使用。

BrainPy介绍

在正式开始之前，我们希望先为读者简单介绍如何使用BrainPy实现计算神经科学模型，以方便读者理解附在每个模型之后的BrainPy实现代码。

BrainPy 是一个用于计算神经科学和类脑计算的Python平台。要使用 BrainPy 进行建模，用户通常需要完成以下三个步骤：

- 1) 为神经元和突触模型定义Python类。BrainPy预先定义了数种基类，用户在实现特定模型时，只需继承相应的基类，并在模型的Python类中定义特定的方法来告知BrainPy该模型在仿真的每个时刻所需的操作。在此过程中，BrainPy在微分方程（如ODE、SDE等）的数值积分、多种后端（如 Numpy 、 PyTorch 等）适配等功能上辅助用户，简化实现的代码逻辑。
- 2) 将模型的Python类实例化为代表神经元群或突触群的对象，将这些对象传入到BrainPy的 Network 类的构造函数中，初始化一个网络，并调用 run 方法进行仿真。
- 3) 调用BrainPy的测度模块 measure 或可视化模块 visualize 等，展示仿真结果。

带着上述对BrainPy的粗略理解，我们希望下述各节中的代码实例能够帮助读者更好地理解计算神经科学模型和其中蕴含的思想。下面，我们将按照1. 神经元模型, 2. 突触模型和3. 网络模型的顺序进行介绍。

关于BrainPy的下载及使用上的更多细节，请参考我们的Github仓库：<https://github.com/PKU-NIP-Lab/BrainPy>。

1. 神经元模型

本章首先介绍了建模神经元的基础——生物背景，随后按照从繁到简的顺序，介绍三类主要的神经元模型：生理模型、简化模型和发放率模型。

注：本章所述模型的完整BrainPy代码请见[附录](#)，或右键点此下载jupyter notebook版本。

1.1 生物背景

1.2 生理模型

1.3 简化模型

1.4 发放率模型

1.1 生物背景

作为神经系统的基本单位，神经元曾经在很长的一段时间内对研究者保持着神秘。直到18世纪，人们还普遍认为神经通过液体的流动与脑相联系。但到了19世纪，神经生物学取得了长足的进步，当时提出的“神经纤维”这一概念在过去的两个世纪中几经修正，终于演化成为今天我们所说的神经元。

与此同时，随着实验技术的进步，学界已为这些在我们神经系统中无休无止地工作的小东西画出了一张基本的肖像。要想用计算神经科学的方法建模神经元，我们必须先从这张真实细胞膜的肖像入手。

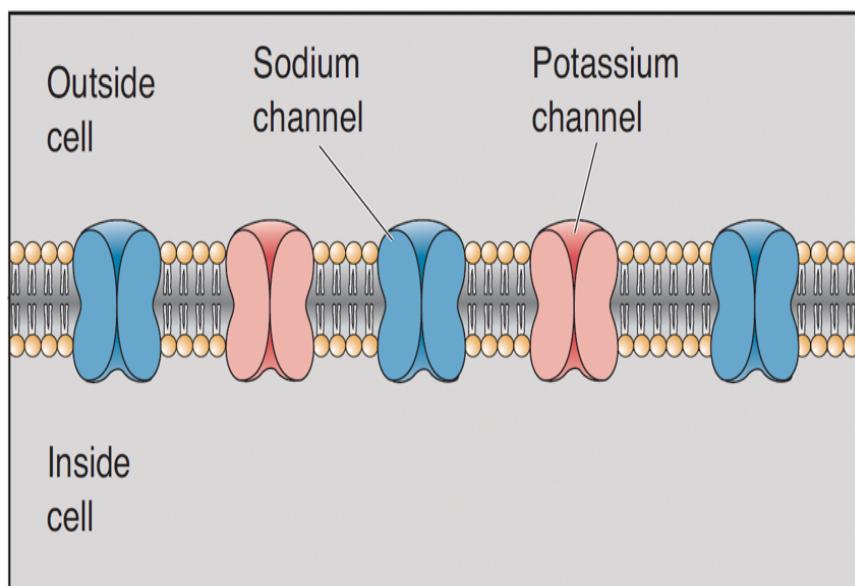


图1-1 神经元细胞膜示意图 (Bear et al., 2015¹)

上图是一张带有离子通道和磷脂双层膜的神经元膜一般性示意图。细胞膜将离子和液体划分为胞内和胞外两侧，部分地限制了胞内和胞外的物质交换，两侧离子不能自由交换达到电中性，于是产生了**膜电位**，即细胞膜两侧的电位差。

细胞膜内外环境状态的改变会引发膜电位的变化。存在在细胞膜附近（不管是膜内还是膜外）的一个离子主要受两种力的支配：细胞内外离子浓度差产生的扩散力和细胞内外电位差产生的电场力。当这两种力达到平衡时，离子的总受力为零，每种离子都达到其自身的离子平衡电位。与此同时，神经元的膜电位维持在一个小于零的值。

这个由所有离子平衡电位整合而成的膜电位称为**静息电位**，神经元则在此时进入所谓的**静息状态**。若不受外部干扰，神经元将自发寻找到平衡的静息状态，并维持在这一状态。

然而，从外部输入到循环输入，从刺激输入到噪声输入，每一毫秒，神经系统都接收到不计其数的外部扰动。面对这些输入，神经元发放**动作电位**（或**峰电位**）来在神经系统中处理、传递信息。

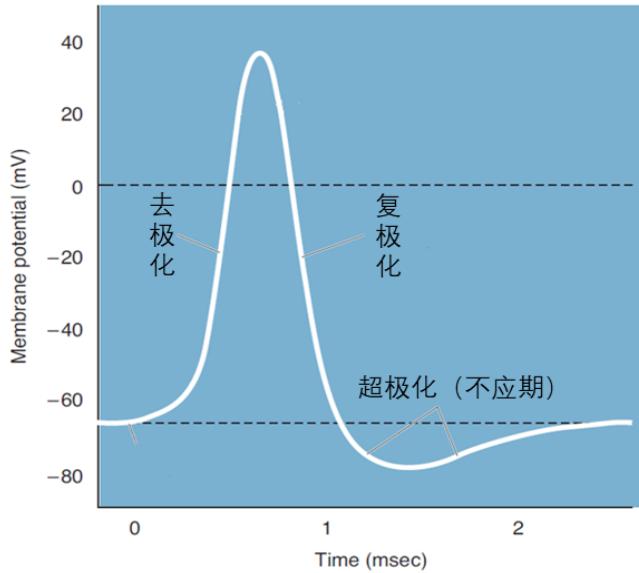


图1-2 动作电位（改编自Bear et al., 2015¹）

图1-2中画出了神经元膜电位在一个动作电位中随时间的变化。由于比如说，外部输入引起的环境变化，图1-1中疏水性磷脂双层膜上的离子通道会在打开和关闭的状态之间切换，调控着离子穿过离子通道进行交换的速率。

在受到外界兴奋性刺激时，特定的离子通道（主要是Na⁺通道和K⁺通道）状态发生改变，膜两侧相应离子的浓度变化，引发膜电位的剧变：它先上升到一个峰值，随后在短时间内迅速跌回一个小于静息电位的值。生物上，当膜电位发生这样的一系列变化时，我们说神经元产生了**动作电位**，或**峰电位**，或说**神经元发放**。

一个动作电位基本可以被分为三个阶段，**去极化、复极化和不应期**。在去极化阶段，钠离子流入细胞，钾离子流出细胞，但钠离子的流入速度更快，因此膜电位从低的静息电位（约-70mV）开始缓慢升高，随后，当膜电位高于阈值电位（约-55mV）后，离子流入和流出速度之间的差值逐渐增大，膜电位快速增长到大于0的峰值（约+40mV）。到达峰值后，钾离子流出速度变得大于钠离子流入速度，膜电位开始降低，并最终复极化到一个可能低于静息电位的值。此后，由于相对更低的膜电位以及离子通道的失活，神经元在短时间内立刻产生另一个动作电位的概率极小，这种情况将一直维持到我们称作不应期的这段时间结束。

单个动作电位的产生已经称得上复杂，但要知道，一个神经元可以在一秒之内产生多个动作电位。这些动作电位是以什么样的模式被产生的？不同类型的神经元可能在面对不同的输入时产生动作电位，而它们发放的特征可以被分为数种发放模式，下图画出了其中一部分。

1.1 生物背景

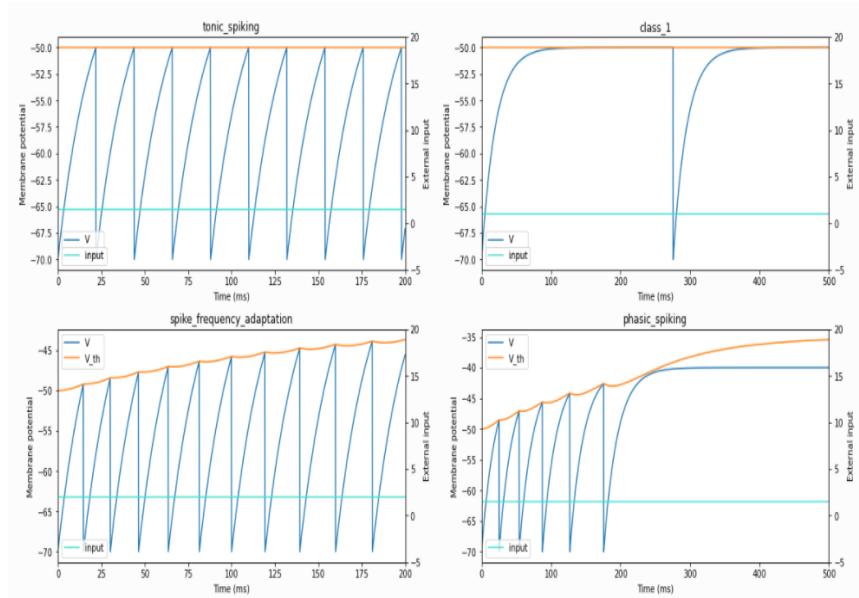


图1-3 部分神经元发放模式

在单神经元层面上，动作电位的形状和上述的发放模式正是计算神经科学的建模目标。

参考资料

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

1.2 生理模型

计算神经科学所希望建模的是真实生物的神经系统，因此，要了解神经元模型，可以先从和生理实际联系最紧密的模型入手。下面，本节将介绍受生理实验启发的最经典的模型：Hodgkin-Huxley模型。

1.2.1 Hodgkin-Huxley模型

Hodgkin和Huxley (1952) 在枪乌贼的巨轴突上用膜片钳技术记录了动作电位的产生，并提出了经典的神经元模型**Hodgkin-Huxley模型 (HH模型)**。

上一节我们已经介绍了神经元膜的一般结构，为了建模这样的结构，HH模型中将生物上的细胞膜转化为等效电路，如图1-4所示。

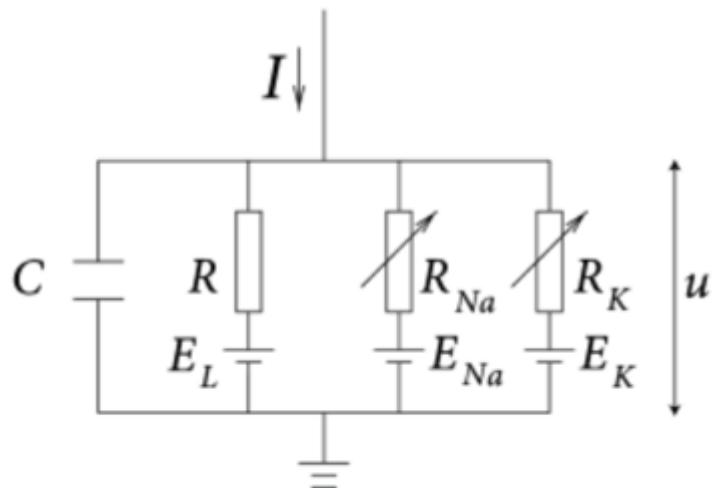


图1-4 神经元细胞膜的等效电路图 (Gerstner et al., 2014 [1](#))

上图是将图1-1中真实神经元膜转换为电子元件所得到的等效电路图，图中电容 C 表示低电导的疏水性磷脂双层膜，电流 I 表示外界输入。

右侧三个并联的电阻中， Na^+ 和 K^+ 通道被单独建模为两个可变电阻 R_{Na} 和 R_K （这是由于 Na^+ 和 K^+ 在动作电位的形成中特别重要），电阻 R 则代表膜上除了 Na^+ 通道和 K^+ 通道之外所有未指明的离子通道，有时也表示为下标 L 或者 $leak$ 。电源 E_{Na} , E_K 和 E_L 对应着由相应离子在膜两侧的浓度差所引起的电位差。

考虑基尔霍夫第一定律，即对于电路中的任一点，流入该点的总电流和流出该点的总电流相等，图1-4可被建模为如下所示的微分方程：

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

1.1 生物背景

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

这就是著名的Hodgkin-Huxley模型。注意在如上的 $\frac{dV}{dt}$ 方程中，右侧的前三项分别代表穿过钠离子通道，钾离子通道和其他非特定离子通道的电流，同时 $I(t)$ 表示外部输入。在方程左侧， $C \frac{dV}{dt} = \frac{dQ}{dt} = I$ 是穿过电容的电流。

在计算通过离子通道的电流时，除了欧姆定律 $I = U/R = gU$ 之外，HH模型还引入了三个门控变量m、n和h来表示离子通道的打开/关闭状态。准确地说，变量m和h控制着钠离子通道的状态，变量n控制着钾离子通道的状态。一个离子通道的真实电导是其最大电导 \bar{g} 和通道门控变量状态的乘积，比如通过Na+通道的电流

$$I = U/R_{Na} = g_{Na} * U_{Na} = \bar{g} * m^3 h * (V - E_{Na})$$

门控变量的动力学可以被表示为一种类马尔可夫的形式，其中 α_x 代表门控变量x的激活速率，而 β_x 代表x的失活速率。下述 α_x 和 β_x 的公式由实验数据拟合得到。

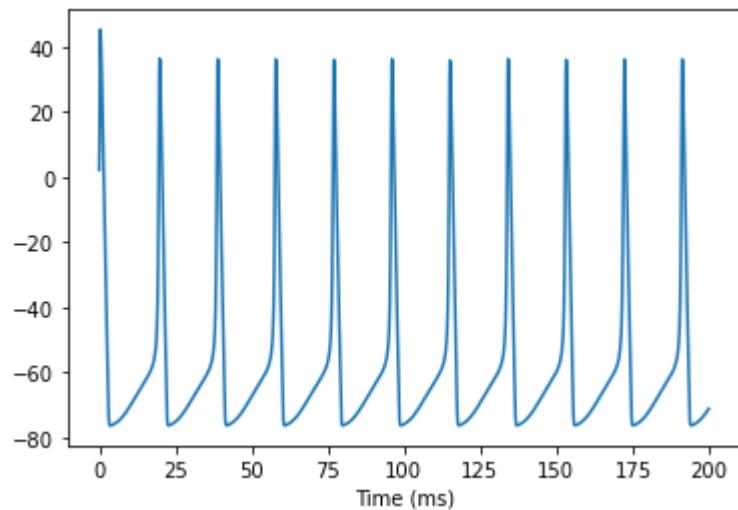
$$\begin{aligned}\alpha_m(V) &= \frac{0.1(V + 40)}{1 - \exp(\frac{-(V+40)}{10})} \\ \beta_m(V) &= 4.0 \exp(\frac{-(V + 65)}{18}) \\ \alpha_h(V) &= 0.07 \exp(\frac{-(V + 65)}{20}) \\ \beta_h(V) &= \frac{1}{1 + \exp(\frac{-(V+35)}{10})} \\ \alpha_n(V) &= \frac{0.01(V + 55)}{1 - \exp(\frac{-(V+55)}{10})} \\ \beta_n(V) &= 0.125 \exp(\frac{-(V + 65)}{80})\end{aligned}$$

1.1 生物背景

```

1      class HH(bp.NeuGroup): → bp.NeuGroup 类:
2          target_backend = 'general' → 神经元群。
3
4          @staticmethod → 设置模型后端。
5          @bp.odeint(method='exponential_euler') → 调用`bp.odeint`积分常微分方程。
6          def integral(V, m, h, n, t, C, gNa, ENa, gK, EK, gL, EL, Iext):
7              alpha_m = 0.1*(V+40)/(1-bp.ops.exp(-(V+40)/10)) α_m(V) = (0.1(V + 40))/(1 - exp(-(V + 40)/10))
8              beta_m = 4.0*bp.ops.exp(-(V+65)/18) β_m(V) = 4.0exp(-(V + 65)/18)
9              dmdt = alpha_m * (1 - m) - beta_m * m dx/dt = α(1 - x) - βx, x = m
10
11             alpha_h = 0.07*bp.ops.exp(-(V+65)/20) α_h(V) = 0.07 exp(-(V + 65)/20)
12             beta_h = 1/(1+bp.ops.exp(-(V+35)/10)) β_h(V) = 1/(1 + exp(-(V + 35)/10))
13             dhdt = alpha_h * (1 - h) - beta_h * h dx/dt = α(1 - x) - βx, x = h
14
15             alpha_n = 0.01*(V+55)/(1-bp.ops.exp(-(V+55)/10)) α_n(V) = (0.01(V + 55))/(1 - exp(-(V + 55)/10))
16             beta_n = 0.125*bp.ops.exp(-(V+65)/80) β_n(V) = 0.125exp(-(V + 65)/80)
17             dnndt = alpha_n * (1 - n) - beta_n * n dx/dt = α(1 - x) - βx, x = n
18
19             I_Na = (gNa * m ** 3.0 * h) * (V - ENa) ] C dV/dt = -(gNa m³ h(V - ENa) + gK n⁴ (V - EK) +
20             I_K = (gK * n ** 4.0) * (V - EK) gLeak(V - E_leak)) + I(t)
21             I_leak = gL * (V - EL)
22             dVdt = (-I_Na - I_K - I_leak + Iext) / C
23
24         return dVdt, dmdt, dhdt, dnndt
25
26     def __init__(self, size, ENa=50., gNa=120., EK=-77., gK=36.,
27                  EL=-54.387, gL=0.03, V_th=20., C=1.0, **kwargs):
28
29         # parameters
30         self.ENa = ENa
31         self.EK = EK
32         self.EL = EL
33         self.gNa = gNa
34         self.gK = gK
35         self.gL = gL
36         self.C = C
37         self.V_th = V_th
38
39         # variables
40         num = bp.size2len(size)
41         self.V = -65. * bp.ops.ones(num)
42         self.m = 0.5 * bp.ops.ones(num)
43         self.h = 0.6 * bp.ops.ones(num)
44         self.n = 0.32 * bp.ops.ones(num)
45         self.spike = bp.ops.zeros(num, dtype=bool)
46         self.input = bp.ops.zeros(num)
47
48     super(HH, self).__init__(size=size, **kwargs) → 传参`size`和`**kwargs`给
49
50     def update(self, _t):
51         V, m, h, n = self.integral(self.V, self.m, self.h, self.n, _t,
52                                     self.C, self.gNa, self.ENa, self.gK, → 数值积分方法采用
53                                     self.EK, self.gL, self.EL, self.input) → 向量形式更新变量。
54         self.spike = (self.V < self.V_th) * (V >= self.V_th) → 判断神经元是否发放。
55         self.V = V
56         self.m = m
57         self.h = h
58         self.n = n
59         self.input[:] = 0 → 重置当前时刻的外部输入。
```

BrainPy仿真的HH模型的V-t图如下所示。我们在上一节中曾经提到，真实的动作电位可以分为去极化、复极化和不应期三个阶段，这三个阶段可以与下图一一对应。另外，在去极化时，可以看到膜电位先是累积外部输入缓慢上升，一旦越过某个特定值（图中约在-55mV左右）就转为快速增长，这也复现了真实动作电位的形状。



参考资料

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

1.3 简化模型

启发自生理实验的Hodgkin-Huxley模型准确但昂贵，因此，研究者们提出了简化模型，希望能降低仿真的运行时间和计算资源消耗。

简化模型的特点是简单、易于计算，同时仍可复现神经元发放的主要特征。尽管它们的表示能力常常不如生理模型，但因为其简洁性，研究者们有时也可以接受一定的精度损失。

本节将从简单到复杂，依次介绍：泄露积分-发放模型、二次积分-发放模型、指数积分-发放模型、适应性指数积分-发放模型、Hindmarsh-Rose模型和归纳积分-发放模型。

1.3.1 泄漏积分-发放模型

最经典的简化模型，莫过于Lapicque (1907) 提出的**泄漏积分-发放模型** (Leaky Integrate-and-Fire model, **LIF model**)。LIF模型是由微分方程表示的积分过程和由条件判断表示的发放过程的结合：

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

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

$$V \leftarrow V_{reset}$$

其中 $\tau = RC$ 是LIF模型的时间常数， τ 越大，模型的动力学就越慢。LIF模型同样可以对应到等效电路图上，但比HH模型的等效电路更加简单，因为它不再建模钠离子通道和钾离子通道。实际上，LIF模型中只有电阻 R ，电容 C ，电源 V_{rest} 和外部输入 I 被建模。

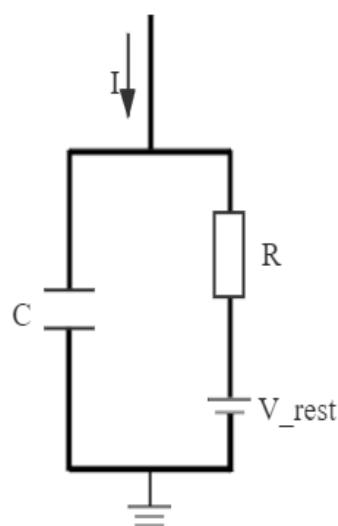


图1-5 LIF模型对应的细胞膜等效电路图（简化后）

1.1 生物背景

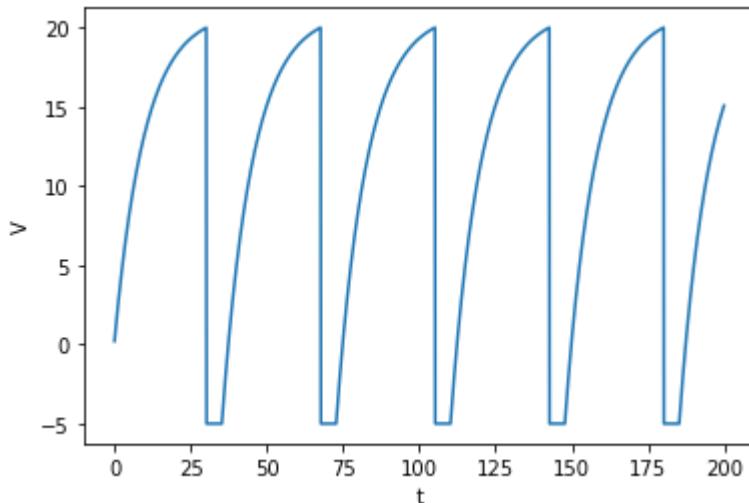
尽管LIF模型可以产生动作电位，但没有建模动作电位的形状。在发放动作电位前，LIF神经元膜电位的增长速度将逐渐降低，而并非像真实神经元那样先缓慢增长，在跨过阈值电位之后转为迅速增长。

原始LIF模型还忽略了不应期。要模拟不应期，必须再补充一个条件判断：如果当前时刻距离上次发放的时间小于不应期时长，则神经元处于不应期，膜电位 V 不再更新。

```

1   class LIF(bp.NeuGroup): → bp.NeuGroup 类:
2     target_backend = ['numpy', 'numba', 'numba-parallel', 'numba-cuda'] → 神经元群。
3
4     @staticmethod
5     def derivative(V, t, Iext, V_rest, R, tau): → 用户也可将后端名字加入列表，指定复数个可用后端。
6       dvdt = (-V + V_rest + R * Iext) / tau →  $\tau \frac{dV}{dt} = -(V - V_{rest}) + RI(t)$ 
7       return dvdt
8
9
10    def __init__(self, size, t_refractory=1., V_rest=0.,
11                 V_reset=-., V_th=20., R=1., tau=10., **kwargs):
12      # parameters
13      self.V_rest = V_rest
14      self.V_reset = V_reset
15      self.V_th = V_th
16      self.R = R
17      self.tau = tau
18      self.t_refractory = t_refractory
19
20      # variables
21      num = bp.size2len(size)
22      self.t_last_spike = bp.ops.ones(num) * -1e7
23      self.input = bp.ops.zeros(num)
24      self.refractory = bp.ops.zeros(num, dtype=bool)
25      self.spike = bp.ops.zeros(num, dtype=bool)
26      self.V = bp.ops.ones(num) * V_rest
27
28      self.integral = bp.odeint(self.derivative) → 调用`bp.odeint`积分常微分方程。
29      super(LIF, self).__init__(size=size, **kwargs) → 传参`size`和`**kwargs`给超类`bp.NeuGroup`的构造函数。
30
31    def update(self, _t):
32      for i in prange(self.size[0]): → 对于神经元群中的每个神经元:
33        spike = 0.
34        refractory = (_t - self.t_last_spike[i] <= self.t_refractory) → 检查神经元是否处于不应期。
35        if not refractory:
36          V = self.integral(self.V[i], _t, self.input[i],
37                            self.V_rest, self.R, self.tau) → 数值积分逐个更新变量。
38          spike = (V >= self.V_th) → 判断神经元是否发放。
39          if spike:
39            V = self.V_reset
40            self.t_last_spike[i] = _t → 重置神经元。
41            self.V[i] = V
42            self.spike[i] = spike
43            self.refractory[i] = refractory or spike
44            self.input[i] = 0. → 重置当前时刻的外部输入。

```

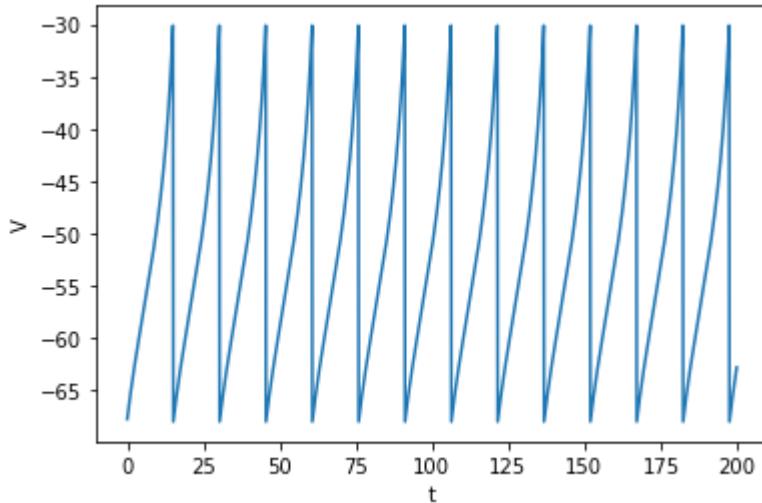


1.3.2 二次积分-发放模型

LIF模型固然简洁，但像上一节末尾所讲的那样，有着诸多限制。为了弥补它在表示能力上的缺陷，Latham等人（2000）提出了**二次积分-发放模型**（Quadratic Integrate-and-Fire model，**QuaIF model**）。在QuaIF模型中，微分方程右侧的二阶项使得神经元能产生和真实神经元更“像”的动作电位。

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

在上式中， a_0 和 V_C 共同控制着动作电位的初始化，其中， a_0 控制着发放前膜电位的增长速度，也即膜电位相对时间变化的斜率； V_c 是动作电位初始化的临界值，当膜电位 V 低于 V_C 时， V 缓慢增长，一旦越过 V_C ， V 就转为迅速增长。



1.3.3 指数积分-发放模型

指数积分-发放模型 (Exponential Integrate-and-Fire model, **ExpIF model**) (Fourcaud-Trocme et al., 2003) 在QualF模型的基础上更上一层，进一步提升了模型生成的动作电位的真实度。

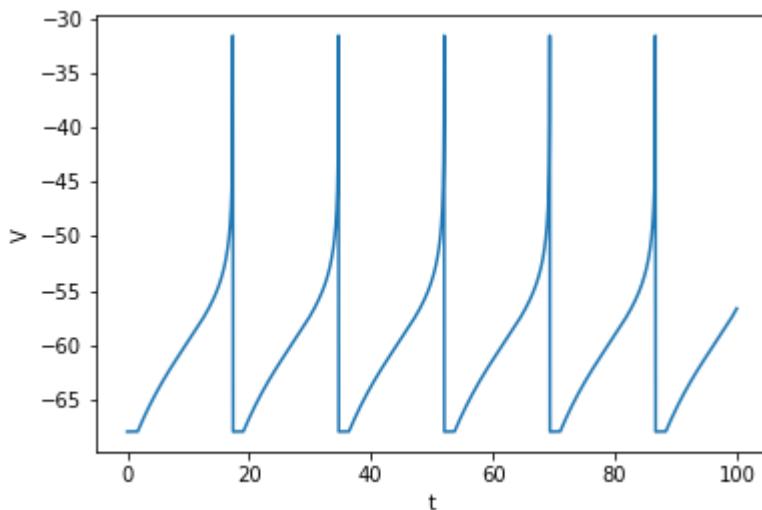
$$\tau \frac{dV}{dt} = -(V - V_{rest}) + \Delta_T e^{\frac{V-V_T}{\Delta_T}} + RI(t)$$

在指数项中 V_T 是动作电位初始化的临界值，在其下 V 缓慢增长，其上 V 迅速增长。 Δ_T 是ExpIF模型中动作电位的斜率。当 $\Delta_T \rightarrow 0$ 时，ExpIF模型中动作电位的形状将趋近于 $V_{th} = V_T$ 的LIF模型 (Fourcaud-Trocme et al., 2003)。

1.1 生物背景

```

1   class ExpIF(bp.NeuGroup): → bp.NeuGroup 类:
2       target_backend = 'general'
3
4       @staticmethod
5       def derivative(V, t, I_ext, V_rest, delta_T, V_T, R, tau):
6           exp_term = bp.ops.exp((V - V_T) / delta_T)
7           dvdt = (- (V - V_rest) + delta_T * exp_term + R * I_ext) / tau →  $\tau \frac{dV}{dt} = -(V - V_{rest}) + \Delta_T e^{\frac{V-V_T}{\Delta T}} + RI(t)$ 
8           return dvdt
9
10      def __init__(self, size, V_rest=-65., V_reset=-68.,
11                  V_th=-30., V_T=-59.9, delta_T=3.48,
12                  R=10., C=1., tau=10., t_refractory=1.7,
13                  **kwargs):
14          # parameters
15          self.V_rest = V_rest
16          self.V_reset = V_reset
17          self.V_th = V_th
18          self.V_T = V_T
19          self.delta_T = delta_T
20          self.R = R
21          self.C = C
22          self.tau = tau
23          self.t_refractory = t_refractory → 模型参数保存为浮点数。
24
25          # variables
26          self.V = bp.ops.ones(size) * V_rest
27          self.input = bp.ops.zeros(size)
28          self.spike = bp.ops.zeros(size, dtype=bool)
29          self.refractory = bp.ops.zeros(size, dtype=bool)
30          self.t_last_spike = bp.ops.ones(size) * -1e7 → 模型变量保存为浮点数的向量。
31
32          self.integral = bp.odeint(self.derivative) → 调用'bp.odeint'积分常微分方程。
33          super(ExpIF, self).__init__(size=size, **kwargs) → 参数'method'置为默认值'euler'。
34
35      def update(self, _t):
36          for i in prange(self.num): → 对于神经元群中的每个神经元:
37              spike = 0.
38              refractory = (_t - self.t_last_spike[i] <= self.t_refractory) → 检查神经元是否处于不应期。
39              if not refractory:
40                  V = self.integral(
41                      self.V[i], _t, self.input[i], self.V_rest, → 数值积分逐个更新变量。
42                      self.delta_T, self.V_T, self.R, self.tau
43                  )
44                  spike = (V >= self.V_th) → 判断神经元是否发放。
45                  if spike:
46                      V = self.V_reset
47                      self.t_last_spike[i] = _t → 重置神经元。
48                      self.V[i] = V
49                      self.spike[i] = spike
50                      self.refractory[i] = refractory or spike
51                      self.input[:] = 0. → 重置当前时刻的外部输入。
```



在上图中可以看到，ExpIF模型中膜电位相对于时间的斜率在每次上升到 V_T 值 (-59.9mV) 附近时，都发生了一个明显的转变，这是由于微分方程中指数项的调控。比起QualF模型，这种转变显得更加自然。

1.3.4 适应性指数积分-发放模型

在以上诸积分-发放模型中，建模了神经元的标准动作电位，但尚有许多神经元的行为未被涉及。

让我们稍稍离题，请读者做这样一种想象：你独自一人来到夜晚的海边，一开始，你闻到海风里的腥味，忍不住深吸一口气（或者捂住鼻子）。但过了一会儿，不管主观上想要亲近还是远离大海，你再也闻不见这种腥气——或者至少是以为自己闻不见了。这是因为你的嗅觉系统习惯了这种刺激，不再无休无止地提醒你的大脑附近存在着异味了。

上面这个例子中，对鱼腥味发生**适应**的是整个嗅觉感知系统。不过，在单神经元尺度上，也存在类似的行为。当特定类型的神经元面对恒定的外部刺激时，一开始神经元高频发放，随后发放率逐渐降低，最终稳定在一个较小值，这就是神经元的适应行为。

为了复现这种行为，研究者们在已有的积分-发放模型（如LIF、QualF和ExpIF模型等）上增加了权重变量 w 。这里我们介绍其中的**适应性指数积分-发放模型**（Adaptive Exponential Integrate-and-Fire model，**AdExIF model**）（Gerstner et al., 2014）。

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

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

如其名所示，AdExIF模型的第一个微分方程和我们上面介绍的ExpIF模型非常相似，唯一区别是方程右侧增加了控制神经元适应行为的权重项，即 $-Rw$ 一项。

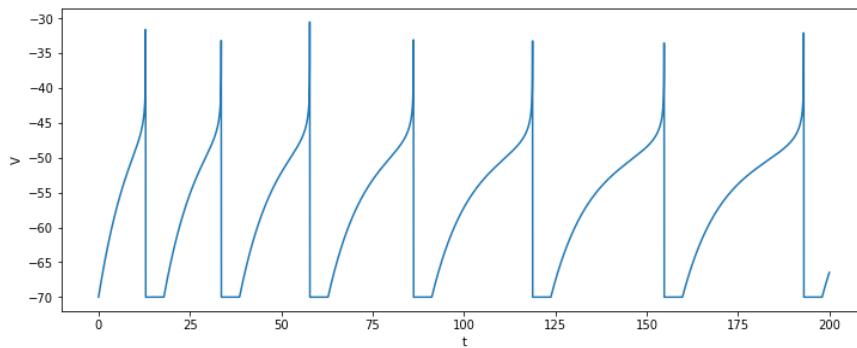
权重项中 w 受到第二个微分方程的调控。 a 描述了权重变量 w 对 V 的下阈值波动的敏感性， b 表示 w 在一次发放后的增长值，另外， w 也随时间衰减。

在这样的一个动力学系统中，给神经元一个恒定输入，在连续数次发放后， w 的值将会上升到一个高点，减慢 V 的增长速度，从而降低神经元的发放率。

1.1 生物背景

```

1  class AdExIF(bp.NeuGroup): → bp.NeuGroup 类:
2      target_backend = 'general' → 神经元群。
3
4      @staticmethod
5      def derivative(V, w, t, I_ext, V_rest, delta_T, V_T, R, tau, tau_w, a):
6          exp_term = bp.ops.exp((V-V_T)/delta_T) }  $\tau \frac{dV}{dt} = -(V - V_{rest}) + \Delta_T e^{\frac{V-V_T}{\Delta T}} + RI(t)$ 
7          dVdt = (- (V - V_rest) + delta_T * exp_term - R * w + R * I_ext) / tau }  $\tau_w \frac{dw}{dt} = a(V - V_{rest}) - w + b \tau_w \sum \delta(t - t_f)$ 
8
9          dwdt = (a * (V - V_rest) - w) / tau_w →  $\tau_w \frac{dw}{dt} = a(V - V_{rest}) - w + b \tau_w \sum \delta(t - t_f)$ 
10
11     return dVdt, dwdt
12
13     def __init__(self, size, V_rest=-65., V_reset=-68.,
14                 V_th=-30., V_T=-59.9, delta_T=3.48,
15                 a=1., b=1., R=10., tau=10., tau_w=30.,
16                 t_refractory=0., **kwargs):
17         # parameters
18         self.V_rest = V_rest
19         self.V_reset = V_reset
20         self.V_th = V_th
21         self.V_T = V_T
22         self.delta_T = delta_T
23         self.a = a
24         self.b = b
25         self.R = R
26         self.tau = tau
27         self.tau_w = tau_w
28         self.t_refractory = t_refractory } 模型参数保存为浮点数。
29
30         # variables
31         num = bp.size2len(size)
32         self.V = bp.ops.ones(num) * V_reset
33         self.w = bp.ops.zeros(size)
34         self.input = bp.ops.zeros(num)
35         self.spike = bp.ops.zeros(num, dtype=bool)
36         self.refractory = bp.ops.zeros(num, dtype=bool)
37         self.t_last_spike = bp.ops.ones(num) * -1e7 } 模型变量保存为浮点数的向量。
38
39         self.integral = bp.odeint(f=self.derivative, method='euler') → 调用`bp.odeint`积分常微分方程。
40                                         → 设置参数`method`来选择所用的数值积分方法。
41         super(AdExIF, self).__init__(size=size, **kwargs) → 传参`size`和`**kwargs`给超类`bp.NeuGroup`的构造函数。
42
43     def update(self, _t):
44         for i in prange(self.size[0]): → 对于神经元群中的每个神经元:
45             spike = 0.
46             refractory = (_t - self.t_last_spike[i] <= self.t_refractory) → 检查神经元是否处于不应期。
47             if not refractory:
48                 V, w = self.integral(self.V[i], self.w[i], _t, self.input[i],
49                                     self.V_rest, self.delta_T, → 数值积分逐个更新变量。
50                                     self.V_T, self.R, self.tau, self.tau_w, self.a)
51                 spike = (V >= self.V_th) → 判断神经元是否发放。
52                 if spike:
53                     V = self.V_rest
54                     w += self.b } 重置膜电位, 给权重w增加狄拉克δ项:
55                     self.t_last_spike[i] = _t }  $\tau_w \frac{dw}{dt} = a(V - V_{rest}) - w + b \tau_w \sum \delta(t - t_f)$ 
56                     self.V[i] = V
57                     self.w[i] = w
58                     self.spike[i] = spike
59                     self.refractory[i] = refractory or spike
60                     self.input[i] = 0. → 重置当前时刻的外部输入。
```



1.3.5 Hindmarsh-Rose模型

神经元的行为并不总是符合标准模板的。比如，不是所有神经元都在每次发放后等待一整个不应期才进行第二次发放。有时，部分神经元面对特定类型的输入，能够产生短时间内的连续发放。和以上所有模型产生的**脉冲式发放**相对地，我们称这种发放模式为**爆发** (bursting) ，或**爆发式发放** (bursting firing) 。

为了模拟神经元的爆发，Hindmarsh和Rose (1984) 提出了**Hindmarsh-Rose模型**，引入了第三个模型变量 z 作为慢变量控制爆发。

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

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

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

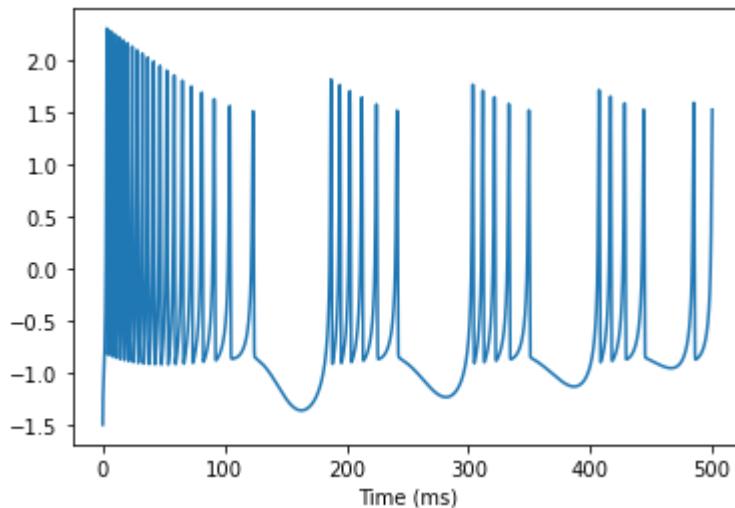
式中，变量 V 表示膜电位， y 和 z 是两个门控变量。在 dV/dt 方程中的参数 b 允许模型在脉冲式发放和爆发式发放之间切换，并控制脉冲式发放的频率。 dz/dt 方程中，参数 r 控制慢变量 z 的变化速率，影响神经元爆发式发放时，每次爆发包含的动作电位个数，并和参数 b 共同控制脉冲式发放的频率；参数 s 控制着神经元的适应行为。其它参数根据发放模式拟合得到。

1.1 生物背景

```

1  class HindmarshRose(bp.NeuGroup): → bp.NeuGroup 类:
2      target_backend = 'general'
3
4      @staticmethod
5      def derivative(V, y, z, t, a, b, I_ext, c, d, r, s, V_rest):
6          dVdt = y - a * V * V * V + b * V * V - z + I_ext   dV/dt =  $y - aV^3 + bV^2 - z + I$ 
7          dydt = c - d * V * V - y                         dy/dt =  $c - dV^2 - y$ 
8          dzdt = r * (s * (V - V_rest) - z)                 dz/dt =  $r(s(V - V_{rest}) - z)$ 
9          return dVdt, dydt, dzdt
10
11     def __init__(self, size, a=1., b=3.,
12                  c=1., d=5., r=0.01, s=4.,
13                  V_rest=-1.6, **kwargs):
14         # parameters
15         self.a = a
16         self.b = b
17         self.c = c
18         self.d = d
19         self.r = r
20         self.s = s
21         self.V_rest = V_rest
22
23     # variables
24     num = bp.size2len(size)
25     self.z = bp.ops.zeros(num)
26     self.input = bp.ops.zeros(num)
27     self.V = bp.ops.ones(num) * -1.6
28     self.y = bp.ops.ones(num) * -10.
29     self.spike = bp.ops.zeros(num, dtype=bool)
30
31     self.integral = bp.odeint(f=self.derivative) → 调用'bp.odeint'积分常微分方程。
32     super(HindmarshRose, self).__init__(size=size, **kwargs) → 参数'method'置为默认值'euler'。
33
34     def update(self, _t):
35         for i in prange(self.num): → 对于神经元群中的每个神经元:
36             V, self.y[i], self.z[i] = self.integral(
37                 self.V[i], self.y[i], self.z[i], _t, → 数值积分逐个更新变量。
38                 self.a, self.b, self.input[i],
39                 self.c, self.d, self.r, self.s,
40                 self.V_rest)
41             self.V[i] = V
42             self.input[i] = 0. → 重置当前时刻的外部输入。

```



下图中画出了 V 、 y 、 z 三个变量随时间的变化。可以看到，慢变量 z 的变化速率确实慢于 V 和 y 。而且， V 和 y 在仿真过程中呈近似周期性的变化。

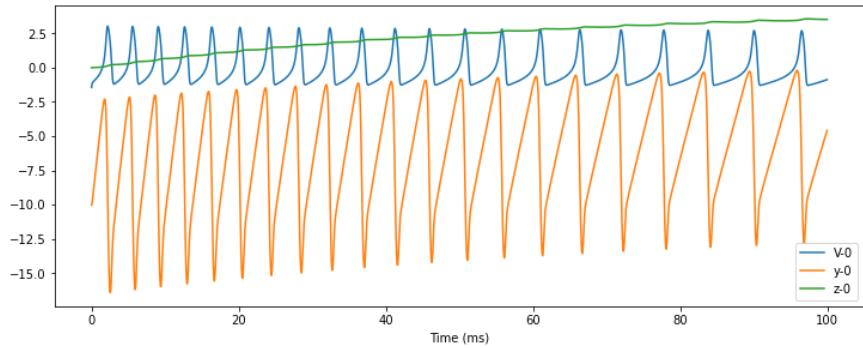


图1-6 Hindmarsh-Rose模型变量随时间的变化

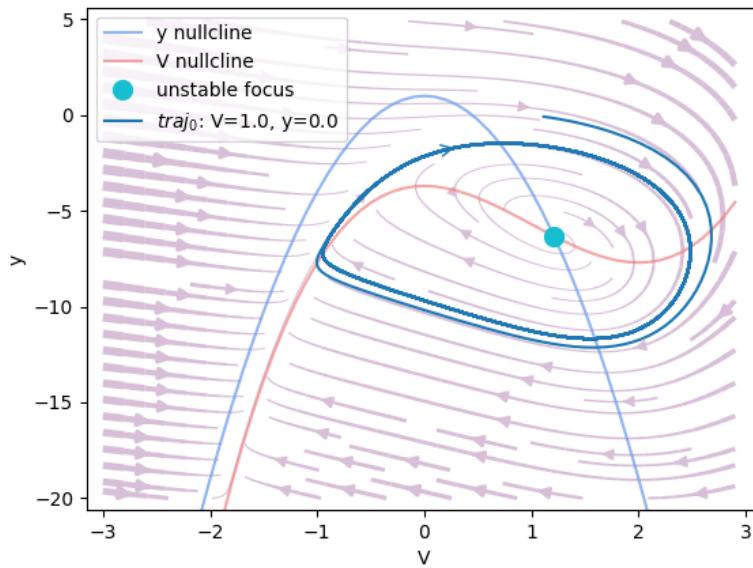
利用BrainPy的理论分析模块 `analysis`，我们可以找出这种周期性的产生原因。将慢变量 z 近似为常数，则Hindmarsh-Rose模型的二维相图中，变量 V 和 y 的轨迹趋近于一个极限环。因此，这两个变量的值会沿极限环周期性变化。

```

68 # Phase plane analysis
69 phase_plane_analyzer = bp.analysis.PhasePlane(
70     neu.integral,                                → 定义要分析的动力学系统。
71     target_vars={'V': [-3., 3.], 'y': [-20., 5.]}, → 相图中希望画出的变量。
72     fixed_vars={'z': 0.},                          → 相图中希望固定的变量。
73     pars_update={'I_ext': param[mode][1], 'a': 1., 'b': 3.,      } 其他希望固定的参数。
74         'c': 1., 'd': 5., 'r': 0.01, 's': 4.,      }
75         'V_rest': -1.6}                         }

76 )
77 phase_plane_analyzer.plot_nullcline()          → 绘制零点线。
78 phase_plane_analyzer.plot_fixed_point()        → 绘制稳定点。
79 phase_plane_analyzer.plot_vector_field()        → 绘制向量场。
80 phase_plane_analyzer.plot_trajectory(          } 绘制轨迹。
81     [{"V": 1., "y": 0., "z": -0.}],           } 定义轨迹的起始点和仿真时间。
82     duration=100.,
83     show=True
84 )

```



1.3.6 归纳积分-发放模型

归纳积分-发放模型 (Generalized Integrate-and-Fire model, **GeneralizedIF model**) (Mihalas et al., 2009) 整合了多种发放模式。该模型包括四个变量，能产生二十种发放模式，并通过调参在各模式之间切换。

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

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

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

当 V 达到 V_{th} 时，神经元发放：

$$I_j \leftarrow R_j I_j + A_j$$

$$V \leftarrow V_{reset}$$

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

在 dV/dt 的方程中，和所有积分-发放模型一样， τ 表示时间常数， V 表示膜电位， V_{rest} 表示静息电位， R 为电阻，而 I 为外部输入。

不过，在GIF模型中，数目可变的内部电流被加入到方程中，写作 $\sum_j I_j$ 一项。每一个 I_j 都代表神经元中的一个内部电流，并以速率 k_j 衰减。 R_j 和 A_j 是自由参数， R_j 描述 I_j 重置值对发放前 I_j 值的依赖， A_j 则是发放后加到 I_j 上的一个常数。

阈值电位 V_{th} 也是可变的，受两个参数的调控： a 描述了 V_{th} 对膜电位 V 的依赖， b 描述了 V_{th} 接近阈值电位在时间趋近于无穷大时的值 $V_{th\infty}$ 的速率。

$V_{threset}$ 是当神经元发放时，阈值电位被重置到的值。

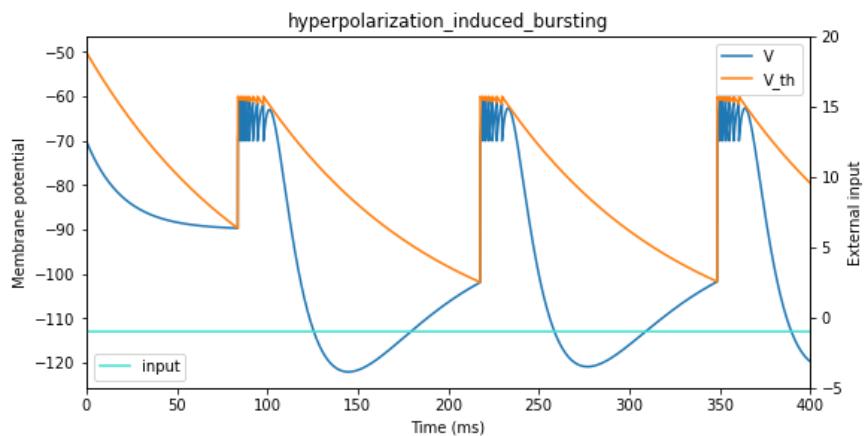
1.1 生物背景

```

1  class GeneralizedIF(bp.NeuGroup): → bp.NeuGroup 类:
2      target_backend = 'general'
3
4      @staticmethod
5      def derivative(I1, I2, V_th, V, t,
6                      k1, k2, a, V_rest, b, V_th_inf,
7                      R, I_ext, tau):
8          dI1dt = - k1 * I1
9          dI2dt = - k2 * I2
10         dVthdt = a * (V - V_rest) - b * (V_th - V_th_inf)
11         dVdt = (- (V - V_rest) + R * I_ext + R * I1 + R * I2) / tau
12         return dI1dt, dI2dt, dVthdt, dVdt
13
14     def __init__(self, size, V_rest=-70., V_reset=-70.,
15                  V_th_inf=-50., V_th_reset=-60., R=20., tau=20.,
16                  a=0., b=0.01, k1=0.2, k2=0.02,
17                  R1=0., R2=1., A1=0., A2=0.,
18                  **kwargs):
19         # params
20         self.V_rest = V_rest
21         self.V_reset = V_reset
22         self.V_th_inf = V_th_inf
23         self.V_th_reset = V_th_reset
24         self.R = R
25         self.tau = tau
26         self.a = a
27         self.b = b
28         self.k1 = k1
29         self.k2 = k2
30         self.R1 = R1
31         self.R2 = R2
32         self.A1 = A1
33         self.A2 = A2
34
35         # vars
36         self.input = bp.ops.zeros(size)
37         self.spike = bp.ops.zeros(size, dtype=bool)
38         self.I1 = bp.ops.zeros(size)
39         self.I2 = bp.ops.zeros(size)
40         self.V = bp.ops.ones(size) * -70.
41         self.V_th = bp.ops.ones(size) * -50.
42
43         self.integral = bp.odeint(self.derivative) → 调用'bp.odeint'积分常微分方程。
44         super(GeneralizedIF, self).__init__(size=size, **kwargs) → 参数'method'置为默认值'euler'。
45                                         → 传参'size'和 '**kwargs' 给超类bp.NeuGroup的构造函数。
46
47     def update(self, _t):
48         for i in prange(self.size[0]): → 对于神经元群中的每个神经元:
49             I1, I2, V_th, V = self.integral(
50                 self.I1[i], self.I2[i], self.V_th[i], self.V[i], _t,
51                 self.k1, self.k2, self.a, self.V_rest,
52                 self.b, self.V_th_inf,
53                 self.R, self.input[i], self.tau
54             ) → 数值积分逐个更新变量。
55             self.spike[i] = self.V_th[i] < V → 判断神经元是否发放。
56             if self.spike[i]:
57                 V = self.V_reset
58                 I1 = self.R1 * I1 + self.A1
59                 I2 = self.R2 * I2 + self.A2
60                 V_th = max(V_th, self.V_th_reset)
61                 self.I1[i] = I1
62                 self.I2[i] = I2
63                 self.V_th[i] = V_th
64                 self.V[i] = V
65                 self.f = 0.
66                 self.input[:] = self.f → 重置当前时刻的外部输入。

```

1.1 生物背景



1.4 发放率模型

发放率模型比简化模型更加简单，只考虑神经元的发放频率。发放率模型很少被单独提出，通常是作为网络中的基本计算单元出现，一个计算单元通常被认为是代表了一个神经元群的平均活动。为了方便，我们通过介绍经典的Wilson和Cowan网络模型来介绍发放率模型。

1.4.1 发放率单元

Wilson和Cowan (1972) 提出一个极简的模型来表示在兴奋性和抑制性皮层神经元微柱中的活动。变量 a_e 和 a_i 中的每个元素都表示一个包含大量神经元的皮层微柱中神经元群的平均活动水平。

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

下标 $x \in \{e, i\}$ 表示该参数或变量对应兴奋性或抑制性的神经元群。在微分方程中， τ_x 表示神经元群的时间常数，参数 k_x 和 r_x 共同控制不应期， a_x 和 θ_x 分别是Sigmoid函数 $S(\text{input})$ 的斜率和相位，且兴奋性和抑制性的神经元群分别收到外界输入 I_{ext_x} 。

```

1 class FiringRateUnit(bp.NeuGroup): → bp.NeuGroup 类:
2     target_backend = 'general' 神经元群。
3
4     @staticmethod
5     def derivative(a_e, a_i, t,
6                     k_e, r_e, c1, c2, I_ext_e, slope_e, theta_e, tau_e,
7                     k_i, r_i, c3, c4, I_ext_i, slope_i, theta_i, tau_i):
8         x_ae = c1 * a_e - c2 * a_i + I_ext_e
9         sigmoid_ae_l = 1 / (1 + bp.ops.exp(-slope_e * (x_ae - theta_e)))
10        sigmoid_ae_r = 1 / (1 + bp.ops.exp(slope_e * theta_e)) S(x) =  $\frac{1}{1 + \exp(-a(x - 0))} - \frac{1}{1 + \exp(a(0 - x))}$ 
11        sigmoid_ae = sigmoid_ae_l - sigmoid_ae_r
12        daedt = (-a_e + (k_e - r_e * a_e) * sigmoid_ae) / tau_e
13
14        x_ai = c3 * a_e - c4 * a_i + I_ext_i  $\tau_e \frac{da_e}{dt} = -a_e + (k_e - r_e * a_e) * S_e(c_1 a_e - c_2 a_i + I_{ext_e})$ 
15        sigmoid_ai_l = 1 / (1 + bp.ops.exp(-slope_i * (x_ai - theta_i)))
16        sigmoid_ai_r = 1 / (1 + bp.ops.exp(slope_i * theta_i))
17        sigmoid_ai = sigmoid_ai_l - sigmoid_ai_r
18        daidt = (-a_i + (k_i - r_i * a_i) * sigmoid_ai) / tau_i
19
20        return daedt, daidt  $\tau_i \frac{da_i}{dt} = -a_i + (k_i - r_i * a_i) * S_i(c_1 a_e - c_2 a_i + I_{ext_i})$ 

```

1.1 生物背景

```
21     def __init__(self, size, c1=12., c2=4., c3=13., c4=11.,
22                  k_e=1., k_i=1., tau_e=1., tau_i=1., r_e=1., r_i=1.,
23                  slope_e=1.2, slope_i=1., theta_e=2.8, theta_i=4.,
24                  **kwargs):
25         # params
26         self.c1 = c1
27         self.c2 = c2
28         self.c3 = c3
29         self.c4 = c4
30         self.k_e = k_e
31         self.k_i = k_i
32         self.tau_e = tau_e
33         self.tau_i = tau_i
34         self.r_e = r_e
35         self.r_i = r_i
36         self.slope_e = slope_e
37         self.slope_i = slope_i
38         self.theta_e = theta_e
39         self.theta_i = theta_i
40
41     # vars
42     self.input_e = bp.backend.zeros(size)
43     self.input_i = bp.backend.zeros(size)
44     self.a_e = bp.backend.ones(size) * 0.1
45     self.a_i = bp.backend.ones(size) * 0.05
46
47     self.integral = bp.odeint(self.derivative) → 调用'bp.odeint'积分常微分方程。
48     super(FiringRateUnit, self).__init__(size=size, **kwargs) → 参数'method'置为默认值'euler'。
49
50     def update(self, _t):
51         self.a_e, self.a_i = self.integral(
52             self.a_e, self.a_i, _t,
53             self.k_e, self.r_e, self.c1, self.c2,
54             self.input_e, self.slope_e,
55             self.theta_e, self.tau_e,
56             self.k_i, self.r_i, self.c3, self.c4,
57             self.input_i, self.slope_i,
58             self.theta_i, self.tau_i)
59         self.input_e[:] = 0. → 数值积分方法采用
60         self.input_i[:] = 0. → 向量形式更新变量。
```

模型参数保存为浮点数。

模型变量保存为浮点数的向量。

调用`bp.odeint`积分常微分方程。

参数`method`置为默认值`euler`。

传参`size`和`**kwargs`给超类`bp.NeuGroup`的构造函数。

重置当前时刻的外部输入。

2. 突触模型

当我们建模了神经元模型后，我们需要建立突触模型来描述神经元之间的信息传递过程。突触把神经元连接起来，使不同神经元得以沟通，对于组成神经网络也是至关重要的。

本章将在[2.1 突触模型](#)中介绍包括化学突触与电突触的模型，其中化学突触包括常见的AMPA模型、NMDA模型等，以及更加抽象的、简化的Alpha模型、单指数衰减模型等。

突触模型另一个重要的方面在于突触可塑性的实现，突触可塑性对学习、记忆与神经网络的计算、训练非常重要，本章将会在[2.2 可塑性模型](#)中介绍突触可塑性的部分，包括突触短时程可塑性（STP）及突触长时程可塑性等。

注：本章所述模型的完整BrainPy代码请见[附录](#)，或右键点此下载jupyter notebook版本。

2.1 突触模型

2.2 可塑性模型

2.1 突触模型

我们在前面的章节中已经学习了如何建模神经元，那么神经元之间是怎么连接起来的呢？神经元的动作电位是如何在不同神经元之间传导的呢？这里，我们将介绍如何用BrainPy来模拟神经元之间的沟通。

注：本章所述模型的完整BrainPy代码请见[附录](#)，或右键点此下载jupyter notebook版本。

2.1.1 化学突触

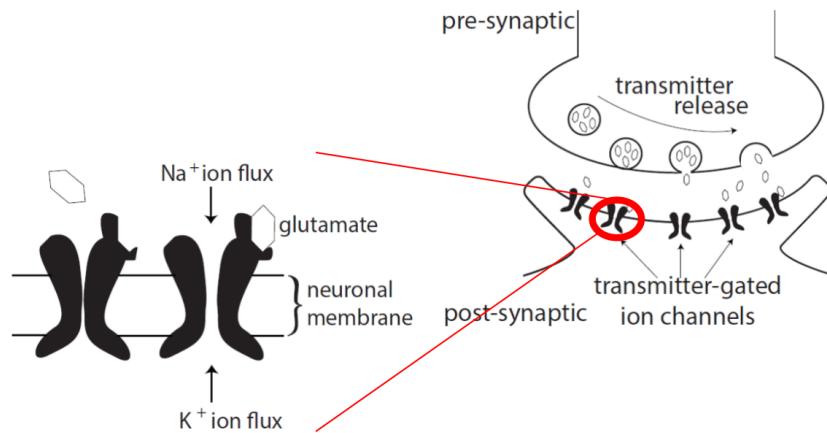
生物背景

图2-1描述了神经元之间信息传递的生物过程。当突触前神经元的动作电位传递到轴突的末端（terminal）时，突触前神经元会释放**神经递质**（又称递质）。神经递质会和突触后神经元上的受体结合，从而引起突触后神经元膜电位的改变，这种改变称为突触后电位（PSP）。根据神经递质种类的不同，突触后电位可以是兴奋性的或抑制性的。例如**谷氨酸**（Glutamate）就是一种重要的兴奋性神经递质，而**GABA**则是一种重要的抑制性神经递质。

神经递质与受体的结合可能会导致离子通道的打开（**离子型受体**）或改变化学反应的过程（**代谢型受体**）。

在本节中，我们将介绍如何使用BrainPy来实现一些常见的突触模型，主要有：

- **AMPA**和**NMDA**：它们都是谷氨酸的离子型受体，被结合后都可以直接打开离子通道。但是NMDA通常会被镁离子（Mg²⁺）堵住，无法对谷氨酸做出反应。由于镁离子对电压敏感，当突触后电位超过镁离子的阈值以后，镁离子就会离开NMDA通道，让NMDA可以对谷氨酸做出反应。因此，NMDA的反应是比较慢的。
- **GABA_A**和**GABA_B**：它们是GABA的两类受体，其中GABA_A是离子型受体，通常可以产生快速的抑制性电位；而GABA_B则为代谢型受体，通常会产生缓慢的抑制性电位。

图 2-1 生物突触 (引自 Gerstner et al., 2014 ¹)

为了简便地建模从神经递质释放到突触后神经元膜电位改变的过程，我们可以使用门控变量 s 来描述每当突触前神经元产生动作电位时，有多少比例的离子通道会被打开。让我们从AMPA的例子开始，看看如何建立突触模型并用BrainPy实现。

AMPA模型

如前所述，AMPA（α-氨基-3-羟基-5-甲基-4-异恶唑丙酸）受体是一种离子型受体，也就是说，当它被神经递质结合后会立即打开离子通道，从而引起突触后神经元膜电位的变化。

我们可以用马尔可夫过程来描述离子通道的开关。如图2-2所示， s 代表通道打开的概率， $1 - s$ 代表离子通道关闭的概率， α 和 β 是转移概率 (transition probability)。由于神经递质能让离子通道打开，所以从 $1 - s$ 到 s 的转移概率受神经递质浓度 (以 $[T]$ 表示) 影响。

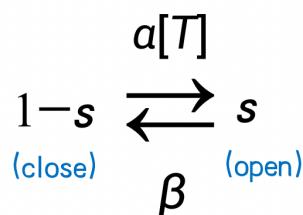


图2-2 离子通道动力学的马尔可夫过程

把该过程用微分方程描述，得到以下式子。

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

其中， $\alpha[T]$ 表示从状态 $(1 - s)$ 到状态 (s) 的转移概率，即激活速率； β 表示从 s 到 $(1 - s)$ 的转移概率，即失活速率。

下面我们来看看如何用BrainPy去实现这样一个模型。首先，我们要定义一个类，因为突触是连接两个神经元的，所以这个类继承自 `bp.TwoEndConn`。在这个类中，和神经元模型一样，我们用一个 `derivative` 函数来实现上述微分方程，并在后面的 `__init__` 函数中初始化这个函数，指定用 `bp.odeint` 来解这个方程，并指定数值积分方法。由于这微分方程是线性的，我们选用 `exponential_euler` 方法。

```

1  import brainpy as bp
2
3
4  class AMPA(bp.TwoEndConn):  突触连接两群神经元，因此继承bp.TwoEndConn
5      target_backend = ['numpy', 'numba']
6
7      @staticmethod
8      def derivative(s, t, TT, alpha, beta):  }  $\frac{ds}{dt} = \alpha[T](1-s) - \beta s$ 
9          ds = alpha * TT * (1 - s) - beta * s
10         return ds
11
12     def __init__(self, pre, post, conn, alpha=0.98, beta=0.18, T=0.5,
13                  T_duration=0.5, **kwargs):
14         # parameters
15         self.alpha = alpha
16         self.beta = beta
17         self.T = T  } 把[T]简化为一个常数T值，持续T_duration这么
18         self.T_duration = T_duration  } 长的时间
19
20         # connections
21         self.conn = conn(pre.size, post.size)
22         self.pre_ids, self.post_ids = conn.requires('pre_ids', 'post_ids')
23         self.size = len(self.pre_ids)  }
24
25         # variables
26         self.s = bp.ops.zeros(self.size)
27         self.t_last_pre_spike = -1e7 * bp.ops.ones(self.size)
28
29         self.int_s = bp.odeint(f=self.derivative, method='exponential_euler')
30         super(AMPA, self).__init__(pre=pre, post=post, **kwargs)
31

```

这边突触前和突触后连接的神经元群均为向量，突触也是向量形式，因此要指定每个突触所连接的突触前和后神经元的id

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

然后我们在 `update` 函数中更新 s 。

```

32     def update(self, _t):
33         for i in range(self.size):  对每一个突触，i为当前处理的突触的id
34             pre_id = self.pre_ids[i]
35             post_id = self.post_ids[i]
36
37             if self.pre.spike[pre_id]:
38                 self.t_last_pre_spike[pre_id] = _t
39                 TT = (_t - self.t_last_pre_spike[pre_id])  } TT表示[T]，如果前神经元在
40                 < self.T_duration) * self.T  } duration时间内曾有发放，则
41             self.s[i] = self.int_s(self.s[i], _t, TT, self.alpha, self.beta)
42

```

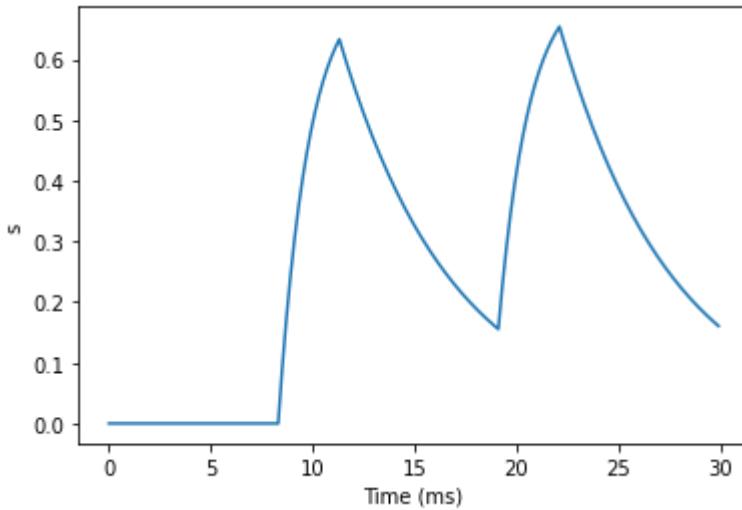
设TT=T，否则设TT=0。

我们已经定义好了一个AMPA类，现在可以画出 s 随时间变化的图了。我们首先写一个 `run_syn` 函数来方便之后运行更多的突触模型，然后把 AMPA类和需要自定义的变量传入这个函数来运行并画图。

1.1 生物背景

```
43 import brainmodels as bm
44
45 bp.backend.set(backend='numba', dt=0.1)      ----- 指定使用numba作为
46                                         backend
47
48
49 def run_syn(syn_model, **kwargs):           ----- 定义一个函数来运行突触模型
50     neu1 = bm.neurons.LIF(2, monitors=['V'])
51     neu2 = bm.neurons.LIF(3, monitors=['V'])    } 从brainmodels包中获取LIF模型来
52                                         作为被连接的两群神经元
53
54     syn = syn_model(pre=neu1, post=neu2, conn=bp.connect.All2All(),
55                      monitors=['s'], **kwargs)          ----- 指定突触前和突触后神经元，以及
56                                         他们的连接方式。这里我们使用
57                                         BrainPy提供的All2All方法（全连
58                                         接）。
59
60     net = bp.Network(neu1, syn, neu2)
61     net.run(30., inputs=(neu1, 'input', 35.))
62     bp.visualize.line_plot(net.ts, syn.mon.s, ylabel='s', show=True)
63
64
65 run_syn(AMPA, T_duration=3.)      ----- 运行AMPA模型，设置
66                                         参数T_duration为3。
```

运行以上代码，即可看到以下的结果：



由上图可以看出，当突触前神经元产生一个动作电位， s 的值会先增加，然后衰减。

NMDA模型

如前所述，NMDA受体一开始被镁离子堵住，而当膜电位达到一定阈值后，镁离子则会移开。我们用 c_{Mg} 表示镁离子的浓度，它对突触后膜的电导 g 的影响可以由以下公式描述：

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

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

其中 g_∞ 代表了镁离子浓度的作用，其值随着镁离子浓度增加而减小；而随着电压 V 增加， g_∞ 较不受 c_{Mg} 影响。 α, β 和 \bar{g} 是一些常数。门控变量 s 和AMPA模型类似，其动力学由以下公式给出：

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

1.1 生物背景

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

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

其中， τ_{decay} 和 τ_{rise} 分别为 s 衰减及上升的时间常数， a 是参数。

接下来我们用 BrainPy 来实现 NMDA 模型，代码如下。

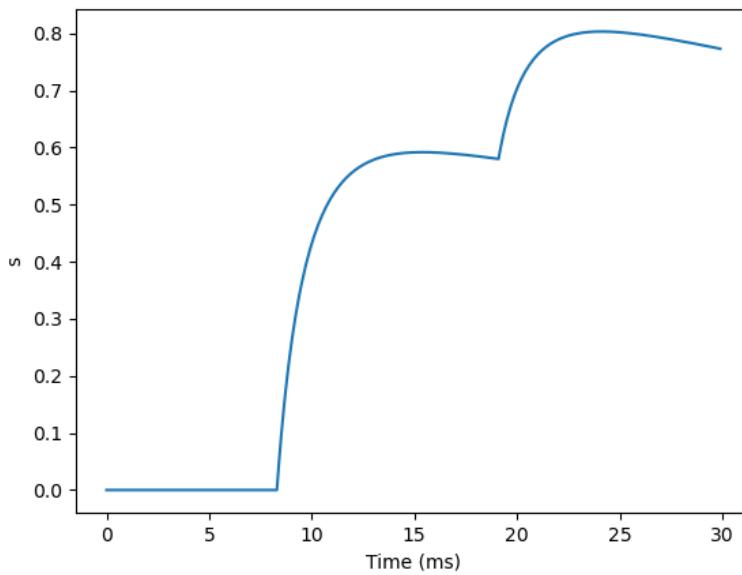
```

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

```

由于我们在实现 AMPA 模型时已经定义了 `run_syn` 函数，在这里我们可以直接调用：

run_syn(NMDA)



由图可以看出，NMDA的衰减过程非常缓慢，第一个突触前神经元的动作电位引起的 s 增加后还没怎么衰减，第二个的值就加上去了，由于我们这里只跑了30ms的模拟，还看不到NMDA衰退的过程。

GABA_B模型

GABA_B是一种代谢型受体，神经递质和受体结合后不会直接打开离子通道，而是通过G蛋白作为第二信使来起作用。因此，这里我们用 $[R]$ 表示多少比例的受体被激活，并用 $[G]$ 表示激活的G蛋白的浓度， s 由 $[G]$ 调节，公式如下：

$$\begin{aligned}\frac{d[R]}{dt} &= k_3[T](1 - [R]) - k_4[R] \\ \frac{d[G]}{dt} &= k_1[R] - k_2[G] \\ s &= \frac{[G]^4}{[G]^4 + K_d}\end{aligned}$$

$[R]$ 的动力学类似于AMPA模型中的 s ，受神经递质浓度 $[T]$ 影响， k_3, k_4 表示转移概率。 $[G]$ 的动力学受 $[R]$ 影响，并由参数 k_1, k_2 控制。 K_d 为一个常数。

用BrainPy实现的代码如下。

1.1 生物背景

```

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

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

```

} TT表示[T],如果前神经元在T_duration时间内曾有发放,则设TT=T,否则设TT=0.

} I = g(V - E)

由于GABA_B的动力学和NMDA一样,也是非常缓慢的,这里我们不再用前面写的只有30ms模拟的 run_syn 函数,而是通过调用BrainPy提供的 bp.inputs.constant_current 方法,先给20ms的输入,接着看剩余1000ms在没有外界输入情况下的衰减。

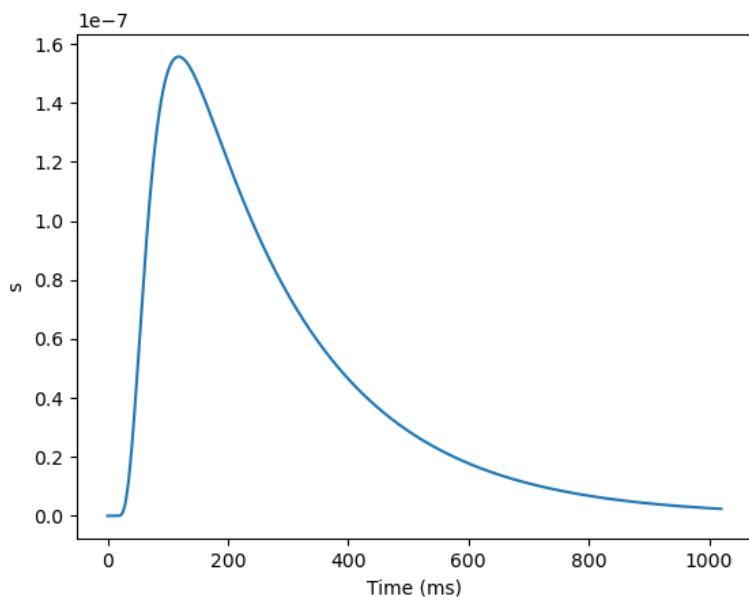
```

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

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

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

```



结果显示，GABA_B的衰减持续数百毫秒。

基于电流与基于电导的模型

细心的读者应该发现，我们刚才对GABA_B门控变量 s 的建模并没有显示出其引起抑制性电位的特点。要体现出兴奋性和抑制性，我们不仅需要建模门控变量 s ，还需要建模通过突触的电流 I （作为突触后神经元的输入）。根据突触电流是否受突触后神经元膜电位的影响不同，分为**基于电流 (current-based)** 与**基于电导 (conductance-based)** 的两种模型。

(1) 基于电流 (Current-based) 的模型

基于电流的模型公式如下：

$$I \propto s$$

在代码实现上，我们通常会乘上一个权重 w 。我们可以通过调整权重 w 的正负值来实现兴奋性和抑制性突触。另外，我们通过使用BrainPy提供的 `register_constant_delay` 函数给变量 `I_syn` 加上延迟时间来实现突触的延迟。

```

1  def __init__(self, pre, post, conn, delay, **kwargs):
2      # ...
3      self.s = bp.ops.zeros(self.size)
4      self.w = bp.ops.ones(self.size) * .2
5      self.I_syn = self.register_constant_delay('I_syn', size=self.size,
6                                                 delay_time=delay)    ————— 使用BrainPy提供的
7                                                 register_constant_delay方
8                                                 法，在输出突触的改变前
9      def update(self, _t):
10         for i in range(self.size):
11             # ...
12             self.I_syn.push(i, self.w[i] * self.s[i])
13             self.post.input[post_id] += self.I_syn.pull(i)    ————— 此时注册了delay的I_syn拥有push和pull的方法。
14                                         可以通过先push后pull来实现延迟。
15                                         延迟的时间点设在把突触电流应用到突触后神经元的输入前。

```

(2) 基于电导 (Conductance-based) 的模型

在基于电导的模型中，电导为 $g = \bar{g}s$ 。因此，根据欧姆定律得公式如下：

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

这里 E 是一个反转电位 (reverse potential)，它可以决定 I 的方向是抑制还是兴奋。例如，当静息电位约为-65mV时，减去比它更低的 E ，例如-75mV，将变为正，从而改变公式中电流的方向并产生抑制电流。兴奋性突触的 E 一般为比较高的值，如0mV。

代码实现上，可以把延迟时间应用到变量 `g` 上。

```

1  def __init__(self, pre, post, conn, g_max, E, delay, **kwargs):
2      self.g_max = g_max
3      self.E = E
4      # ...
5      self.s = bp.ops.zeros(self.size)
6      self.g = self.register_constant_delay('g', size=self.size, delay_time=delay)    ————— 这里把delay的时间
7                                         注册到电导变量中。
8
9      def update(self, _t):
10         for i in range(self.size):
11             # ...
12             self.g.push(i, self.g_max * self.s[i])    —————  $g = \bar{g}s$ 
13             self.post.input[post_id] -= self.g.pull(i) * (self.post.V[post_id] - self.E)    —————  $I = g(V - E)$ 
14

```

现在可以回顾一下我们刚才实现的NMDA模型和GABA_B模型，它们都是基于电导的模型，在NMDA模型中， $E = 0mV$ ，因此产生兴奋性电流；而在GABA_B模型中， $E = -95mV$ ，产生抑制性电流。

抽象的简化模型

前面我们建模了几种经典的化学突触模型，它们的门控变量 s 的动力学都有着先上升后下降的特征。当我们不需要具体地建模某种生物学突触时，只要把握了突触的基本动力学特征（先上升后下降）即可。这里，我们会介绍四种抽象的简化模型及其在BrainPy上的实现，这些抽象模型既可以是基于电流的，也可以是基于电导的模型，可以根据需要选择。

(1) 双指数差 (Differences of two exponentials)

我们首先来看**双指数差** (Differences of two exponentials) 模型，它有两个指数项相减，公式如下：

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

其中 t_s 表示突触前神经元产生动作电位的时间， τ_1 和 τ_2 为时间常数。

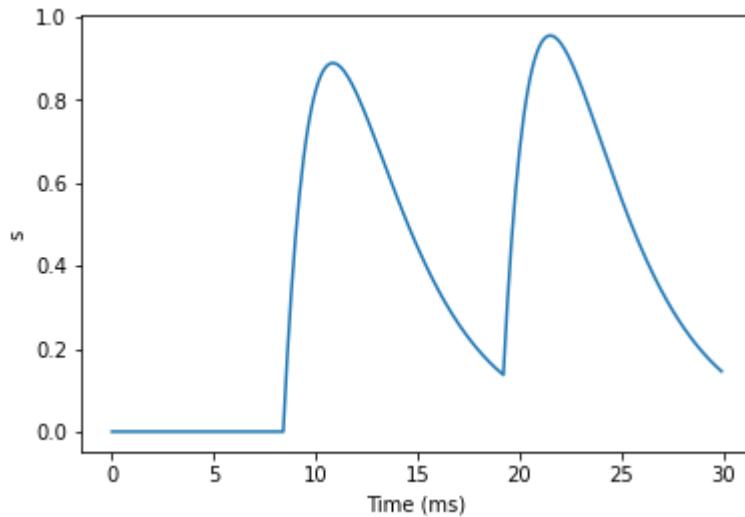
在BrainPy的实现中，我们采用以下微分方程形式：

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

这里我们用 `update` 函数来控制 x 增加的逻辑。代码如下：

```

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



(2) Alpha突触

Alpha突触的动力学由以下公式给出：

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

和双指数差模型类似， t_s 表示突触前神经元产生动作电位的时间，不同的是这里只有一个时间常数 τ 。微分方程形式如下：

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

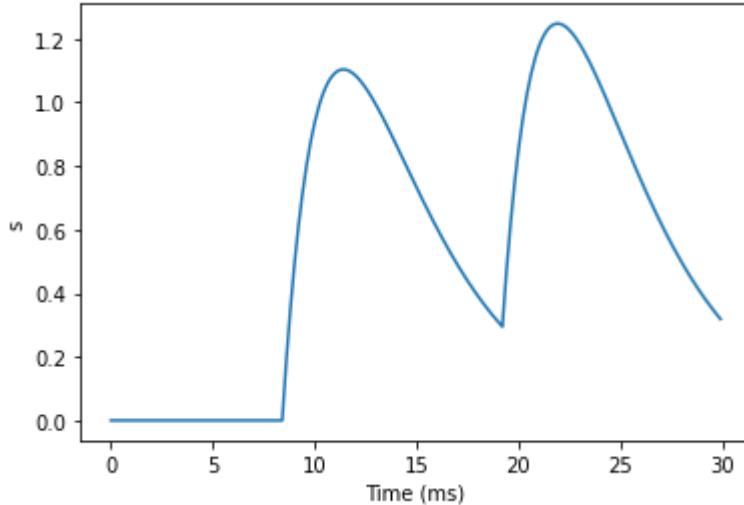
$$\frac{dx}{dt} = -\frac{2x}{\tau} - \frac{s}{\tau^2}$$

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

可以看出alpha模型和双指数差模型其实很相似，相当于 $\tau = \tau_1 = \tau_2$ 。因此，代码实现上也很接近：

1.1 生物背景

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



(3) 单指数衰减 (Single exponential decay)

下面我们来介绍一种更加简化的模型，它忽略了上升的过程，而只建模了衰减 (decay) 的过程。单指数衰减 (Single exponential decay) 模型用一个指数项来描述衰减的过程，公式如下：

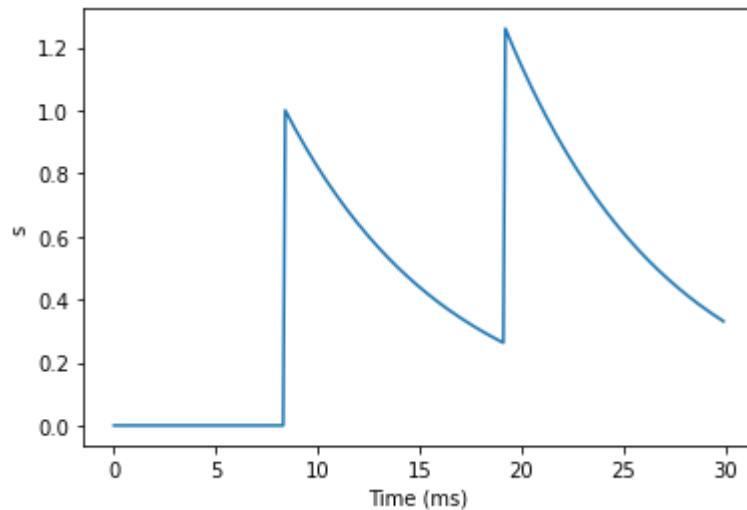
$$\frac{ds}{dt} = -\frac{s}{\tau_{decay}}$$

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

代码实现如下：

1.1 生物背景

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



(4) 电压跳变 (Voltage jump)

电压跳变 (Voltage jump) 模型比单指数衰减模型还要更加简化，它连衰退的过程也忽略了，公式如下：

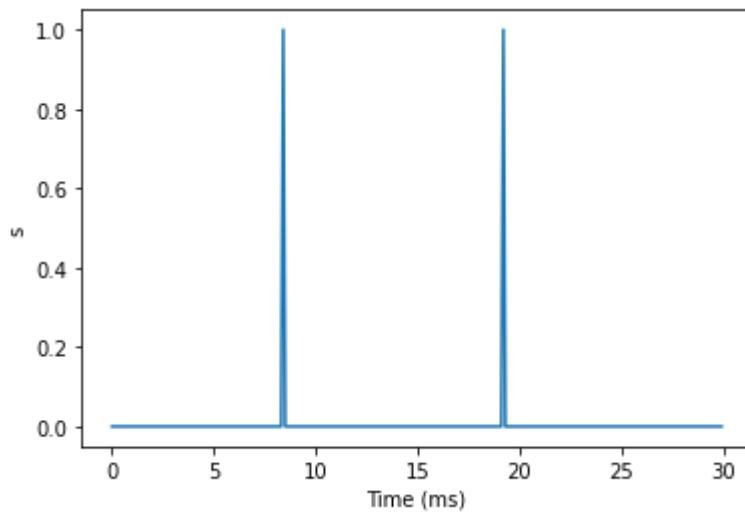
if (fire), then $s \leftarrow s + 1$

在实现上，只需要在 `update` 函数中更新 s 即可。代码如下：

```

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

```



2.1.2 电突触

除了前面介绍的化学突触以外，电突触在我们神经系统中也很常见。

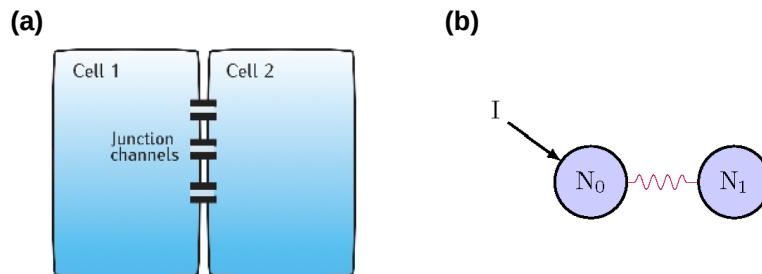


图2-3 (a) 神经元间的缝隙连接. (b) 等效模型.
(引自 *Sterratt et al., 2011* ²)

如图2-3a所示，两个神经元通过连接通道（junction channels）相连，可以直接导电，这种连接又称为缝隙连接（gap junction）。因此，可以看作是两个神经元由一个常数电阻连起来，如图2-3b所示。

1.1 生物背景

根据欧姆定律可得以下公式：

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

这里 V_0 和 V_1 分别为两个神经元的膜电位，突触权重 w 表示常数电导。

在BrainPy的实现中，只需要在 update 函数里更新即可。

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

定义好了缝隙连接的类以后，我们跑模拟来看给0号神经元输入时，1号神经元的电位变化。我们首先实例化两个LIF神经元模型，并用缝隙连接把它们连接起来。然后仅给0号神经元 neu0 一个恒定的电流，neu1 没有外界输入。

1.1 生物背景

```
import matplotlib.pyplot as plt

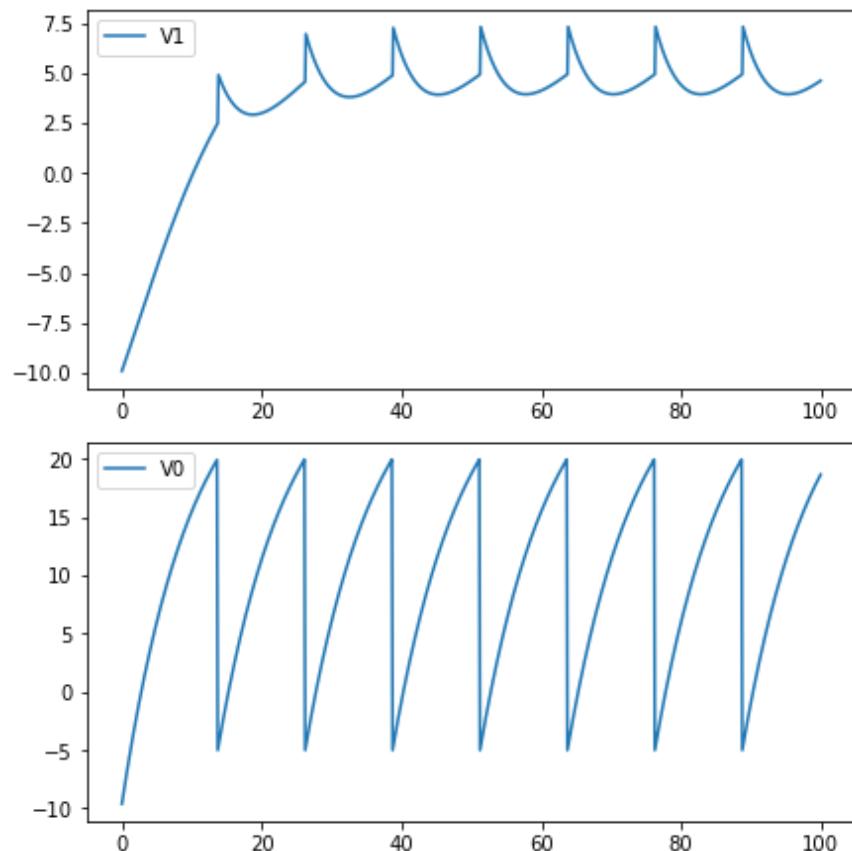
neu0 = bm.neurons.LIF(1, monitors=['V'], t_refractory=0)
neu0.V = bp.ops.ones(neu0.V.shape) * -10.
neu1 = bm.neurons.LIF(1, monitors=['V'], t_refractory=0)
neu1.V = bp.ops.ones(neu1.V.shape) * -10.
syn = Gap_junction(pre=neu0, post=neu1, conn=bp.connect.All
                     k_spikelet=5.)
syn.w =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

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

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

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

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



1.1 生物背景

结果图中，下图 V_0 表示0号神经元的膜电位变化，而上图 V_1 为1号神经元的膜电位。0号神经元因为有电流输入而有持续的发放，并给1号神经元输入，导致 V_1 产生阈值下的改变。

参考资料

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

2.2 突触可塑性

在前一节中，我们讨论了突触动力学，但还没有涉及到突触可塑性。接下来我们将在本节中介绍如何使用BrainPy来实现突触可塑性。

突触可塑性指的是突触强度（synaptic efficacy）或突触权重（synaptic weight）的变化，主要分为短时程可塑性（short-term plasticity）与长时程可塑性（long-term plasticity）。我们将首先介绍突触短时程可塑性，然后介绍几种不同的突触长时程可塑性模型。

注：本章所述模型的完整BrainPy代码请见[附录](#)，或右键点此下载jupyter notebook版本。

2.2.1 突触短时程可塑性（STP）

我们首先从实验结果来介绍突触短时程可塑性。在图2-4中，上图表示突触前神经元的动作电位，下图为突触后神经元的膜电位。我们可以看到，当突触前神经元在短时间内持续发放的时候，突触后神经元的反应越来越弱，呈现出短时程抑制（short term depression）效果。而当突触前神经元停止发放几百毫秒后，再来一个动作电位，此时突触后神经元的反应基本恢复到一开始的状态，因此这个抑制效果持续的时间很短，称为短时程可塑性。

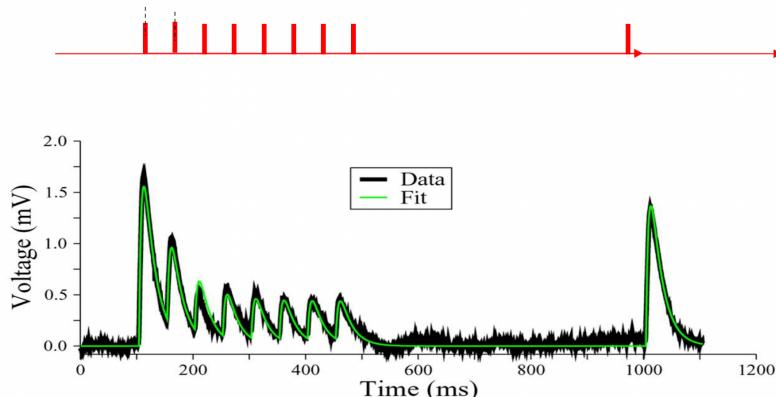


图2-4 突触短时程可塑性 (改编自 Gerstner et al., 2014¹)

那么接下来就让我们来看看描述短时程可塑性的计算模型²。短时程可塑性主要由神经递质释放的概率 u 和神经递质的剩余量 x 两个变量来描述。整体的动力学方程如下：

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

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

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

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

其中，突触电流 I 的动力学可以采用上一节介绍的任意一种 s 的动力学模型，这里我们采用简单、常用的单指数衰减 (single exponential decay) 模型来描述。 U 和 A 分别为 u 和 I 的增量，而 τ_f 和 τ_d 则分别为 u 和 x 的时间常数。

在该模型中， u 主要贡献了短时程易化 (Short-term facilitation; STF) 的效果，它的初始值为 0，并随着突触前神经元的每次发放而增加；而 x 则主要贡献短时程抑制 (Short-term depression; STD) 效果，它的初始值为 1，并在每次突触前神经元发放时都会被用掉一些（即减少）。易化和抑制两个方向的效果是同时发生的，因此 τ_f 和 τ_d 的大小关系决定了可塑性的哪个方向的变化起主要作用。

用 BrainPy 实现的代码如下，由于突触可塑性也是发生在突触上的，这里和突触模型一样，继承自 `bp.TwoEndConn`。

```

9  class STP(bp.TwoEndConn):
10     target_backend = ['numpy', 'numba']
11
12     @staticmethod
13     def derivative(s, u, x, t, tau, tau_d, tau_f):
14         dsdt = -s / tau
15         dudt = -u / tau_f
16         dxdt = (1 - x) / tau_d
17         return dsdt, dudt, dxdt
18
19     def __init__(self, pre, post, conn, delay=0., U=0.15, tau_f=1500.,
20                  tau_d=200., tau_u=8., **kwargs):
21         # parameters
22         self.tau_d = tau_d
23         self.tau_f = tau_f
24         self.tau = tau
25         self.U = U
26         self.delay = delay
27
28         # connections
29         self.conn = conn(pre.size, post.size)
30         self.pre_ids, self.post_ids = conn.requires('pre_ids', 'post_ids')
31         self.size = len(self.pre_ids)
32
33         # variables
34         self.s = bp.ops.zeros(self.size)
35         self.x = bp.ops.ones(self.size)
36         self.u = bp.ops.zeros(self.size)
37         self.w = bp.ops.ones(self.size)
38         self.I_syn = self.register_constant_delay('I_syn', size=self.size,
39                                               delay_time=delay)
40
41         self.integral = bp.odeint(f=self.derivative, method='exponential_euler')
42
43     super(STP, self).__init__(pre=pre, post=post, **kwargs)
44

```

1.1 生物背景

```
45     def update(self, _t):
46         for i in range(self.size):
47             pre_id = self.pre_ids[i]
48
49             self.s[i], u, x = self.integral(self.s[i], self.u[i], self.x[i], _t,
50                                             self.tau, self.tau_d, self.tau_f)
51
52             if self.pre.spike[pre_id] > 0:
53                 u += self.U * (1 - self.u[i])
54                 self.s[i] += self.w[i] * u * self.x[i]
55                 x -= u * self.x[i]
56                 self.u[i] = u
57                 self.x[i] = x
58
59             # output
60             post_id = self.post_ids[i]
61             self.I.syn.push(i, self.s[i])
62             self.post.input[post_id] += self.I.syn.pull(i)
63
```

定义好STP的类以后，接下来让我们来定义跑模拟的函数。跟突触模型一样，我们需要实例化两个神经元群并把它们连接在一起。结果画图方面，除了 s 的动力学以外，我们也希望看到 u 和 x 随时间的变化，因此我们制定 `monitors=['s', 'u', 'x']`。

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

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

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

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

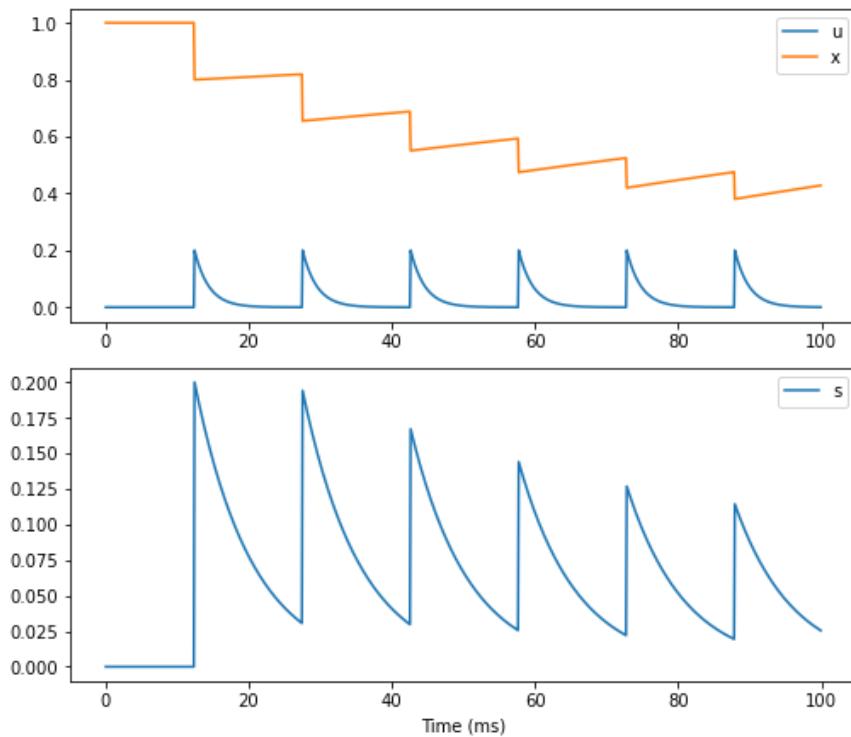
    fig.add_subplot(gs[1, 0])
    plt.plot(net.ts, syn.mon.s[:, 0], label='s')
    plt.legend()

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

接下来，我们设 `tau_d > tau_f`，让我们来看看结果。

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

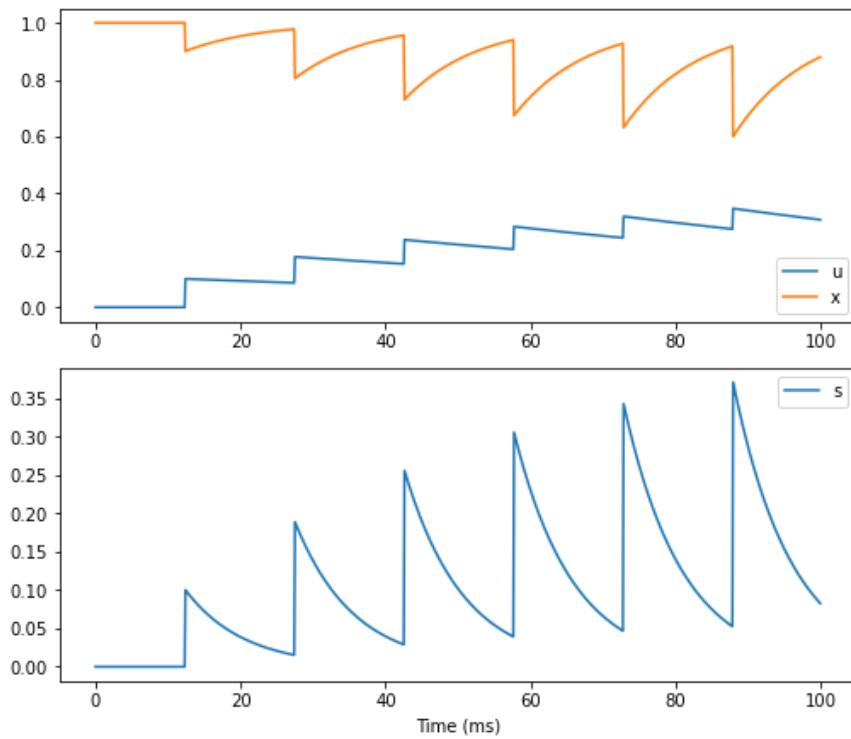
1.1 生物背景



从结果图中，我们可以看出当设置 $\tau_d > \tau_f$ 时， x 每次用掉以后恢复得很慢，而 u 每次增加后很快又衰减下去了，因此从 s 随时间变化的图中我们可以看到STD效果为主。

接下来看看当我们设置 `tau_f > tau_d` 时的结果。

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

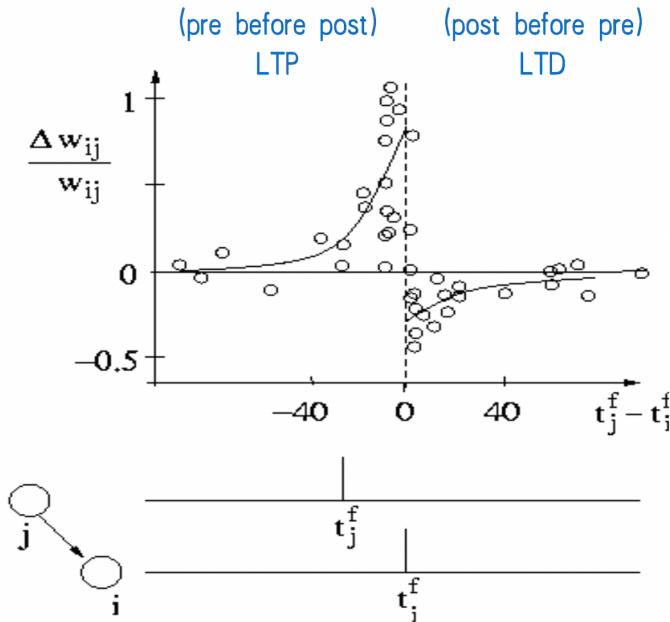


结果图显示，当 $\tau_f > \tau_d$ 时， x 每次用掉后很快又补充回去了，这表示突触前神经元总是有足够的神经递质可用。同时， u 的衰减非常缓慢，即释放神经递质的概率越来越高，从 s 的动力学可以看出STF效果占主要地位。

2.2.2 突触长时程可塑性

脉冲时间依赖可塑性 (STDP)

图2-5显示了实验上观察到的脉冲时间依赖可塑性 (spiking timing dependent plasticity; STDP) 的现象。x轴为突触前神经元和突触后神经元产生脉冲 (spike) 的时间差，位于零点左侧的数据点为突触前神经元先于突触后神经元发放的情况，由图可见此时突触权重的改变量为正，表现出长时程增强 (long term potentiation; LTP) 的现象；而零点右侧则是突触后神经元比突触前神经元更先发放的情况，表现出长时程抑制 (long term depression; LTD)。

图2-5 脉冲时间依赖可塑性 (改编自 Bi & Poo, 2001³)

STDP的计算模型如下：

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

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

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

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

其中 w 为突触权重， s 与上一节讨论的一样为门控变量。与STP模型类似，这里由 A_s 和 A_t 两个变量分别控制LTD和LTP。 ΔA_s 和 ΔA_t 分别为 A_s 和 A_t 的增量，而 τ_s 和 τ_t 则分别为它们的时间常数。

根据这个模型，当突触前神经元先于突触后神经元发放时，在突触后神经元发放之前，每当突触前神经元有一个脉冲， A_s 便增加，而由于此时突触后神经元没有脉冲，因此 A_t 保持在初始值0， w 暂时不会有变化。直到突触后神经元发放时， w 的增量将会是 $A_s - A_t$ ，由于 $A_s > A_t$ ，此时会表现出长时程增强（LTP）。反之亦然。

现在让我们看看如何使用BrainPy来实现这个模型。其中 s 动力学的实现部分，我们跟STP模型一样采用单指数衰减模型。

1.1 生物背景

```

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

```

我们通过给予突触前和突触后的两群神经元不同的电流输入来控制它们产生脉冲的时间。首先我们在 $t = 5ms$ 时刻给突触前神经元第一段电流（每一段强度为 $30 \mu A$ ，并持续 $15ms$ ，保证LIF模型会产生一个脉冲），然后在 $t = 10ms$ 才给突触后神经元一个输入。每段输入之间间隔 $15ms$ 。以此在前三对脉冲中保持 $t_{post} = t_{pre} + 5\circ$ 。接下来我们设置一个较长的间隔，然后把刺激顺序调整为 $t_{post} = t_{pre} - 3\circ$

1.1 生物背景

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

接下来跑模拟的代码和STP类似，这里我们画出突触前后神经元的脉冲时间以及 s 和 w 随时间的变化。

```

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

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

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

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

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

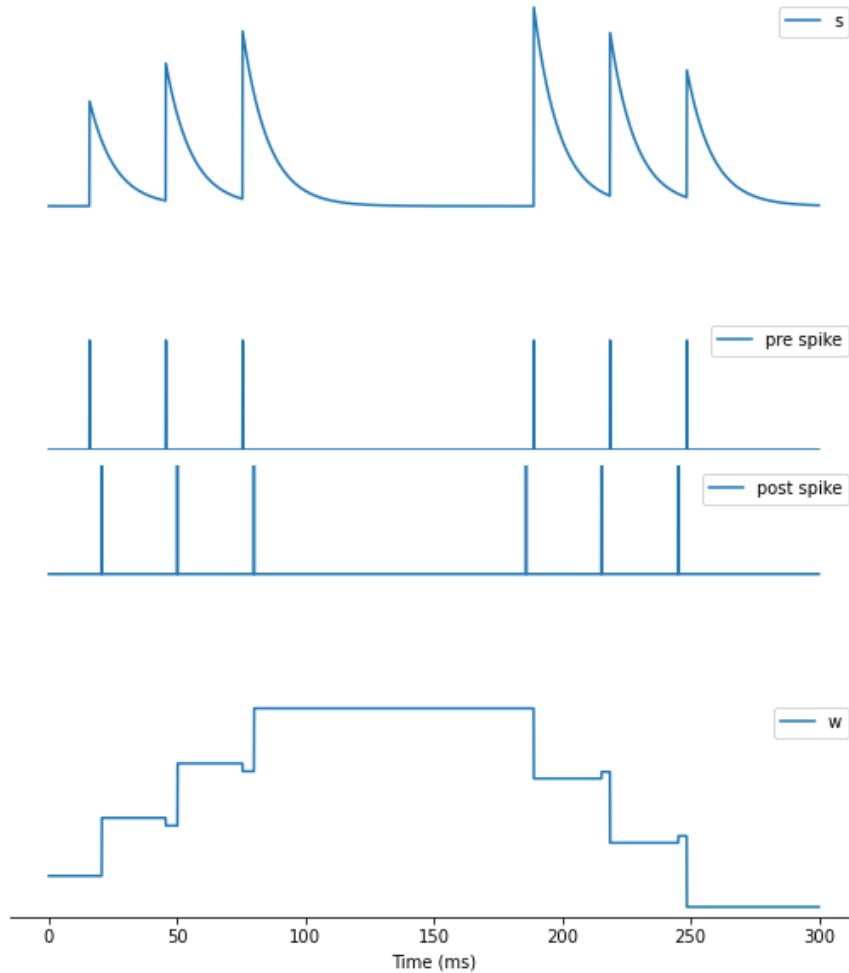
ax1=fig.add_subplot(gs[1, 0])
plt.plot(net.ts, pre.mon.spike[:, 0], label="pre spike")
plt.ylim(0, 2)
hide_spines(ax1)
plt.legend(loc = 'center right')

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

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

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

```



结果正如我们所预期的，在150ms前，突触前神经元的脉冲时间在突触后神经元之前， w 增加，呈现LTP。而150ms后，突触后神经元先于突触前神经元发放， w 减少，呈现LTD。

Oja法则

接下来我们看基于赫布学习律 (Hebbian learning) 的发放率模型 (firing rate model)。赫布学习律认为相互连接的两个神经元在经历同步的放电活动后，它们之间的突触连接就会得到增强。由于赫布学习律仅关心两个神经元的同步发放，而不关心发放的次序，可以用忽略具体的脉冲时间的发放率模型来实现。我们首先看赫布学习律的一般形式，对于神经元 j 到神经元 i 的连接，用 r_j 和 r_i 分别表示前神经元组和后神经元组的发放率，根据赫布学习律的局部性 (locality) 特性， w_{ij} 的变化受 w 本身及 r_j, r_i 的影响，得以下微分方程：

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

把上式右边经过泰勒展开可得下式：

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

赫布学习律的关键在于第六项，只有当第六项的系数 c_{11} 非0才满足赫布学习律的同步发放。前面我们看了赫布学习律的一般形式，接下来我们看一个具体的例子。

Oja法则的公式如下，对应于上式第5、6项系数非零，其中 γ 为学习速率(learning rate)。

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

下面我们用BrainPy来实现Oja法则。

```

1  class Oja(bp.TwoEndConn):           ↗ 学习法则也是发生在两群神经元之间的,
2    target_backend = 'numpy'
3
4    @staticmethod
5    def derivative(w, t, gamma, r_pre, r_post):      ↗  $\frac{dw}{dt} = \gamma(r_i r_j - w r_i^2)$ 
6      dwdt = gamma * (r_post * r_pre - r_post * r_post * w)
7      return dwdt
8
9    def __init__(self, pre, post, conn, gamma=.005, w_max=1., w_min=0., **kwargs):
10      # params
11      self.gamma = gamma
12      self.w_max = w_max
13      self.w_min = w_min
14      # no delay in firing rate models
15
16      # conns
17      self.conn = conn(pre.size, post.size)
18      self.conn_mat = conn.requires('conn_mat')
19      self.size = bp.ops.shape(self.conn_mat)           ↗ 对学习法则来说, 以矩阵形式存储变量更佳, 此时
20
21      # data
22      self.w = bp.ops.ones(self.size) * 0.5           ↗ 我们需要一个conn_mat, 结构如图。
23
24      self.integral = bp.odeint(f=self.derivative)
25      super(Oja, self).__init__(pre=pre, post=post, **kwargs)
26
27    def update(self, _t):
28      w = self.conn_mat * self.w
29      self.post.r = np.sum(w.T * self.pre.r, axis=1)   ↗ 发放率模型中,  $r_i = \sum_j w_{ij} r_j$ 
30
31      # resize to matrix
32      dim = self.size
33      r_post = np.vstack((self.post.r,) * dim[0])       ↗ 突触权重为矩阵, 而突触前后神经元的r为
34      r_pre = np.vstack((self.pre.r,) * dim[1]).T        ↗ 向量, 因此我们要把r扩展为跟w相同大小
35
36      self.w = self.integral(w, _t, self.gamma, r_pre, r_post)  ↗ 的矩阵。
37

```

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

由于Oja法则是发放率模型，它需要突触前后神经元具有变量 r ，因此我们定义一个简单的发放率神经元模型来观察两组神经元的学习规则。

```

39  class neu(bp.NeuGroup):
40    target_backend = 'numpy'
41
42    def __init__(self, size, **kwargs):
43      self.r = bp.ops.zeros(size)           ↗ 我们需要拥有参数r的基于发放率的神
44      super(neu, self).__init__(size=size, **kwargs)  经元模型
45
46    def update(self, _t):
47      self.r = self.r           ↗ 由于我们不关心神经元的动力学, 因此用最简单的模型
48

```

我们打算实现如图2-6所示的连接。突触后神经元群 i （紫色）同时接受两群神经元 j_1 （蓝色）和 j_2 （红色）的输入。我们给 i 和 j_2 完全相同的刺激，而给 j_1 的刺激一开始跟 i 一致，但后来就不一致了。因此，根据赫布学习律，我们预期同步发放时，突触权重 w 会增加，当 j_1 不再与 i 同步发放时，则 w_{ij_1} 停止增加。

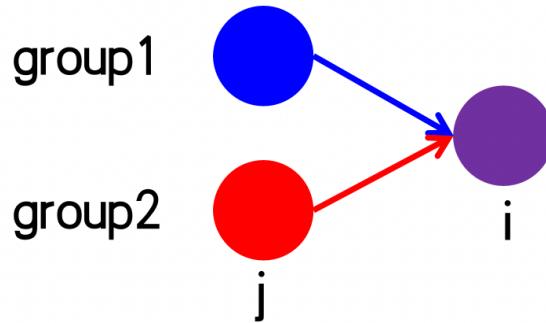


图2-6 神经元的连接

```

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

current1为给group1的输入
current2为给group2的输入
current3为给突触后神经元的输入（跟group2完全一致）

```

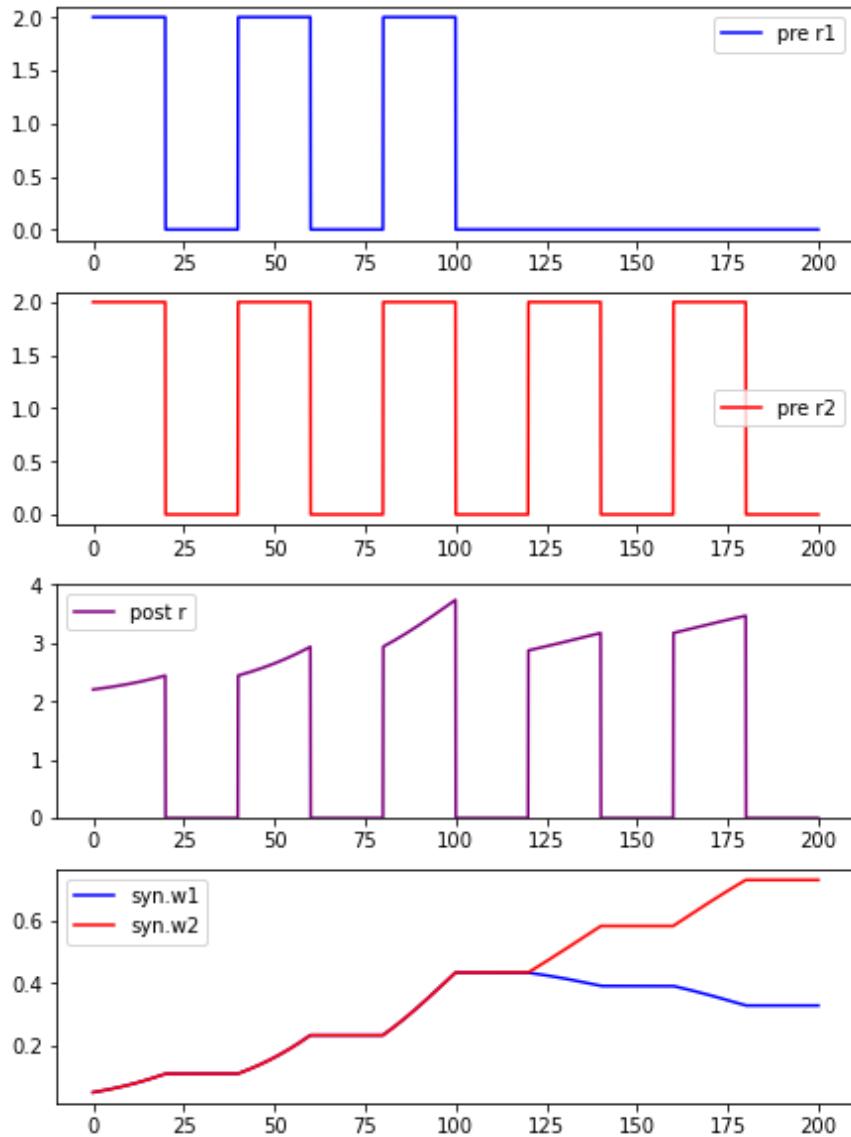
57
58 # simulate
59 neu_pre = neu(2, monitors=['r'])
60 neu_post = neu(2, monitors=['r'])
61 syn = Oja(pre=neu_pre, post=neu_post, conn=bp.connect.All2All(), monitors=['w'])
62 net = bp.Network(neu_pre, syn, neu_post)
63 net.run(duration=200., inputs=[(neu_pre, 'r', current_pre.T, '='),  

64                               (neu_post, 'r', current_post.T)])
65
```

运行的方式与其他突触模型相同。这里我们用列表的形式同时给突触前神经元与后神经元不同的输入。

```

66 # plot
67 fig, gs = bp.visualize.get_figure(4, 1, 2, 6)
68
69 fig.add_subplot(gs[0, 0])
70 plt.plot(net.ts, neu_pre.mon.r[:, 0], 'b', label='pre r1') —————> 通过索引[:, 0], 得到group 1的值
71 plt.legend()
72
73 fig.add_subplot(gs[1, 0])
74 plt.plot(net.ts, neu_pre.mon.r[:, 1], 'r', label='pre r2') —————> 通过索引[:, 1], 得到group 2的值
75 plt.legend()
76
77 fig.add_subplot(gs[2, 0])
78 plt.plot(net.ts, neu_post.mon.r[:, 0], color='purple', label='post r')
79 plt.ylim([0, 4])
80 plt.legend()
81
82 fig.add_subplot(gs[3, 0])
83 plt.plot(net.ts, syn.mon.w[:, 0, 0], 'b', label='syn.w1')
84 plt.plot(net.ts, syn.mon.w[:, 1, 0], 'r', label='syn.w2')
85 plt.legend()
86 plt.show()
```



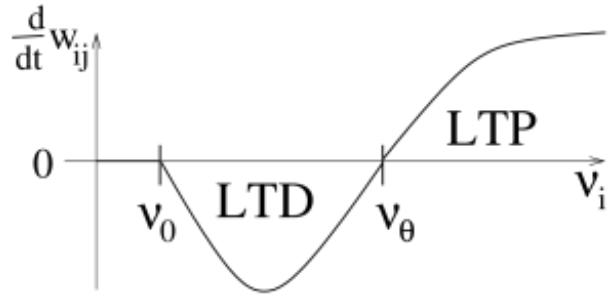
从结果可以看到，在前100ms内， j_1 和 j_2 均与*i*同步发放，他们对应的 w_1 和 w_2 也同步增加，显示出LTP。而100ms后， j_1 （蓝色）不再发放，只有 j_2 （红色）与*i*同步发放，因此 w_1 不再增加， w_2 则持续增加。该结果符合赫布学习律。

BCM法则

现在我们来看赫布学习律的另一个例子——BCM法则。它的公式如下：

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

其中 η 为学习速率， r_θ 为学习的阈值（见图2-7）。图2-7画出了上式的右边，当发放频率高于阈值时呈现LTP，低于阈值时则为LTD。因此，这是一种频率依赖可塑性（回想一下，前面介绍的STDP为时间依赖可塑性），可以通过调整阈值 r_θ 实现选择性。

图2-7 BCM法则 (引自 Gerstner et al., 2014¹)

我们将实现和Oja法则相同的连接方式（图2-6），但给的刺激不同。在这里，我们让 j_1 （蓝色）和 j_2 （红色）交替发放，且 j_1 的发放率比 j_2 高。我们动态调整阈值为 r_i 的时间平均，即 $r_\theta = f(r_i) = \frac{\int dt r_i}{T}$ 。BrainPy 实现的代码如下。

```

1  class BCM(bp.TwoEndConn):
2      target_backend = ['numpy', 'numba']
3
4      @staticmethod
5      def derivative(w, t, lr, r_pre, r_post, r_th):
6          dwdt = lr * r_post * (r_post - r_th) * r_pre
7          return dwdt
8
9      def __init__(self, pre, post, conn, lr=0.005, w_max=1., w_min=0., **kwargs):
10         # parameters
11         self.lr = lr
12         self.w_max = w_max
13         self.w_min = w_min
14         self.dt = bp.backend.get_dt()
15
16         # connections
17         self.conn = conn(pre.size, post.size)
18         self.conn_mat = conn.requires('conn_mat')
19         self.size = bp.ops.shape(self.conn_mat)
20
21         # variables
22         self.w = bp.ops.ones(self.size) * .5
23         self.sum_post_r = np.zeros(post.size[0])
24
25         self.int_w = bp.odeint(f=self.derivative, method='rk4')
26
27         super(BCM, self).__init__(pre=pre, post=post, **kwargs)
28
29     def update(self, _t):
30         # update threshold
31         self.sum_post_r += self.post.r
32         r_th = self.sum_post_r / (_t / self.dt + 1)
33
34         # resize to matrix
35         dim = self.size
36         r_th = np.vstack((r_th,) * dim[0])
37         r_post = np.vstack((self.post.r,) * dim[0]).T
38         r_pre = np.vstack((self.pre.r,) * dim[1]).T
39
40         # update w
41         w = self.int_w(self.w * self.conn_mat, _t, self.lr, r_pre, r_post, r_th)
42         self.w = np.clip(w, self.w_min, self.w_max)           —————> 给w值设置边界
43
44         # output
45         self.post.r = np.sum(w.T * self.pre.r, axis=1)    —————> 跟Oja法则一样,  $r_i = \sum_j w_{ij} r_j$ 
```

定义了BCM类以后，我们可以跑模拟了。

```

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

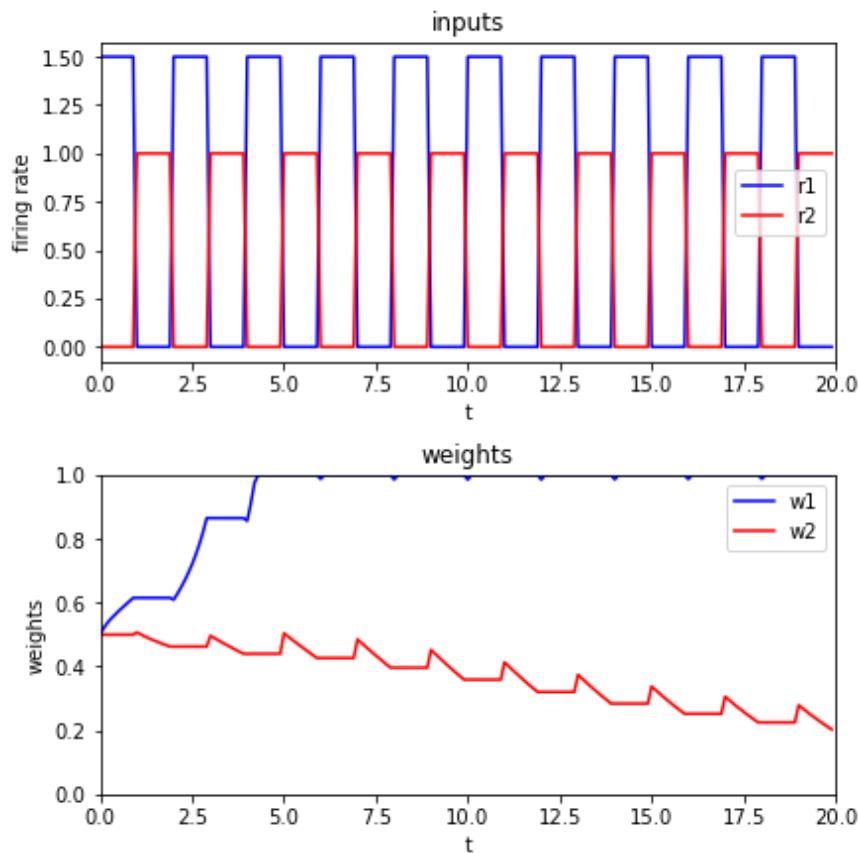
# simulate
pre = neu(20, monitors=['r'])
post = neu(1, monitors=['r'])
bcm = BCM(pre=pre, post=post, conn=bp.connect.All2All(),
           monitors=['w'])
net = bp.Network(pre, bcm, post)
net.run(duration, inputs=(pre, 'r', input_r.T, "="))

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

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

plt.show()

```



结果显示，每次发放率都比较高的 j_1 （蓝色），其对应的 w_1 持续增加，显示出LTP。而 w_2 则呈现出LTD，这个结果显示出BCM法则的选择功能。

参考资料

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

3. 网络模型

到此，读者已经了解了几种最常见、最经典的神经元和突触模型，是时候更进一步了。本节中，我们将介绍计算神经科学中两类重要的网络模型：脉冲神经网络（spiking neural networks）和发放率神经网络（firing rate neural networks）。

脉冲神经网络的特点是网络分别建模、计算每个神经元和突触。研究者希望通过这种仿真来观察大规模神经网络的行为，并验证相关理论推导。

而发放率神经网络则将一个神经元群简化为单个的发放率单元，以计算整个神经元群的发放率代替对单神经元的模拟。这种模拟经常可以得到和脉冲神经网络在网络尺度上类似的结果，而计算过程却被大大地简化了。

本章中，我们将介绍分属于这两类网络模型的兴奋-抑制平衡网络、抉择网络和连续吸引子神经网络。

注：本章所述模型的完整BrainPy代码请见[附录](#)，或右键点此下载jupyter notebook版本。

3.1 兴奋-抑制平衡网络

3.2 抉择网络

3.3 连续吸引子神经网络

3.1 兴奋-抑制平衡网络

上世纪90年代初，学界发现，在大脑皮层中神经元有时表现出一种在时间上不规则的发放特征。这种特征广泛地存在于脑区中，但当时人们对它的产生机制和主要功能都了解不多。

Vreeswijk和Sompolinsky (1996) 提出了**兴奋-抑制平衡网络** (E/I balanced network)，希望能够解释神经元这种不规则的发放，并提示了这种结构在功能上可能的优势。

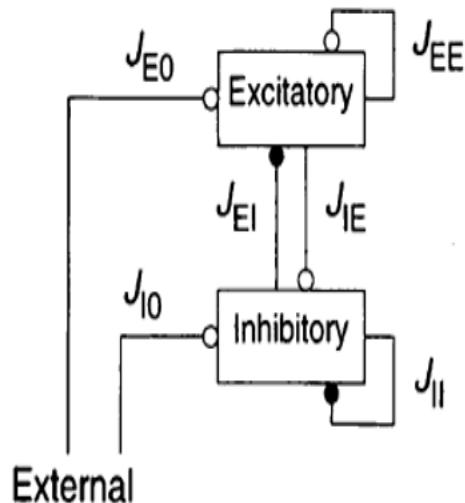


图3-1 兴奋-抑制平衡网络结构 (*Vreeswijk and Sompolinsky, 1996*¹)

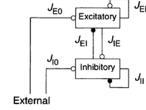
图3-1画出了兴奋-抑制平衡网络的结构。该网络由兴奋性LIF神经元和抑制性LIF神经元构成，其数量比 $N_E : N_I = 4 : 1$ 。在网络两类神经元之间和同类神经元之内，建立了四组指型突触连接，分别是兴奋-兴奋连接 (E2E conn)，兴奋-抑制连接 (E2I conn)，抑制-兴奋连接 (I2E conn)，抑制-抑制连接 (I2I conn)。在代码中我们通过定义符号相反的突触权重，来指明突触连接的兴奋性或抑制性。

1.1 生物背景

```

9  N_E = 500
10 N_I = 500
11 prob = 0.1
53
54 neu_E = brainmodels.neurons.LIF(N_E, monitors=['spike'])
55 neu_I = brainmodels.neurons.LIF(N_I, monitors=['spike'])
56 neu_E.V = V_rest + np.random.random(N_E) * (V_th - V_rest)
57 neu_I.V = V_rest + np.random.random(N_I) * (V_th - V_rest)
58
59 syn_E2E = brainmodels.synapses.Exponential(
60     pre=neu_E, post=neu_E,
61     conn=bp.connect.FixedProb(prob=prob))
62 syn_E2I = brainmodels.synapses.Exponential(
63     pre=neu_E, post=neu_I,
64     conn=bp.connect.FixedProb(prob=prob))
65 syn_I2E = brainmodels.synapses.Exponential(
66     pre=neu_I, post=neu_E,
67     conn=bp.connect.FixedProb(prob=prob))
68 syn_I2I = brainmodels.synapses.Exponential(
69     pre=neu_I, post=neu_I,
70     conn=bp.connect.FixedProb(prob=prob))
71
72 JE = 1 / np.sqrt(prob * N_E)
73 JI = 1 / np.sqrt(prob * N_I)
74 syn_E2E.w = JE
75 syn_E2I.w = JE
76 syn_I2E.w = -JI
77 syn_I2I.w = -JI
78
    } 从brainmodels包中导入LIF神经元模型。
    } 建立兴奋性和抑制性神经元群。
    } 从brainmodels包中导入指数型突触模型。
    } 在神经元群之间建立E2E、E2I、I2E、I2I的突触连接。
    } 突触连接权重 =  $J_E = J_I = \frac{1}{\sqrt{p N_E}} = \frac{1}{\sqrt{p N_I}}$ 

```



注：LIF神经元和指数型突触的实现请参见第1节《神经元模型》和第2节《突触模型》

兴奋-抑制平衡网络在结构上最大的特征是神经元间强随机突触连接，连接概率为0.1，属于稀疏连接。

这种强的突触连接使得网络中每个神经元都会接收到很大的来自网络内部的兴奋性和抑制性输入。但是，这两种输入一正一负相互抵消，最后神经元接收到的总输入将保持在一个相对小的数量级上，仅足以让神经元的膜电位上升到阈值电位，引发其产生动作电位。

由于突触连接和噪声带来的随机性，网络中神经元接收到的输入也在时间和空间上具有一定的随机性（尽管总体保持在阈值电位量级上），这使得神经元的发放也具有随机性，保证兴奋-抑制平衡网络能够自发产生前述的时间上不规则的发放特征。

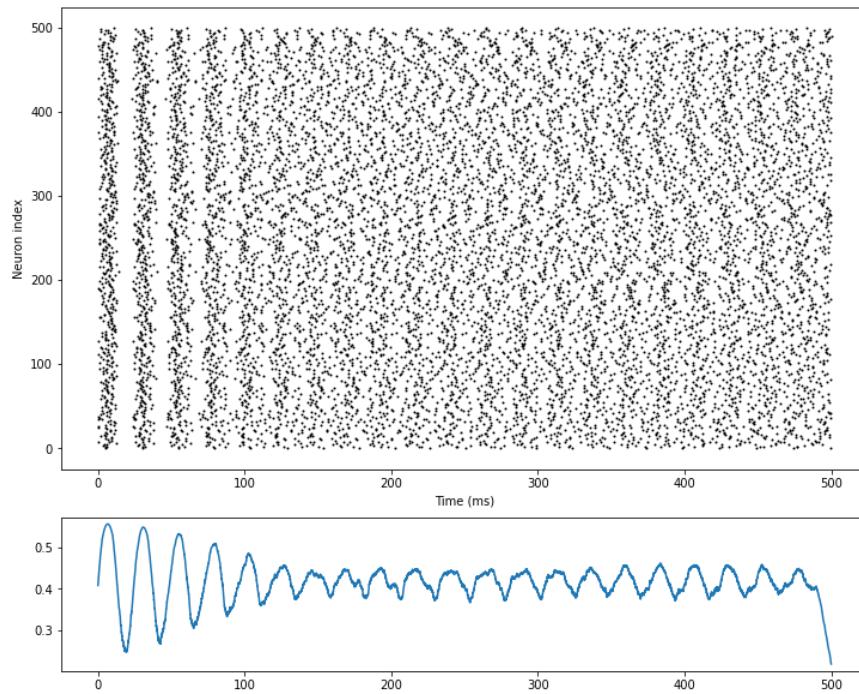
下述仿真结果中，可以看到网络中的神经元从一开始的强同步发放慢慢变为时间上不规则的发放。

```

79 net = bp.Network(neu_E, neu_I,
80                     syn_E2E, syn_E2I,
81                     syn_I2E, syn_I2I)
82 net.run(500., inputs=[(neu_E, 'input', 3.), (neu_I, 'input', 3.)], report=True)
83
84 fig, gs = bp.visualize.get_figure(4, 1, 2, 10)
85 fig.add_subplot(gs[0:3, 0])
86 bp.visualization.raster_plot(net.ts, neu_E.mon.spike)
87
88 fig.add_subplot(gs[3, 0])
89 rate = bp.measure.firing_rate(neu_E.mon.spike, 5.)
90 plt.plot(net.ts, rate)
91 plt.show()
92
    } 网络仿真500ms，给所有神经元
    } 大小为3的恒定外部输入。
    } 画出兴奋-抑制平衡网络中兴奋性神经元的
    } 发放点阵图，以及其发放率随时间的变化。

```

1.1 生物背景



与此同时，作者还提出了这种发放特征在大脑中可能提供的功能：兴奋-抑制平衡网络可以快速跟踪外部刺激的变化。假如该网络真的是大脑中神经元产生不规则发放背后的机制，那么真实的神经元网络也可能拥有同样的特性。

如图3-2所示，当没有外部输入时，兴奋-抑制平衡网络中神经元的膜电位相对均匀且随机地分布在静息电位 V_0 和阈值电位 θ 之间。当网络接收到一个小的外部恒定输入时，那些膜电位原本就落在阈值电位附近的神经元（图中标为红色）就能很快地发放，在网络尺度上，表现为网络的发放率随输入变化而快速改变。

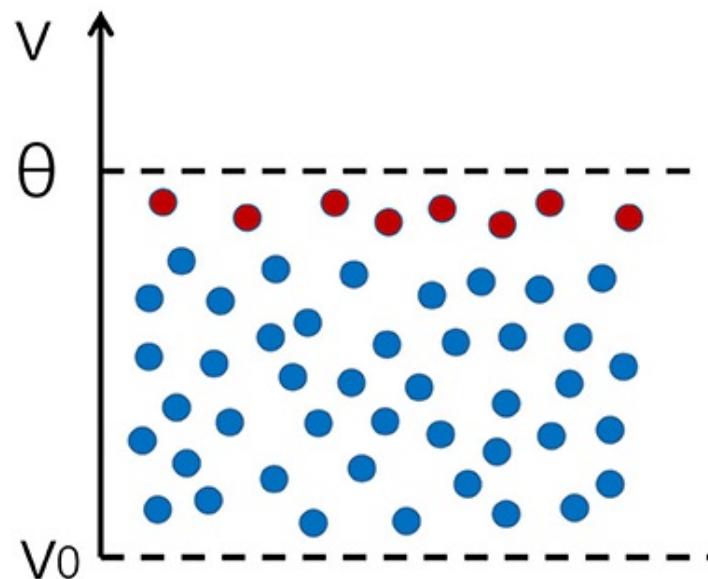


图3-2 兴奋-抑制平衡网络中神经元膜电位的分布 (Tian et al., 2020²)

1.1 生物背景

仿真证实，在这种情况下，网络对输入产生反应的延迟时间和突触的延迟时间处于同一量级，并且二者都远小于单神经元从静息电位开始积累同样大小的外部输入直到产生动作电位所需的延迟时间（Vreeswijk和Sompolinsky, 1996; Tian et al., 2020）。因此，兴奋-抑制平衡网络面对外部输入的变化可以快速反应，改变自身的活跃水平。

参考资料

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

3.2 抢择网络

计算神经科学的网络建模也可以对标特定的生理实验任务，比如视觉运动区分实验（Parker和Newsome，1998；Roitman和Shadlen，2002）。

在该实验中，参与实验的猕猴将观看一段随机点的运动展示。在展示过程中，随机点以一定比例（该比例被定义为一致度（coherence））向特定方向运动，其他点则向随机方向运动。猕猴被要求判断随机点一致运动的方向，并通过眼动给出答案。同时，研究者通过电生理手段记录猕猴LIP神经元的活动。

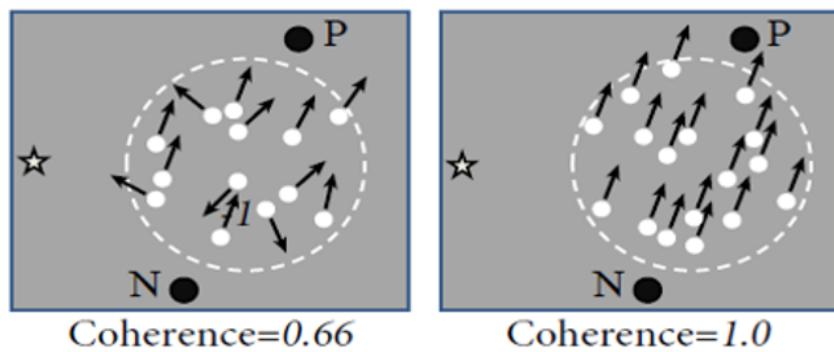
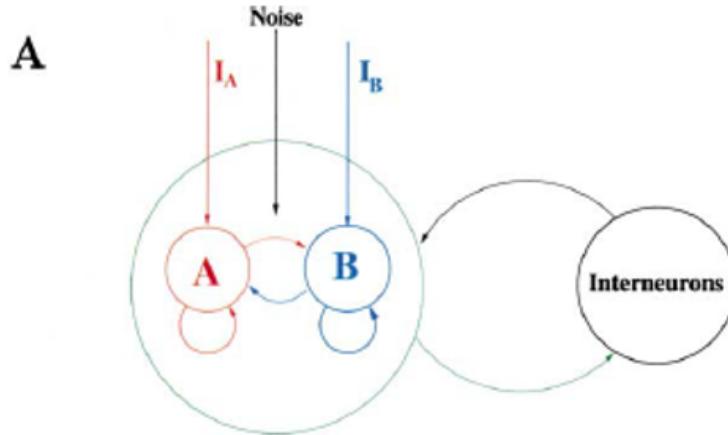


图3-3 生理实验中随机点的运动示意图 (Gerstner et al., 2014¹)

Wang等人（2002；2006）先后提出了本节所述的脉冲神经网络模型和发放率神经网络模型，希望建模在视觉运动区分实验中猕猴大脑新皮层的抉择回路的活动。

3.2.1 脉冲神经网络

Wang（2002）首先提出了本节所述的抉择脉冲神经网络。如图3-4所示，网络基于3.1节所述的兴奋-抑制平衡网络。兴奋性神经元和抑制型神经元的数量比是 $N_E : N_I = 4 : 1$ ，调整参数使得网络处在平衡状态下。

图3-4 抢择网络结构 (Wang, 2002²)

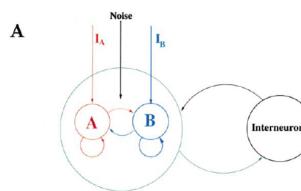
为了简化模型，实验被设定为一个二选一的任务：在兴奋性神经元群中，特别地标出两个选择性子神经元群A和B，其他的兴奋性神经元称为非选择性神经元，用下标_{non}表示。A群和B群的数目均为兴奋性神经元的0.15倍 ($N_A = N_B = 0.15N_E$)，它们分别代表着两个相反的运动方向，可以视作随机点要么向左，要么向右，没有第三个方向，网络的抉择结果也必须在这两个子群中产生。非选择性神经元的数目为

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

```

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

```



抉择网络中共有四组突触——E2E， E2I， I2E和I2I突触连接，其中兴奋性突触实现为AMPA突触，抑制性突触实现为GABAa突触。

由于网络需要在A群和B群之间作出抉择，所以这两个子神经元群之间必须形成一种竞争关系。一个选择性子神经元群应当激活自身，并同时抑制另一个选择性子神经元群。

因此，网络中的E2E连接被建模为有结构的连接。如表3-1所示， $w+ > 1 > w-$ 。这样，在A群或B群的内部，兴奋性突触连接更强，形成了一种相对的自激活；而在A、B两个选择性子神经元群之间或是A群、B群和非选择性子神经元群之间，兴奋性突触连接较弱，实际上形成了相对的抑制。A和B两个神经元因此产生竞争，迫使网络做出二选一的抉择。

表3-1 决策网络中兴奋性神经元间连接权重的分布

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

```

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

```

定义E2E连接的结构性权重。
在选择性神经元群内部的突触连接，较其他突触连接更强。

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

抉择网络接受的外部输入可分为两类：

- 所有神经元都收到从其他脑区传来的非特定的背景输入，表示为AMPA突触介导的高频泊松输入（2400Hz）。

1.1 生物背景

```
488 # def background poisson input
489 class PoissonInput(bp.NeuGroup):
490     target_backend = 'general'
491
492     def __init__(self, size, freqs, dt, **kwargs):
493         self.freqs = freqs
494         self.dt = dt
495
496         self.spike = bp.ops.zeros(size, dtype=bool)
497
498     super(PoissonInput, self).__init__(size=size, **kwargs)
499
500     def update(self, _t):
501         self.spike = np.random.random(self.size) \
502             < self.freqs * self.dt / 1000. ] 根据给定的泊松频率生成峰电位。
503
504     # poisson_freq = 2400Hz
505     neu_poisson_A = PoissonInput(N_A, freqs=poisson_freq, dt=dt)
506     neu_poisson_B = PoissonInput(N_B, freqs=poisson_freq, dt=dt)
507     neu_poisson_non = PoissonInput(N_non, freqs=poisson_freq, dt=dt)
508     neu_poisson_I = PoissonInput(N_I, freqs=poisson_freq, dt=dt) ] 定义生成背景输入的神经元,
509
510     syn_back2A_AMPA = AMPA(pre=neu_poisson_A, post=neu_A,
511                             conn=bp.connect.OneToOne())
512     syn_back2B_AMPA = AMPA(pre=neu_poisson_B, post=neu_B,
513                             conn=bp.connect.OneToOne())
514     syn_back2non_AMPA = AMPA(pre=neu_poisson_non, post=neu_non,
515                             conn=bp.connect.OneToOne())
516
517     syn_back2I_AMPA = AMPA(pre=neu_poisson_I, post=neu_I,
518                             conn=bp.connect.OneToOne()) ] 定义介导背景泊松输入的AMPA突触。
```

2) 仅选择性的A群和B群收到外部传来的刺激输入，表示为AMPA突触介导的较低频泊松输入（约100Hz内）。

给予A和B神经元群的泊松输入的频率均值（ μ_A 、 μ_B ）有一定差别，对应到生理实验上，代表猕猴看到的随机点朝两个相反方向运动的比例（用 coherence表示）不同。这种输入上的差别引导着网络在两个子神经元群中做出抉择。

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

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

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

每50毫秒，泊松输入的实际频率 f_x 遵循由均值 μ_x 和方差 δ^2 定义的高斯分布，重新进行一次采样。

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

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

1.1 生物背景

```

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

```

计算给予A、B两个神经元群的刺激泊松输入的泊松频率。
 $\mu_A = \mu_0 + \rho_A * c$
 $\mu_B = \mu_0 - \rho_B * c$

存储模型参数和变量。

仅在[stim_start_t, stim_end_t]这一时间范围内产生刺激泊松输入。

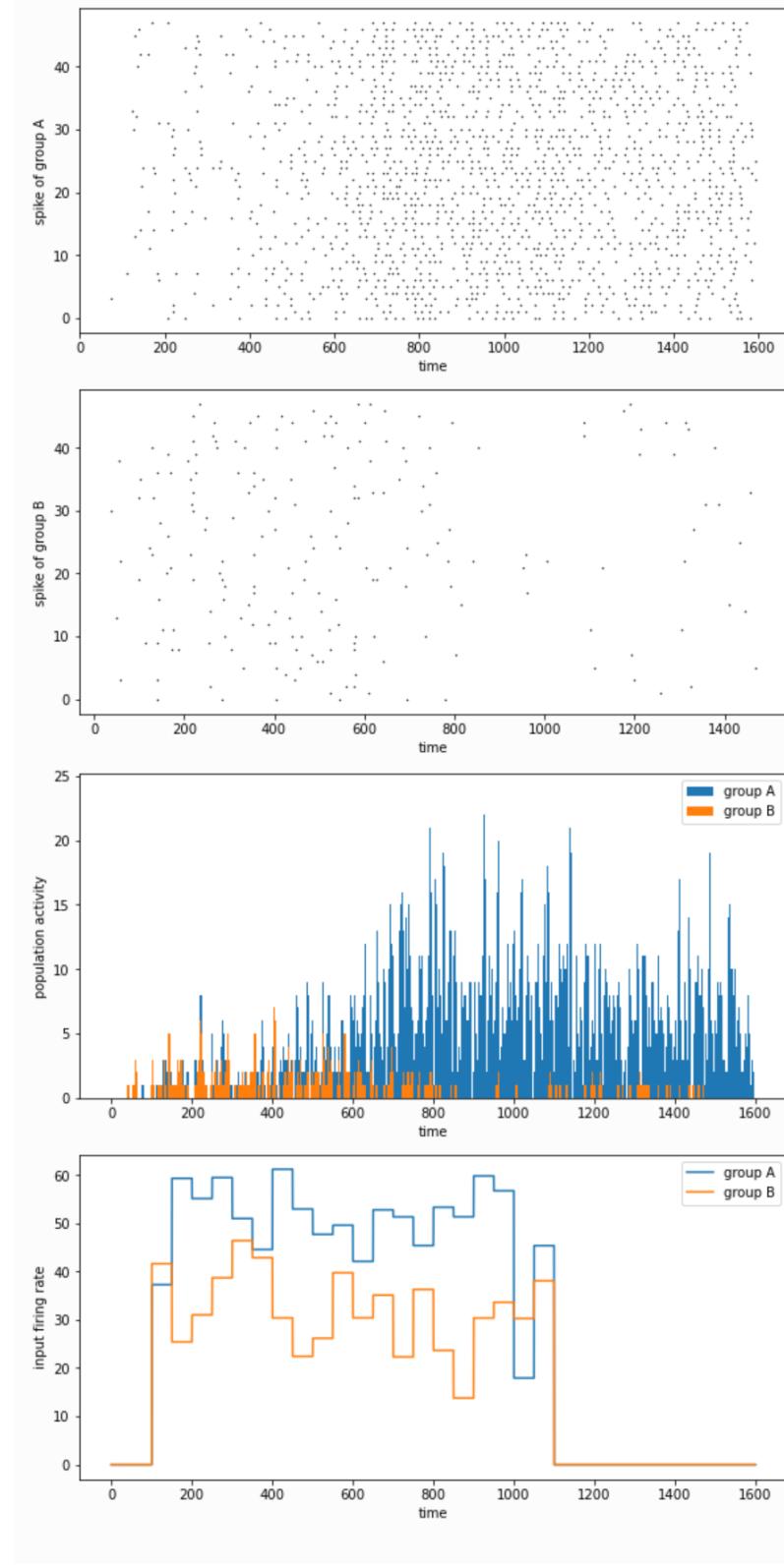
每50ms, 为当前神经元生成一个新的泊松频率。该频率采样服从高斯分布。
 $f_A \sim N(\mu_A, \delta^2)$
 $f_B \sim N(\mu_B, \delta^2)$

根据泊松频率生成当前神经元的发放状态, 从而生成刺激泊松输入。

定义生成刺激输入的神经元, 为A群、B群中每个神经元提供相对低频的刺激泊松输入。

定义介导刺激泊松输入的AMPA突触。

下图中可以看到, 在本次仿真中, 子神经元群A收到的刺激输入平均大于B收到的刺激输入。经过一定的延迟时间, A群的活动水平明显高于B群, 说明网络做出了正确的选择。



3.2.2 发放率神经网络

我们在上一节中介绍了Wang (2002) 提出的抉择模型，现在来介绍他们后续做的一个基于发放率 (firing rate) 的简化模型 (Wong & Wang, 2006)。该模型的实验背景与上一节的相同，在脉冲神经网络模型的基础上，他们使用平均场近似 (mean-field approach) 等方法，使用一群神经元的发放率来表示整群神经元的状态，而不再关注每个神经元的脉冲。他们拟合出输入-输出函数 (input-output function) 来表示给一群神经元一个外界输入电流 I 时，这群神经元的发放率 r 如何改变，即 $r = f(I)$ 。经过这样的简化后，我们就可以很方便地对其进行动力学分析。

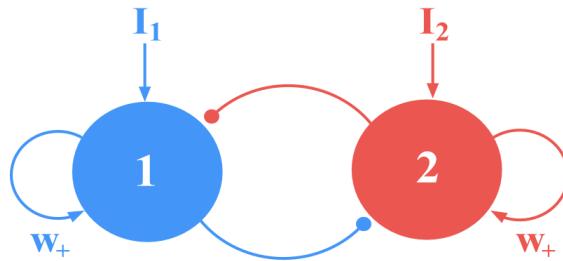


图3-5 简化的抉择模型 (引自 Wong & Wang, 2006³)

基于发放率的抉择模型如图3-5所示， S_1 (蓝色) 和 S_2 (红色) 分别表示两群神经元的状态，同时也分别对应着两个选项。他们都由兴奋性的神经元组成，且各自都有一个循环 (recurrent) 连接。而同时它们都会给对方一个抑制性的输入，以此形成相互竞争的关系。该模型的动力学方程如下：

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

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

其中 τ 为时间常数， γ 为拟合得到的常数， r_1 和 r_2 分别为两群神经元的发放率，其输入-输出函数为：

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

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

$I_{syn,i}$ 的公式由图3-5的模型结构给出：

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

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

1.1 生物背景

其中 I_0 为背景电流，外界输入 I_1, I_2 则由总输入的强度 μ_0 及一致性

(coherence) c' 决定。一致性越高，则越明确 S_1 是正确答案，而一致性越低则表示越随机。公式如下：

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

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

接下来，我们将继承 `bp.NeuGroup` 类，并用BrainPy提供的相平面分析方法 `bp.analysis.PhasePlane` 进行动力学分析。首先，我们把上面的动力学公式写到一个 `derivative` 函数中，定义一个Decision类（本章代码请查看[附录](#)，或[点此](#)下载jupyter notebook）。

```

1  from collections import OrderedDict
2  import brainpy as bp
3
4  bp.backend.set(backend='numba', dt=0.1)
5
6
7  class Decision(bp.NeuGroup):
8      target_backend = ['numpy', 'numba']
9
10     @staticmethod
11     def derivative(s1, s2, t, I, coh,
12                     JAext, J_rec, J_inh, I_0,
13                     a, b, d, tau_s, gamma):
14         I1 = JAext * I * (1. + coh)    ]   I1 = JAextμ0(1 + c')
15         I2 = JAext * I * (1. - coh)    ]   I2 = JAextμ0(1 - c')
16
17         Isyn1 = Jrec * s1 - Jinh * s2 + I0 + I1
18         r1 = (a * Isyn1 - b) / (1. - bp.ops.exp(-d * (a * Isyn1 - b)))
19         ds1dt = -s1 / tau_s + (1. - s1) * gamma * r1
20
21         Isyn2 = Jrec * s2 - Jinh * s1 + I0 + I2
22         r2 = (a * Isyn2 - b) / (1. - bp.ops.exp(-d * (a * Isyn2 - b)))
23         ds2dt = -s2 / tau_s + (1. - s2) * gamma * r2
24
25         return ds1dt, ds2dt
26
27     def __init__(self, size, coh, JAext=.00117, J_rec=.3725, J_inh=.1137,
28                 I_0=.3297, a=270., b=108., d=0.154, tau_s=.06, gamma=0.641,
29                 **kwargs):
30         # parameters
31         self.coh = coh
32         self.JAext = JAext
33         self.J_rec = J_rec
34         self.J_inh = J_inh
35         self.I0 = I_0
36         self.a = a
37         self.b = b
38         self.d = d
39         self.tau_s = tau_s
40         self.gamma = gamma
41
42         # variables
43         self.s1 = bp.ops.ones(size) * .06
44         self.s2 = bp.ops.ones(size) * .06
45         self.input = bp.ops.zeros(size)
46
47         self.integral = bp.odeint(f=self.derivative, method='rk4', dt=0.01)
48
49         super(Decision, self).__init__(size=size, **kwargs)
50
51     def update(self, _t):
52         for i in range(self.size):
53             self.s1[i], self.s2[i] = self.integral(self.s1[i], self.s2[i], _t,
54                                                 self.input[i], self.coh,
55                                                 self.JAext, self.J_rec,
56                                                 self.J_inh, self.I0,
57                                                 self.a, self.b, self.d,
58                                                 self.tau_s, self.gamma)
59             self.input[i] = 0.

```

$$\left. \begin{aligned} I_{syn,1} &= J_{11}S_1 - J_{12}S_2 + I_0 + I_1 \\ r_1 &= \frac{aI_{syn,1} - b}{1 - \exp(-d(aI_{syn,1} - b))} \\ \frac{ds_1}{dt} &= -S_1/\tau + (1 - S_1)\gamma r_1 \end{aligned} \right\} \begin{aligned} I_{syn,2} &= J_{22}S_2 - J_{21}S_1 + I_0 + I_2 \\ r_2 &= \frac{aI_{syn,2} - b}{1 - \exp(-d(aI_{syn,2} - b))} \\ \frac{ds_2}{dt} &= -S_2/\tau + (1 - S_2)\gamma r_2 \end{aligned}$$

接下来，我们想要看模型在不同输入情况下的动力学，因此，我们先定义一个对抉择模型做相平面分析的方法，可以让我们改变 I （即外界输入强度 μ_0 ）和 coh （即输入的一致性 c' ），而固定了参数的值等。

```

62 def phase_analyze(I, coh):
63     decision = Decision(I, coh=coh)
64
65     phase = bp.analysis.PhasePlane(decision.integral,           神经元模型中定义微分方程的函数
66                                     target_vars=OrderedDict(s2=[0., 1.],           相平面分析的两个变量
67                                     s1=[0., 1.]),
68                                     fixed_vars=None,
69                                     pars_update=dict(I=I, coh=coh,
70                                         J_Aext=.00117, J_rec=.3725,
71                                         J_inh=.1137, I_B=.3297,
72                                         a=270., b=108., d=0.154,
73                                         tau_s=.06, gamma=0.641),
74                                     numerical_resolution=.001,
75                                     options={'escape_sympy_solver': True})           指定参数值
76
77     phase.plot_nullcline()
78     phase.plot_fixed_point()
79     phase.plot_vector_field(show=True)           由于模型中的微分方程非常复杂，这里我们使用数值解，直接跳过sympy提供的解析解，并设定数值解的解析度(resolution)。

```

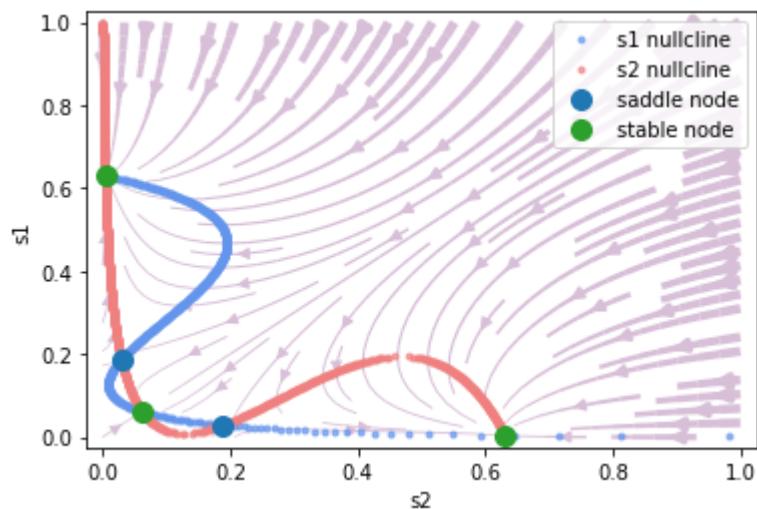
现在让我们来看看当没有外界输入，即 $\mu_0 = 0$ 时的动力学。

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

```

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

```



由此可见，用BrainPy进行动力学分析是非常方便的。向量场和不动点(fixed point)表示了不同初始值下最终会落在哪个选项。

1.1 生物背景

这里，x轴是 S_2 ，代表选项2，y轴是 S_1 ，代表选项1。可以看到，左上的不动点表示选项1，右下的不动点表示选项2，左下的不动点表示没有选择。

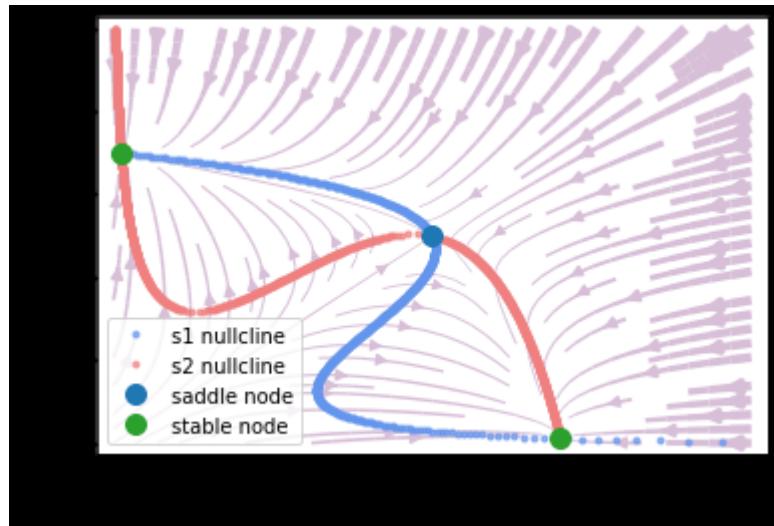
现在让我们看看当我们把外部输入强度固定为30时，在不同一致性（coherence）下的相平面。

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

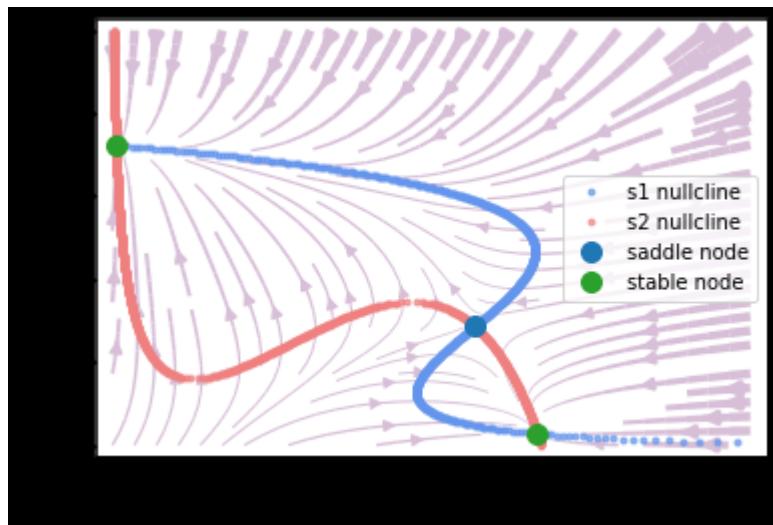
# coherence = 51.2%
print("coherence = 51.2%")
phase_analyze(I=30., coh=0.512)

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

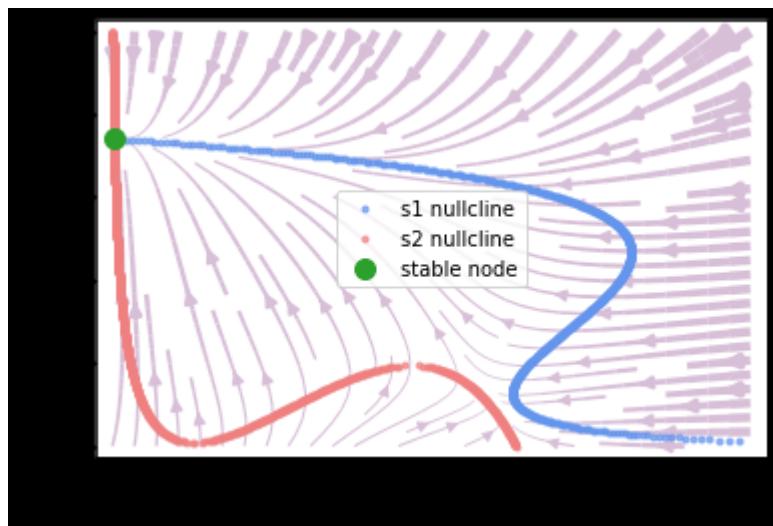
```
coherence = 0%
plot nullcline ...
plot fixed point ...
Fixed point #1 at s2=0.6993504413889349, s1=0.0116220495267
Fixed point #2 at s2=0.49867489858358865, s1=0.49867489858358865
Fixed point #3 at s2=0.011622051540013889, s1=0.69935043555
plot vector field ...
```



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



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



参考资料

1.1 生物背景

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

3.2 连续吸引子模型 (CANN)

这里我们将介绍发放率模型的另一个例子——连续吸引子神经网络 (CANN)。图3-6呈现了一维CANN的结构。

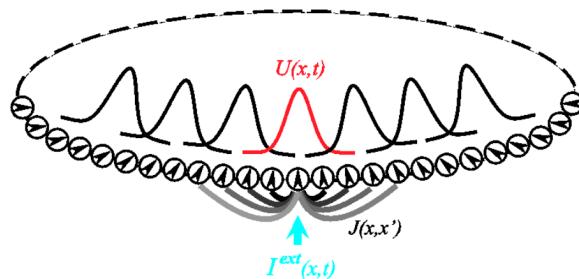


图3-6 连续吸引子神经网络 (引自 Wu et al., 2008¹)

神经元的突触总输入 u 的动力学方程如下：

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

其中 x 表示神经元的参数空间位点， $r(x',t)$ 为神经元(x')的发放率，由以下公式给出：

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

而神经元(x)和(x')之间的兴奋性连接强度 $J(x,x')$ 由高斯函数给出：

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

外界输入 I_{ext} 与位置 $z(t)$ 有关，公式如下：

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

用BrainPy实现的代码如下，我们通过继承 `bp.NeuGroup` 来创建一个 `CANN1D` 的类。

1.1 生物背景

```

1 import brainpy as bp
2 import numpy as np
3 bp.backend.set(backend='numpy', dt=0.1)
4
5 class CANN1D(bp.NeuGroup):
6     target_backend = ['numpy', 'numba']
7
8     def __init__(self, num, tau=1., k=8.1, a=0.5, A=10., J0=4.,
9                  z_min=-np.pi, z_max=np.pi, **kwargs):
10        # parameters
11        self.tau = tau # The synaptic time constant
12        self.k = k # Degree of the rescaled inhibition
13        self.a = a # Half-width of the range of excitatory connections
14        self.A = A # Magnitude of the external input
15        self.J0 = J0 # maximum connection value
16
17        # feature space
18        self.z_min = z_min
19        self.z_max = z_max
20        self.z_range = z_max - z_min
21        self.x = np.linspace(z_min, z_max, num) # The encoded feature values
22
23        # variables
24        self.u = np.zeros(num)
25        self.input = np.zeros(num)
26
27        # The connection matrix
28        self.conn_mat = self.make_conn(self.x)
29
30        super(CANN1D, self).__init__(size=num, **kwargs)
31
32        self.rho = num / self.z_range # The neural density
33        self.dx = self.z_range / num # The stimulus density
34
35    @staticmethod
36    @bp.odeint(method='rk4', dt=0.05)
37    def int_u(u, t, conn, k, tau, Iext):
38        r1 = np.square(u)
39        r2 = 1.0 + k * np.sum(r1)
40        r = r1 / r2
41        Irec = np.dot(conn, r)
42        du = (-u + Irec + Iext) / tau
43        return du
44
45    def dist(self, d):
46        d = np.remainder(d, self.z_range)
47        d = np.where(d > 0.5 * self.z_range, d - self.z_range, d)
48        return d
49
50    def make_conn(self, x):
51        assert np.ndim(x) == 1
52        x_left = np.reshape(x, (-1, 1))
53        x_right = np.repeat(x.reshape((1, -1)), len(x), axis=0)
54        d = self.dist(x_left - x_right)
55        Jxx = self.J0 * np.exp(-0.5 * np.square(d / self.a)) / (
56            np.sqrt(2 * np.pi) * self.a)
57        return Jxx
58
59    def get_stimulus_by_pos(self, pos):
60        return self.A * np.exp(-0.25 * np.square(self.dist(self.x -
61            pos) / self.a))
62
63    def update(self, _t):
64        self.u = self.int_u(self.u, _t, self.conn_mat, self.k, self.tau,
65                            self.input)
66        self.input[:] = 0.

```

z_{range} 表示 x 的范围，这里为环上的坐标， $x \in [-\pi, \pi]$

$\rho \int dx'$

处理环上的距离：
最远的距离为 z_{range} 的一半，因此 d 不能超过 $0.5 * z_{range}$

计算距离矩阵 \longrightarrow

	x_1	x_2	\dots
x_1		d_{12}	
x_2			d_{21}
\dots			

 $d = |x - x'|$

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

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

这里我们用函数 `dist` 与 `make_conn` 来计算两群神经元之间的连接强度

$J(x, x')$ 。其中 `dist` 函数用来处理环上的距离。

接下来我们可以调用刚才定义的 `get_stimulus_by_pos` 方法获取外界输入电流大小。例如在简单的群体编码（population coding）中，我们给一个 `pos=0` 的外界输入，并按以下方式运行：

1.1 生物背景

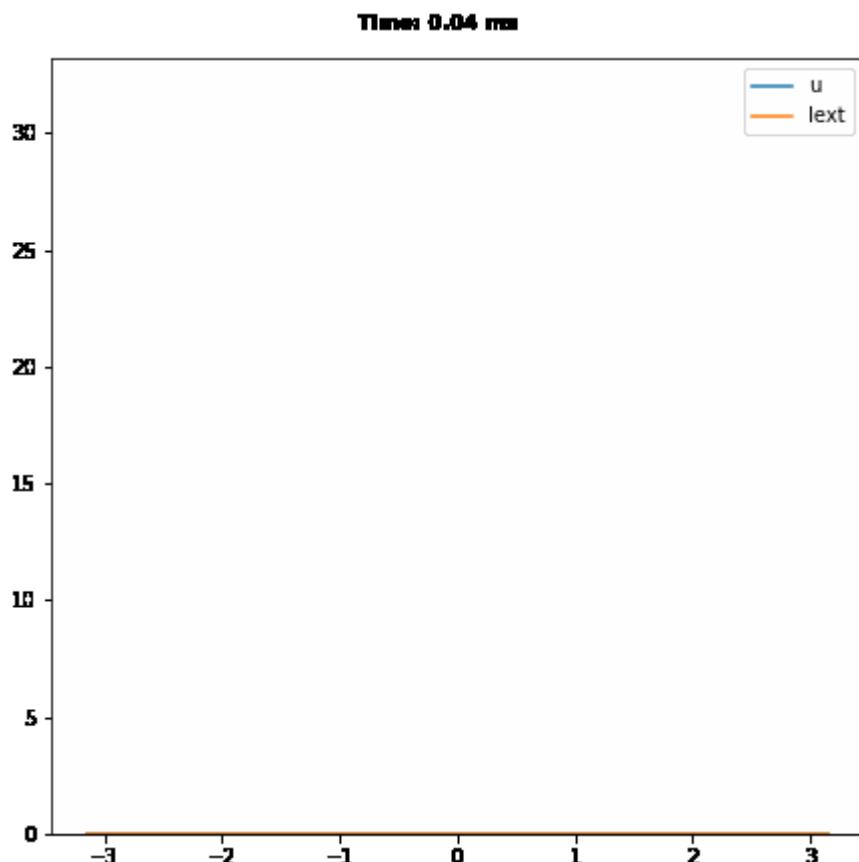
```
cann = CANN1D(num=512, k=0.1, monitors=['u'])

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

我们写一个 `plot_animate` 的函数来方便重复调用 `bp.visualize.animate_1D` 画结果图。

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

# 调用函数
plot_animate(frame_step=1, frame_delay=100)
```



可以看到， u 的形状编码了外界输入的形状。

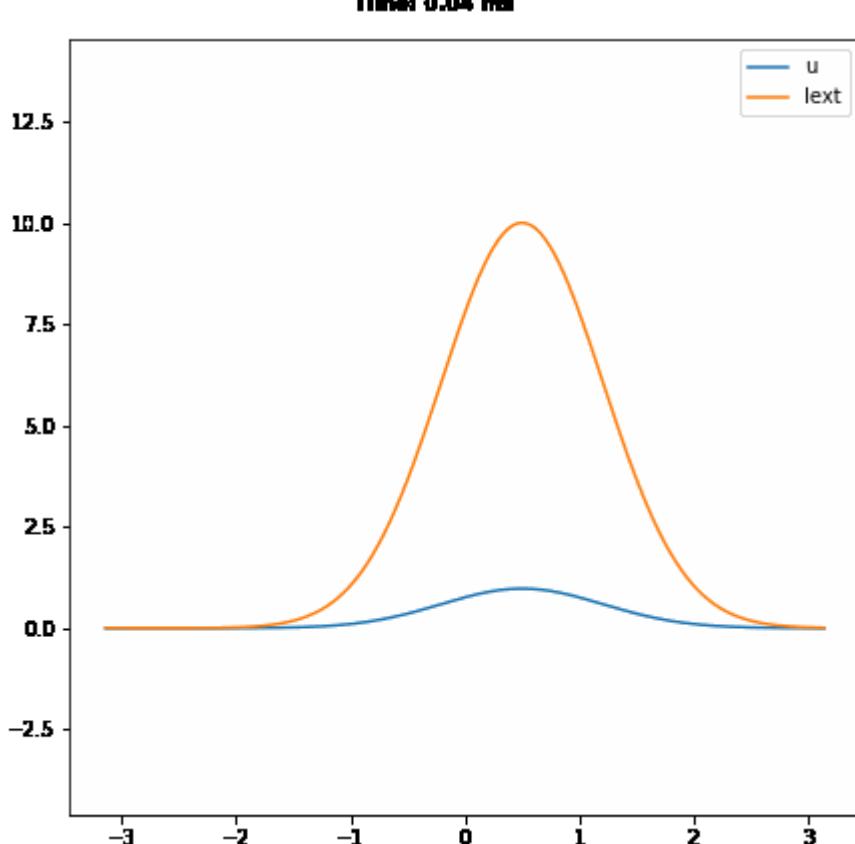
1.1 生物背景

现在我们给外界输入加上随机噪声，看看 u 的形状如何变化。

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

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

plot_animate()
```



我们可以看到 u 的形状保持一个类似高斯的钟形，这表明CANN可以进行模板匹配。

接下来我们用 `np.linspace` 函数来产生不同的位置，得到随时间平移的输入，我们将会看到 u 跟随着外界输入移动，即平滑跟踪。

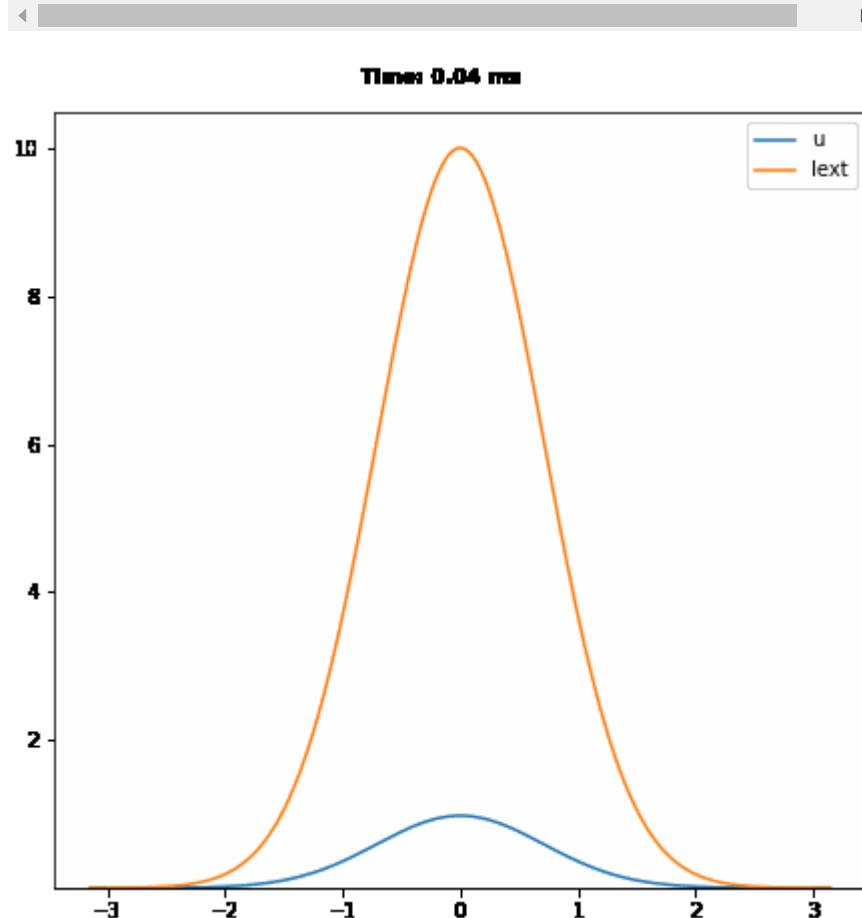
```

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

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

plot_animate()

```



参考资料

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

附录

附录中所载代码为本书正文展示代码的完整版。我们亦提供常用模型的调用接口，如需查看，请参见github仓库<https://github.com/PKU-NIP-Lab/BrainModels>及文档[https://brainmodels.readthedocs.io/en/latest/。](https://brainmodels.readthedocs.io/en/latest/)

- 附录1. [神经元模型](#)
- 附录2. [突触模型](#)
- 附录3. [网络模型](#)

生理模型

Hodgkin-Huxley模型

```

import brainpy as bp
from numba import prange

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

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

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

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

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

    return dVdt, dmdt, dhdt, dnndt

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

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

```

```

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

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

import brainpy as bp

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

```

简化模型

LIF模型

```

import brainpy as bp
from numba import prange

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

    Parameters
    ----------
    size : int
        Number of neurons.
    t_refractory : float, optional
        Refractory time after each spike. Default is 1.
    V_rest : float, optional
        Resting potential. Default is 0.
    V_reset : float, optional
        Reset potential. Default is -5.
    V_th : float, optional
        Threshold potential. Default is 20.
    R : float, optional
        Resistance. Default is 1.
    tau : float, optional
        Time constant. Default is 10.
    **kwargs : dict, optional
        Other keyword arguments.

    Methods
    -------
    update(_t)
        Update the neuron states at time _t.
    """

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

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

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

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

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

```

1.1 生物背景

```
import brainpy as bp

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

QualF模型

```

import brainpy as bp
from numba import prange

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

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

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

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

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

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

```

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

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

ExplF模型

```

import brainpy as bp
from numba import prange

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

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

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

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

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

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

```

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

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

AdExIF 模型

```

import brainpy as bp
from numba import prange

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

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

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

    return dVdt, dwdt

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

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

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

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

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

```

```

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

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

```

Hindmarsh-Rose模型

```

import brainpy as bp
from numba import prange

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

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

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

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

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

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

```

```

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

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

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

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

```

GeneralizedIF模型

```

import brainpy as bp
from numba import prange

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

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

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

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

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

```

```

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

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

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

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

```

```

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

```

```

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

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

```

```

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

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

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

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


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

```

发放率模型

发放率单元

```

import brainpy as bp
from numba import prange

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

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

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

        return daedt, daidt

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

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

```

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

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



附录：突触模型

突触动力学模型

AMPA模型

```

import brainpy as bp

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

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

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

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

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

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

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

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

```

```
import brainmodels as bm

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

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

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

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

    ◀ ▶

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

NMDA模型

```

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

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

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

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

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

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

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

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

            # output

```

1.1 生物背景

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

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

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

◀ ▶

```
run_syn(NMDA)
```

GABA_B模型

1.1 生物背景

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

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

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

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

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

双指数差 (Differences of two exponentials)

```

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

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

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

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

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

        self.integral = bp.odeint(f=self.derivative, method='rk4',
                                  args=(self.tau1, self.tau2))

    super(Two_exponentials, self).__init__(pre=pre, post=post,
                                           conn=conn, integral=self.integral,
                                           target_backend=target_backend)

```

```

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

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

```

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

Alpha突触

```

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

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

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

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

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

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

```

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

```

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

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

```

run_syn(Alpha)

单指数衰减 (Single exponential decay)

```

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

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

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

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

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

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

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

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

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

    < >
run_syn(Exponential)

```

电压跳变 (Voltage jump)

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

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

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

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

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

    def run_syn(self):
        pass
```

缝隙连接 (Gap junction)

```

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

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

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

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

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

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

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

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

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

```

```
import matplotlib.pyplot as plt

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

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

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

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

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

突触可塑性模型

突触短时程可塑性 (STP)

```

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

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

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

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

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

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

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

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

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

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

```

```

    self.x[i] = x

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

```

```

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

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

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

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

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

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

```

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

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

脉冲时间依赖可塑性 (STDP)

```

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

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

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

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

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

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

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

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

```

1.1 生物背景

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

```

Oja法则

```

import numpy as np

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

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

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

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

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

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

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

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

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

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

```

1.1 生物背景

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

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

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

```

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

# simulate
neu_pre = neu(2, monitors=['r'])
neu_post = neu(2, monitors=['r'])
syn = Oja(pre=neu_pre, post=neu_post, conn=bp.connect.All2A
net = bp.Network(neu_pre, syn, neu_post)
net.run(duration=200., inputs=[(neu_pre, 'r', current_pre.1
                               (neu_post, 'r', current_post

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

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

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

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

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

```

BCM法则

```

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

plt.show()
```

脉冲神经网络

兴奋-抑制平衡网络

```

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

bp.backend.set('numba')

N_E = 500
N_I = 500
prob = 0.1

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

tau_decay = 2.

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

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

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

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

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

```

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

抉择网络

```

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

```

```

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

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

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

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

delay_syn = 0.5 # ms

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

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

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

```

```

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

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

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

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

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


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

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

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

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

```

```

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

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

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

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

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

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

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

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

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

    def update(self, _t):
        self.s = self.integral(self.s, _t, self.tau_decay)
        for i in range(self.pre.size[0]):
            self.g[i] = self.g[i] * (1 - self.s[i])

```

```

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

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

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

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

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

```

```

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

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

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

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

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

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

```

1.1 生物背景

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

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

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

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

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

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

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

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

```

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

syn_A2non_AMPA.g_max = g_max_E2E_AMPA
syn_A2non_NMDA.g_max = g_max_E2E_NMDA

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

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

syn_B2non_AMPA.g_max = g_max_E2E_AMPA
syn_B2non_NMDA.g_max = g_max_E2E_NMDA

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

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

syn_non2non_AMPA.g_max = g_max_E2E_AMPA
syn_non2non_NMDA.g_max = g_max_E2E_NMDA

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

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

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

```

```

        delay=delay_syn)

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

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

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

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

```

1.1 生物背景

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

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

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

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

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

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

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

1.1 生物背景

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

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

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

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

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

1.1 生物背景

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

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

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

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

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

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

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

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

plt.show()
```

发放率神经网络

抉择模型

```

from collections import OrderedDict
import brainpy as bp

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

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

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

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

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

    return ds1dt, ds2dt

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

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

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

```

```

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

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

```

```

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

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

```

```

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

```

```

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

```

1.1 生物背景

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

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

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

连续吸引子模型 (CANN)

```

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

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

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

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

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

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

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

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

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

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

```

```

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

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

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

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

```

```

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

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

```

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

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

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

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

plot_animate()
```

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

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

plot_animate()
```