

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 脉冲神经网络	1.4.1
3.2 发放率神经网络	1.4.2

BrainPy介绍

本章将介绍计算神经科学中的一系列神经元模型、突触模型和网络模型。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[神经元模型](#), [突触模型](#), and [网络模型](#)的顺序进行介绍。

BrainPy的文档及更多细节请参考我们的Github仓库：<https://github.com/PKU-NIP-Lab/BrainPy>和<https://github.com/PKU-NIP-Lab/BrainModels>。

1. 神经元模型

按照从繁到简的顺序，我们可以将神经元模型主要分为三类：生理模型、简化模型和发放率模型。

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。

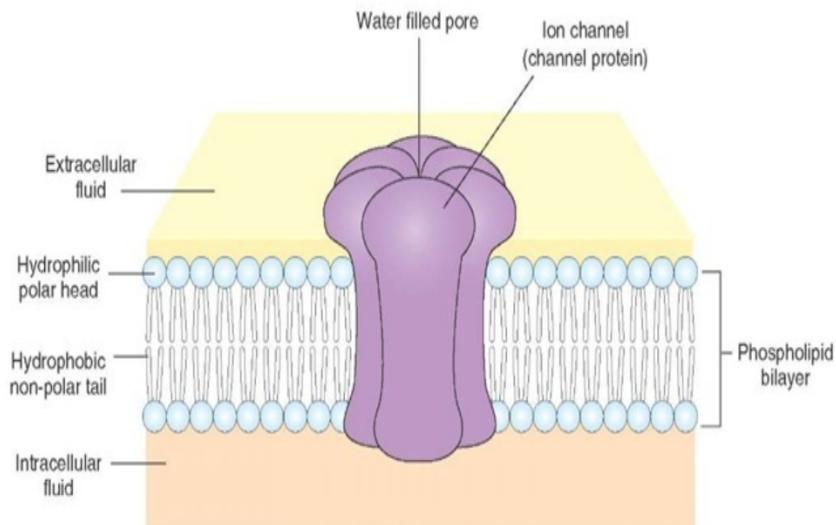


Fig. 1-1 Neuron membrane diagram | what-when-how.com

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。

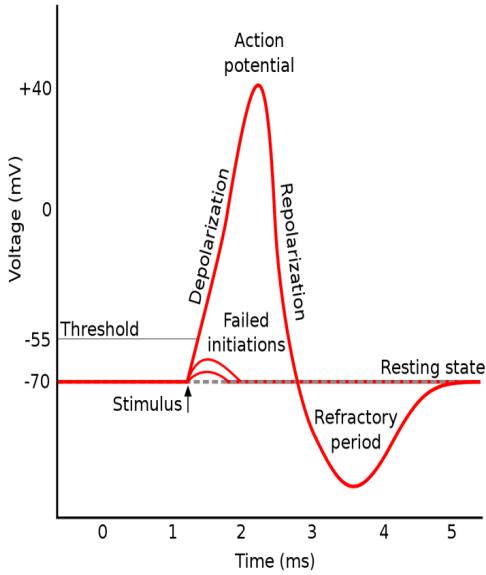
**Fig. 1-2 Action Potential | Wikipedia**

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

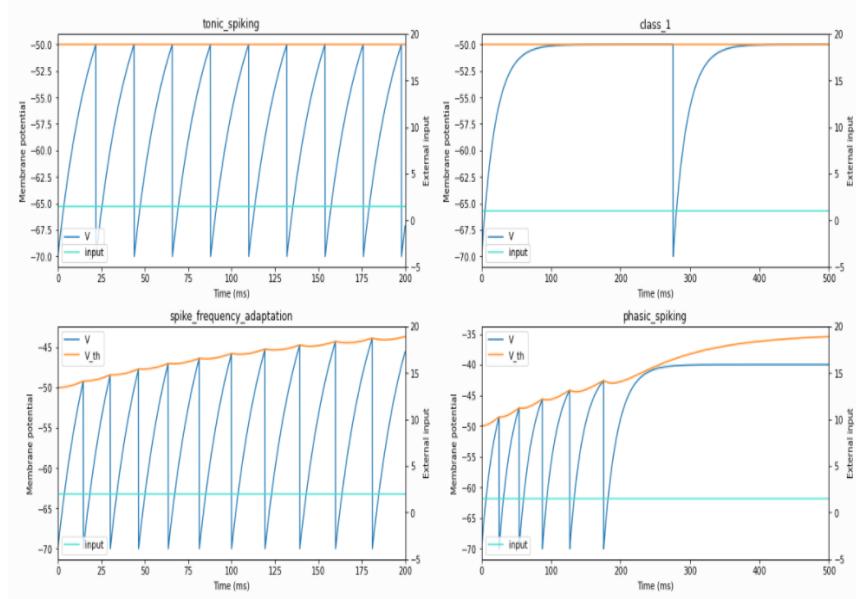


Figure 1-3 Some firing patterns

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。

1.2 生理模型

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细胞膜的一般性模板。HH模型中将神经元细胞膜建模为等效电路，如下图所示。

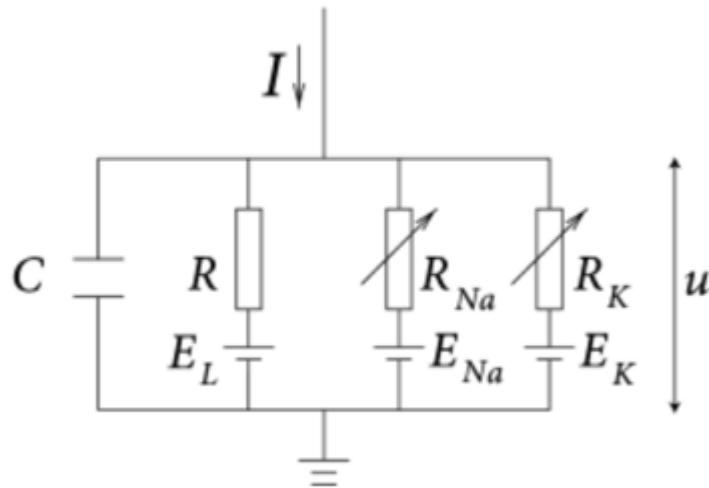


Fig. 1-4 Equivalent circuit diagram | NeuroDynamics

图1-4中所示的，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。

在图1-4中，电容 C 表示电导率很低的疏水性磷脂双层膜，电流 I 表示外界刺激。

由于钠离子通道和钾离子通道在动作电位的形成中非常重要，这两个离子通道被单独建模为电路图右侧所示的两个并联的可变电阻 R_{Na} 和 R_K ，而电阻 R 代表膜上所有非特定的离子通道。电源 E_{Na} , E_K 和 E_L 对应着由相应离子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：

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

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

这就是HH模型。注意在如上的第1个方程中，右侧的前3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 $I(t)$ 表示一个外部输入。在方程左侧， $C \frac{dV}{dt} = \frac{dQ}{dt} = I$ 是穿过电容的电流。

在计算经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 来控制离子通道的打开/关闭状态。准确的说，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，并且，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。

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。 α_x 和 β_x 的公式（如下所示）是由实验数据拟合得到的。

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

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

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

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

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

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

```

1   class HH(bp.NeuGroup): → bp.NeuGroup class:
2       target_backend = 'general' → Group of neurons
3   → Set backend for model.
4       @staticmethod → Call `bp.odeint` to integrate ODEs.
5       @bp.odeint(method='exponential_euler') → Set parameter 'method' to choose
6       def integral(V, m, h, n, t, C, gNa, ENa, gK, EK, gL, EL, Iext): numerical integration methods.
7           alpha_m = 0.1*(V+40)/(1-bp.ops.exp(-(V+40)/10)) α_m(V) = (0.1(V + 40))/(1 - exp(-(V + 40)/10))
8           beta_m = 4.0*bp.ops.exp(-(V+65)/18) β_m(V) = 4.0exp(-(V + 65)/18)
9           dmdt = alpha_m * (1 - m) - beta_m * m dx/dt = α(1 - x) - βx, x = m
10
11          alpha_h = 0.07*bp.ops.exp(-(V+65)/20) α_h(V) = 0.07 exp(-(V + 65)/20)
12          beta_h = 1/(1+bp.ops.exp(-(V+35)/10)) β_h(V) = 1/(1 + exp(-(V + 35)/10))
13          dhdt = alpha_h * (1 - h) - beta_h * h dx/dt = α(1 - x) - βx, x = h
14
15          alpha_n = 0.01*(V+55)/(1-bp.ops.exp(-(V+55)/10)) α_n(V) = (0.01(V + 55))/(1 - exp(-(V + 55)/10))
16          beta_n = 0.125*bp.ops.exp(-(V+65)/80) β_n(V) = 0.125exp(-(V + 65)/80)
17          dndt = 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 - ELeak)) + I(t)
21          I_leak = gL * (V - EL)
22          dVdt = (- I_Na - I_K - I_leak + Iext) / C
23
24      return dVdt, dmdt, dhdt, dnkt

```

1.1 生物背景

```

25
26     def __init__(self, size, ENa=50., gNa=120., EK=-77., gK=36.,
27                  EL=-54.387, gL=0.03, V_th=20., C=1.0, **kwargs):
28         # parameters
29         self.ENa = ENa
30         self.EK = EK
31         self.EL = EL
32         self.gNa = gNa
33         self.gK = gK
34         self.gL = gL
35         self.C = C
36         self.V_th = V_th
37
38         # variables
39         num = bp.size2len(size)
40         self.V = -65. * bp.ops.ones(num)
41         self.m = 0.5 * bp.ops.ones(num)
42         self.h = 0.6 * bp.ops.ones(num)
43         self.n = 0.32 * bp.ops.ones(num)
44         self.spike = bp.ops.zeros(num, dtype=bool)
45         self.input = bp.ops.zeros(num)
46
47     super(HH, self).__init__(size=size, **kwargs) → Pass `size` and `**kwargs` to
48
49     def update(self, _t):
50         V, m, h, n = self.integral(self.V, self.m, self.h, self.n, _t,           Update variables with
51                                         self.C, self.gNa, self.ENa, self.gK, → numerical integration
52                                         self.EK, self.gL, self.EL, self.input)   in vector form.
53         self.spike = (self.V < self.V_th) * (V >= self.V_th) → Judge if the neuron spikes.
54         self.V = V
55         self.m = m
56         self.h = h
57         self.n = n
58         self.input[:] = 0 → Reset external input
                                         for this time step.

```

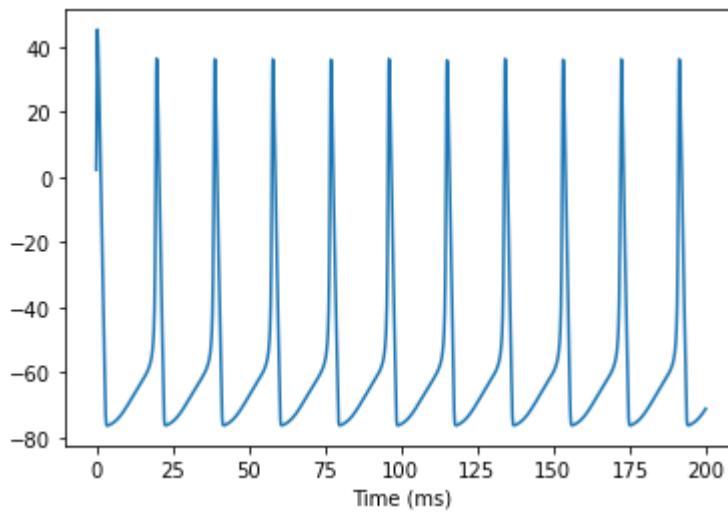
Model parameters saved as floating point numbers.

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

ENa	EK	...	V_th
V[0]	V[1]	...	V[100]
m[0]	m[1]	...	m[100]
...

在我们的github仓库中运行代码：<https://github.com/PKU-NIP-Lab/BrainModels>
(如无特殊说明, 下同)

BrainPy仿真的HH模型的V-t图如下所示。真实神经元产生动作电位的三个阶段，去极化、复极化和不应期都可以对应到下图中。另外在去极化时，HH模型的膜电位先是缓慢积累外部输入，一旦其值高于某个特定值，膜电位就转为快速增长，这也复现了真实动作电位的形状。



1.3 简化模型

启发自生理实验的Hodgkin-Huxley模型准确但昂贵。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的消耗。

简化模型简单、易于计算，并且它们仍然能够复现神经元发放的主要特征。尽管它们的表示能力不如生理模型，但和它们的简便相比，在特定场景下研究者们有时也可以接受一定的精度损失。

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,

$$V \leftarrow V_{reset}$$

$\tau = RC$ 是LIF模型的时间常数， τ 越大，模型的动力学就越慢。如上所示的方程对应于一个比HH模型的等效电路图更加简单的等效电路，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E和外部输入I被建模。

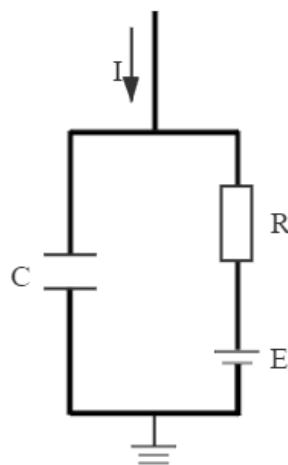


Fig1-4 Equivalent circuit of LIF model

比起HH模型，LIF模型没有建模动作电位的形状，也就是说，在发放一个峰电位之前，LIF神经元的膜电位不会骤增。并且在原始模型中，不应期也被忽视了。为了仿真模拟不应期，必须再补充一个条件判断：

如果

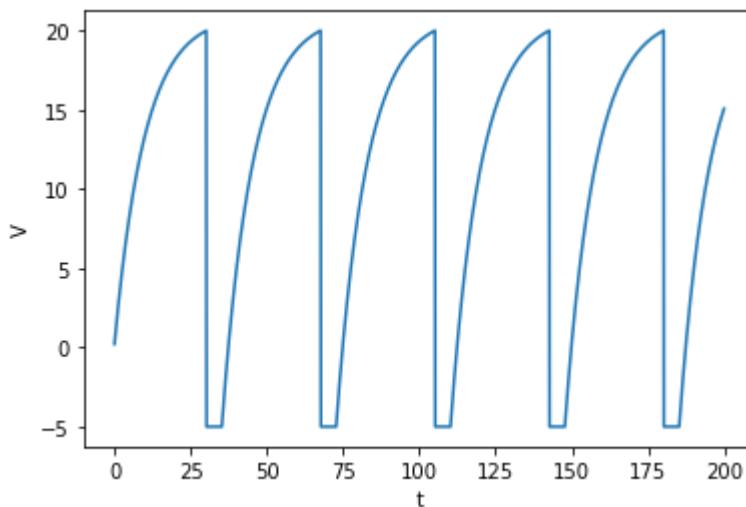
$$t - t_{lastspike} \leq refractoryperiod$$

则神经元处在不应期中，膜电位 V 不再更新。

```

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

```



1.3.2 二次积分-发放模型

为了追求更强的表示能力，Latham等人（2000）提出了**二次积分-发放模型**

(Quadratic Integrate-and-Fire model, **QualF model**)，他们在微分方程的右侧添加了一个二阶项，使得神经元能产生更好的动作电位。

1.1 生物背景

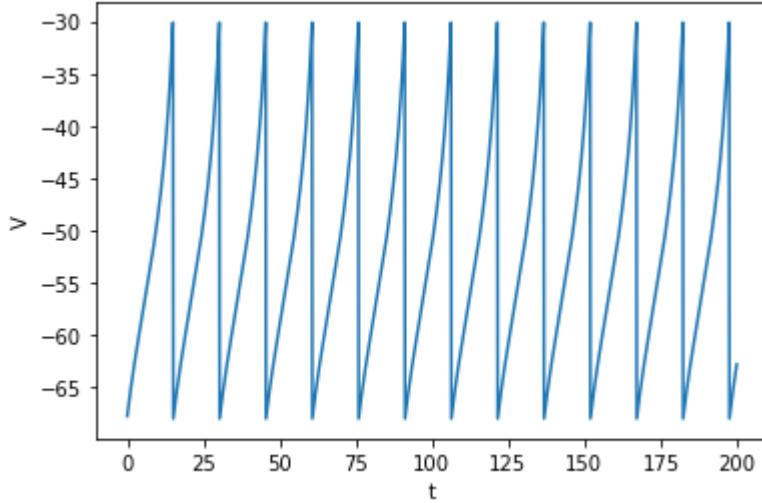
$$\tau \frac{dV}{dt} = a_0(V - V_{rest})(V - V_c) + RI(t)$$

在上式中， a_0 是控制着膜电位发放前的斜率的参数， V_c 是动作电位初始化的临界值。当低于 V_C 时，膜电位 V 缓慢增长，一旦越过 V_C ， V 就转为迅速增长。

```

1   class QuaIF(bp.NeuGroup): ────────── bp.NeuGroup class:
2       target_backend = 'general'
3
4       @staticmethod
5       def derivative(V, t, I_ext, V_rest, V_c, R, tau, a_0):
6           dVdt = (a_0 * (V - V_rest) * (V - V_c) + R * I_ext) / tau ───→ τ  $\frac{dV}{dt}$  =  $a_0(V - V_{rest})(V - V_c) + RI(t)$ 
7           return dVdt
8
9       def __init__(self, size, V_rest=-65., V_reset=-68.,
10                  V_th=-30., V_c=-50.0, a_0=.07,
11                  R=1., tau=10., t_refractory=0., **kwargs):
12           # parameters
13           self.V_rest = V_rest
14           self.V_reset = V_reset
15           self.V_th = V_th
16           self.V_c = V_c
17           self.a_0 = a_0
18           self.R = R
19           self.tau = tau
20           self.t_refractory = t_refractory
21
22           # variables
23           num = bp.size2len(size)
24           self.V = bp.ops.ones(num) * V_reset
25           self.input = bp.ops.zeros(num)
26           self.spike = bp.ops.zeros(num, dtype=bool)
27           self.refractory = bp.ops.zeros(num, dtype=bool)
28           self.t_last_spike = bp.ops.ones(num) * -1e7
29
30           self.integral = bp.odeint(f=self.derivative, method='euler') ───→ Call 'bp.odeint' to integrate ODEs.
31           super(QuaIF, self).__init__(size=size, **kwargs) ───→ Set parameter 'method' to choose
32                                         numerical integration methods.
33
34       def update(self, _t):
35           for i in range(self.size[0]): ────────── For each neuron in neuron group.
36               spike = 0.
37               refractory = (_t - self.t_last_spike[i] <= self.t_refractory) ───→ Check if neuron is
38               if not refractory:                                         in refractory period.
39                   V = self.integral(self.V[i], _t, self.input[i],
40                           self.V_rest, self.V_c, self.R,                                Update variables
41                           self.tau, self.a_0)                                         with numerical integration
42                   spike = (V >= self.V_th) ────────── Check if neuron spikes.
43                   if spike:                                                 Reset neuron.
44                       V = self.V_rest
45                       self.t_last_spike[i] = _t
46                       self.V[i] = V
47                       self.spike[i] = spike
48                       self.refractory[i] = refractory or spike
49                       self.input[i] = 0.

```



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) 的表示能力比QualIF模型更强。ExpIF模型在微分方程右侧增加了指数项，使得模型现在可以产生更加真实的动作电位。

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

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等同于 $V_{th} = V_T$ 的LIF模型 (Fourcaud-Trocme et al., 2003)。

```

1  class ExpIF(bp.NeuGroup):
2      target_backend = 'general'                                bp.NeuGroup class:
3
4      @staticmethod
5      def derivative(V, t, I_ext, V_rest, delta_T, V_T, R, tau):
6          exp_term = bp.ops.exp((V - V_T) / delta_T)
7          dvdt = (- (V - V_rest) + delta_T * exp_term + R * I_ext) / tau
8          return dvdt
9
10     def __init__(self, size, V_rest=-68., 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.t_refractory = t_refractory
23
24         # variables
25         self.V = bp.ops.ones(size) * V_rest
26         self.input = bp.ops.zeros(size)
27         self.spike = bp.ops.zeros(size, dtype=bool)
28         self.refractory = bp.ops.zeros(size, dtype=bool)
29         self.t_last_spike = bp.ops.ones(size) * -1e7
30
31         self.integral = bp.odeint(self.derivative)           Call 'bp.odeint' to integrate ODEs.
32         super(ExpIF, self).__init__(size=size, **kwargs)       Parameter 'method' is set to
                                                               default value 'euler'.
33
34

```

Model parameters saved as floating point numbers.

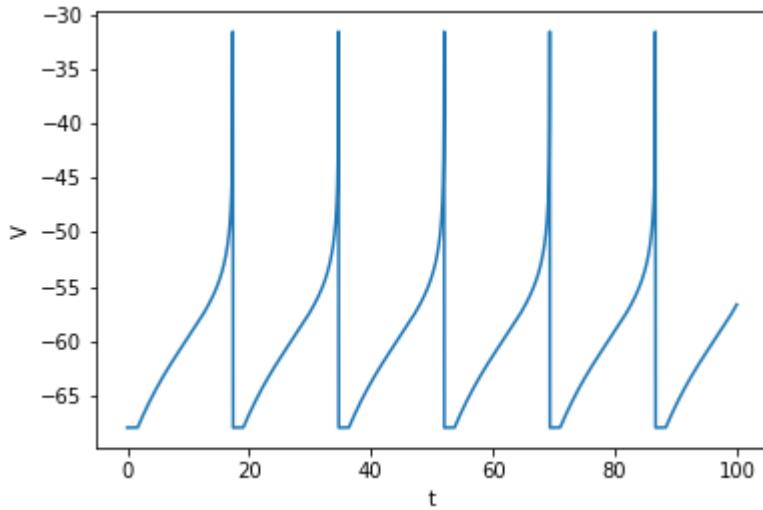
Model variables saved as vectors of floating point numbers.

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

```

35     def update(self, _t):
36         for i in prange(self.num):————— For each neuron in neuron group.
37             spike = 0.
38             refractory = (_t - self.t_last_spike[i] <= self.t_refractory) —— Check if neuron is
39             if not refractory:
40                 V = self.integral(
41                     self.V[i], _t, self.input[i], self.V_rest, —————— Update variables
42                     self.delta_T, self.V_T, self.R, self.tau —————— with numerical integration
43                 ) one by one.
44                 spike = (V >= self.V_th) —————— Check if neuron spikes.
45                 if spike:
46                     V = self.V_reset
47                     self.t_last_spike[i] = _t
48                     self.V[i] = V
49                     self.spike[i] = spike
50                     self.refractory[i] = refractory or spike
51             self.input[:] = 0.

```



1.3.4 适应性指数积分-发放模型

当面对恒定的外部刺激时，神经元一开始高频发放，随后发放率逐渐降低，最终稳定在一个较小值，这种现象生物上称为**适应**。

为了复现神经元的适应行为，研究者们在已有的积分-发放模型，如LIF、QuaIF和ExpIF模型上增加了权重变量w。这里我们介绍其中一个经典模型，**适应性指数积分-发放模型** (Adaptive Exponential Integrate-and-Fire model, **AdExIF model**) (Gerstner et al., 2014)。

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

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

就如它的名字所示，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，不同的是适应项，即方程中 $-Rw$ 这一项。

权重项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，并且w也会随时间衰减。

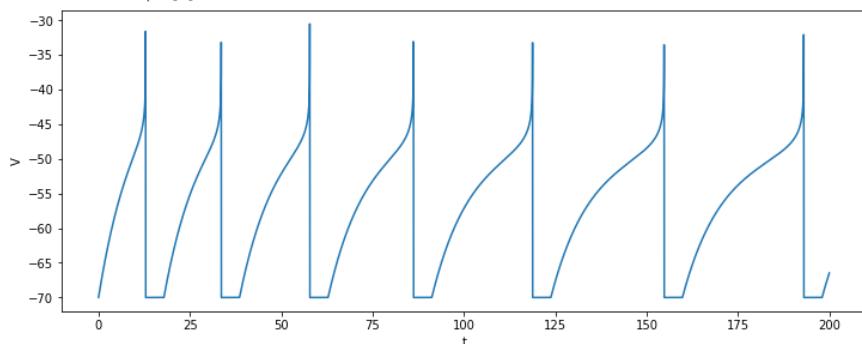
给神经元一个恒定输入，在连续发放几个动作电位之后，w的值将会上升到一个高点，减慢V的增长速度，从而降低神经元的发放率。

1.1 生物背景

```

1 class AdExIF(bp.NeuGroup): → bp.NeuGroup class:
2     target_backend = 'general' → Group of neurons
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
29
30     # variables
31     num = bp.size2len(size)
32     self.V = bp.ops.ones(num) * V_reset
33     self.w = bp.ops.zeros(size)
34     self.input = bp.ops.zeros(num)
35     self.spike = bp.ops.zeros(num, dtype=bool)
36     self.refractory = bp.ops.zeros(num, dtype=bool)
37     self.t_last_spike = bp.ops.ones(num) * -1e7
38
39     self.integral = bp.odeint(f=self.derivative, method='euler') → Call `bp.odeint` to integrate ODEs.
40
41     super(AdExIF, self).__init__(size=size, **kwargs) → Set parameter `method` to choose
42
43     def update(self, _t):
44         for i in prange(self.size[0]): → For each neuron in neuron group.
45             spike = 0.
46             refractory = (_t - self.t_last_spike[i] <= self.t_refractory) → Check if neuron is
47             if not refractory: → in refractory period.
48                 V, w = self.integral(self.V[i], self.w[i], _t, self.input[i], → Update variables
49                                         self.V_rest, self.delta_T, → with numerical integration
50                                         self.V_T, self.R, self.tau, self.tau_w, self.a) → one by one.
51                 spike = (V >= self.V_th) → Check if neuron spikes.
52                 if spike:
53                     V = self.V_rest
54                     w += self.b
55                     self.t_last_spike[i] = _t → Reset membrane potential,
56                     self.V[i] = V → add the Dirac δ term of w:
57                     self.w[i] = w →  $\tau_w \frac{dw}{dt} = a(V - V_{rest}) - w + b\tau_w \sum \delta(t - t^f)$ 
58                     self.spike[i] = spike
59                     self.refractory[i] = refractory or spike
60                     self.input[i] = 0.

```



1.3.5 Hindmarsh-Rose模型

为了模拟神经元中的**爆发式发放** (bursting, 即在短时间内的连续发放) , Hindmarsh和Rose (1984) 提出了**Hindmarsh-Rose模型**, 引入了第三个模型变量 z 作为慢变量来控制神经元的爆发。

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

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

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

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

The parameter b in $\frac{dV}{dt}$ equation allows the model to switch between spiking and bursting states, and controls the spiking frequency. r controls slow variable z 's variation speed, affects the number of spikes per burst when bursting, and governs the spiking frequency together with b . The parameter s governs adaptation, and other parameters are fitted by firing patterns.

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 允许模型在发放和爆发两个状态之间切换，并且控制着发放的频率。参数 r 控制着慢变量 z 的变化速度，影响着神经元爆发式发放时，每次爆发包含的动作电位个数，并且和 b 一起统筹控制发放频率，参数 s 控制着适应行为。其它参数根据发放模式拟合得到。

```

1  class HindmarshRose(bp.NeuGroup): → bp.NeuGroup class:
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 - aV3 + bV2 - z + I
7          dydt = c - d * V * V - y                           dy/dt = c - dV2 - y
8          dzdt = r * (s * (V - V_rest) - z)                  dz/dt = r(s(V - Vrest) - z)
9          return dVdt, dydt, dzdt
10
11     def __init__(self, size, a=1., b=3.,
12                 c=1., d=5., r=0.01, s=4.,
13                 V_rest=-1.6, **kwargs):
14         # parameters
15         self.a = a
16         self.b = b
17         self.c = c
18         self.d = d
19         self.r = r
20         self.s = s
21         self.V_rest = V_rest
22
23         # variables
24         num = bp.size2len(size)
25         self.z = bp.ops.zeros(num)
26         self.input = bp.ops.zeros(num)
27         self.V = bp.ops.ones(num) * -1.6
28         self.y = bp.ops.ones(num) * -10.
29         self.spike = bp.ops.zeros(num, dtype=bool)
30
31         self.integral = bp.odeint(f=self.derivative) → Call `bp.odeint` to integrate ODEs.
32         super(HindmarshRose, self).__init__(size=size, **kwargs) → Parameter `method` is set to
33
34     def update(self, _t): → default value `euler`.
35         for i in prange(self.num): → Pass `size` and `**kwargs` to
36             V, self.y[i], self.z[i] = self.integral(           superclass bp.NeuGroup's constructor.
37             self.V[i], self.y[i], self.z[i], _t,           → For each neuron in neuron group.
38             self.a, self.b, self.input[i],           Update variables with
39             self.c, self.d, self.r, self.s,           numerical integration
40             self.V_rest)           one by one.
41             self.V[i] = V
42             self.input[i] = 0. → Reset external input
43

```

Model parameters saved as floating point numbers.

Model variables saved as vectors of floating point numbers.

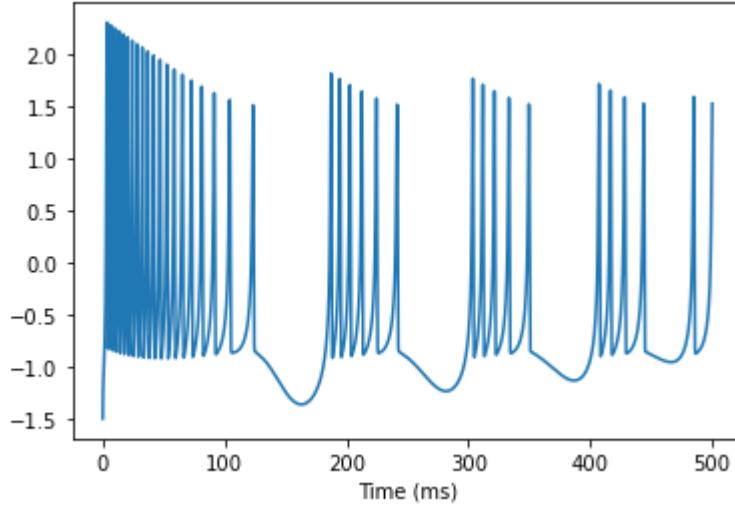
→ Pass `size` and `**kwargs` to

superclass bp.NeuGroup's constructor.

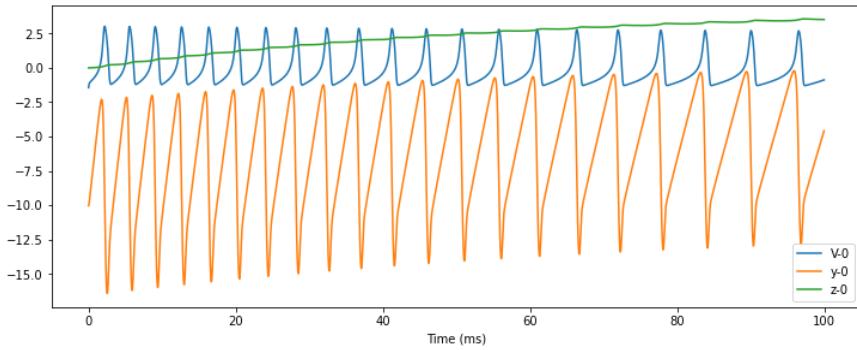
→ For each neuron in neuron group.

Update variables with numerical integration one by one.

→ Reset external input for this time step.



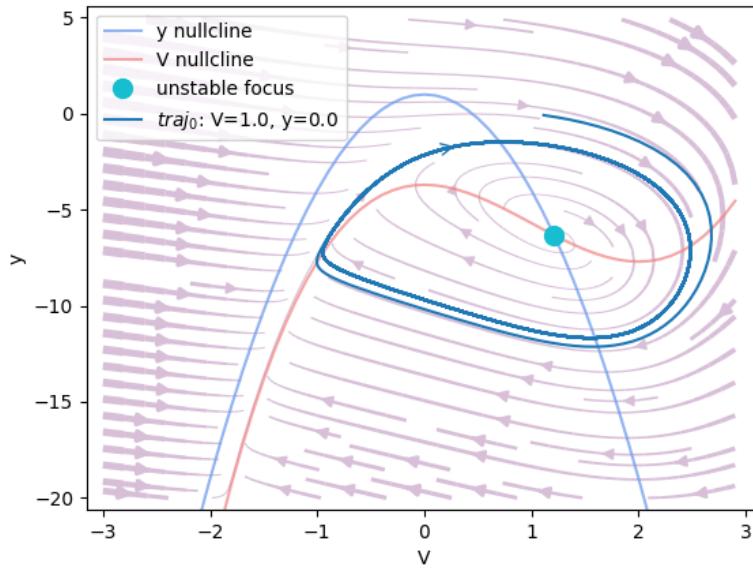
在下图中，画出了三个变量随时间的变化，可以看到慢变量 z 的改变要慢于 V 和 y 。而且， V 和 y 在仿真过程中呈周期性变化。



利用BrainPy的理论分析模块 `analysis`，我们可以分析出这种周期性的产生原因。在模型的相图中， V 和 y 的轨迹趋近于一个极限环，因此他们的值会沿着极限环发生周期性的改变。

```

68 # Phase plane analysis
69 phase_plane_analyzer = bp.analysis.PhasePlane(
70     neu.integral,                                     ┏━━━ Dynamic system to be analyzed.
71     target_vars={'V': [-3., 3.], 'y': [-20., 5.]},   ┏━━━ Variables to be showed in phase plane.
72     fixed_vars={'z': 0.},                            ┏━━━ Variables to be fixed in phase plane.
73     pars_update={'I_ext': param[mode][1], 'a': 1., 'b': 3., } ┏━ Other parameters to be fixed.
74         'c': 1., 'd': 5., 'r': 0.01, 's': 4.,           └━
75         'V_rest': -1.6}                                └━
76 )
77 phase_plane_analyzer.plot_nullcline() ┏━━━ Plot nullcline.
78 phase_plane_analyzer.plot_fixed_point() ┏━━━ Plot fixed points.
79 phase_plane_analyzer.plot_vector_field() ┏━━━ Plot vector field.
80 phase_plane_analyzer.plot_trajectory(          ┏━ Plot trajectory.
81     [{'V': 1., 'y': 0., 'z': -0.}],             ┏━ Define start point of trajectory and
82     duration=100.,                             ┏━ simulation duration.
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。该模型拥有四个模型变量，能产生多于20种发放模式，并可以通过调整参数在各模式之间切换。

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

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

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

当 V 达到 V_{th} 时，GeneralizedIF模型发放：

$$I_j \leftarrow R_j I_j + A_j$$

$$V \leftarrow V_{reset}$$

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

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 描述了 I_j 的重置值对发放前的 I_j 的值的依赖， A_j 是在发放后加到 I_j 上的一个常数值。

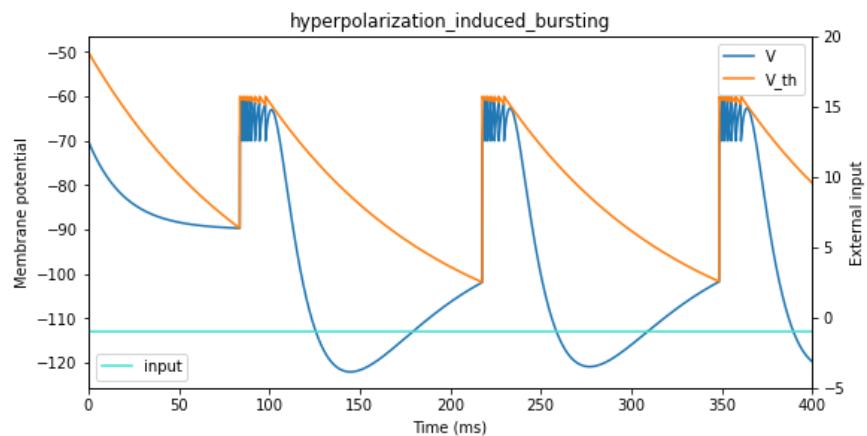
可变的阈值电位 V_{th} 受两个参数的调控： a 描述了 V_{th} 对膜电位 V 的依赖， b 描述了

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  class GeneralizedIF(bp.NeuGroup): → bp.NeuGroup class:
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                                 $dI_1/dt = -k_1 I_1$ 
9          dI2dt = - k2 * I2                                 $dI_2/dt = -k_2 I_2$ 
10         dVthdt = a * (V - V_rest) - b * (V_th - V_th_inf)  $dV_{th}/dt = a(V - V_{rest}) - b(V_{th} - V_{th\infty})$ 
11         dVdt = (- (V - V_rest) + R * I_ext + R * I1 + R * I2) / tau  $\tau dV/dt = -(V - V_{rest}) + R \sum I_j + RI(t),$ 
12         return dI1dt, dI2dt, dVthdt, dVdt                 $j = 1, 2$ 
13
14     def __init__(self, size, V_rest=-70., V_reset=-70.,
15                  V_th_inf=-50., V_th_reset=-60., R=20., tau=20.,
16                  a=0., b=0.01, k1=0.2, k2=0.02,
17                  R1=0., R2=1., A1=0., A2=0.,
18                  **kwargs):
19         # params
20         self.V_rest = V_rest
21         self.V_reset = V_reset
22         self.V_th_inf = V_th_inf
23         self.V_th_reset = V_th_reset
24         self.R = R
25         self.tau = tau
26         self.a = a
27         self.b = b
28         self.k1 = k1
29         self.k2 = k2
30         self.R1 = R1
31         self.R2 = R2
32         self.A1 = A1
33         self.A2 = A2
34
35         # vars
36         self.input = bp.ops.zeros(size)
37         self.spike = bp.ops.zeros(size, dtype=bool)
38         self.I1 = bp.ops.zeros(size)
39         self.I2 = bp.ops.zeros(size)
40         self.V = bp.ops.ones(size) * -70.
41         self.V_th = bp.ops.ones(size) * -50.
42
43         self.integral = bp.odeint(self.derivative) → Call `bp.odeint` to integrate ODEs.
44         super(GeneralizedIF, self).__init__(size=size, **kwargs) → Parameter `method` is set to
45                                         default value `euler`. → Pass `size` and `**kwargs` to
46                                         superclass bp.NeuGroup's constructor.
46
47     def update(self, _t):
48         for i in prange(self.size[0]): → For each neuron in neuron group.
49             I1, I2, V_th, V = self.integral(→ Update variables with
50                 self.I1[i], self.I2[i], self.V_th[i], self.V[i], _t, numerical integration
51                 self.k1, self.k2, self.a, self.V_rest, one by one.
52                 self.b, self.V_th_inf,
53                 self.R, self.input[i], self.tau)
54
55             self.spike[i] = self.V_th[i] < V → Check if neuron spikes.
56             if self.spike[i]: → Reset membrane potential
57                 V = self.V_reset → and threshold potential,
58                 I1 = self.R1 * I1 + self.A1 → update I1, I2.
59                 I2 = self.R2 * I2 + self.A2 →  $V = V_{reset}$ 
60                 V_th = max(V_th, self.V_th_reset) →  $I_1 = R_1 I_1 + A_1$ 
61                 self.I1[i] = I1 →  $I_2 = R_2 I_2 + A_2$ 
62                 self.I2[i] = I2 →  $V_{th} = \max(V_{th}, V_{threset})$ 
63                 self.V_th[i] = V
64                 self.V[i] = V
65                 self.f = 0. → Reset external input
66                 self.input[:] = self.f → for this time step.
```

1.1 生物背景



1.4 放率模型

放率模型比简化模型更加简单。在这些模型中，每个计算单元代表一个神经元群，而单神经元模型中的膜电位变量 V 也被放率变量 a （或 r 或 ν ）所取代。下面我们将介绍一个经典的放率单元。

1.4.1 放率单元

Wilson和Cowan (1972) 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复数神经元的皮层微柱中神经元群的平均活动水平。

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

下标 $x \in \{e, i\}$ 表示该参数或变量对应着兴奋性还是抑制性的神经元群。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 分别代表Sigmoid函数 $\mathcal{S}(input)$ 的斜率和相位参数，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):
2      target_backend = 'general'
3
4      @staticmethod
5      def derivative(a_e, a_i, t,
6                      k_e, r_e, c1, c2, I_ext_e, slope_e, theta_e, tau_e,
7                      k_i, r_i, c3, c4, I_ext_i, slope_i, theta_i, tau_i):
8          x_ae = c1 * a_e - c2 * a_i + I_ext_e
9          sigmoid_ae_l = 1 / (1 + bp.ops.exp(-slope_e * (x_ae - theta_e)))
10         sigmoid_ae_r = 1 / (1 + bp.ops.exp(slope_e * theta_e))
11         sigmoid_ae = sigmoid_ae_l - sigmoid_ae_r
12         daedt = (- a_e + (k_e - r_e * a_e) * sigmoid_ae) / tau_e
13
14         x_ai = c3 * a_e - c4 * a_i + I_ext_i
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

```

1.1 生物背景

```

1  def __init__(self, size, c1=12., c2=4., c3=13., c4=11.,
2      k_e=1., k_i=1., tau_e=1., tau_i=1., r_e=1., r_i=1.,
3      slope_e=1.2, slope_i=1., theta_e=2.8, theta_i=4.,
4      **kwargs):
5          # params
6          self.c1 = c1
7          self.c2 = c2
8          self.c3 = c3
9          self.c4 = c4
10         self.k_e = k_e
11         self.k_i = k_i
12         self.tau_e = tau_e
13         self.tau_i = tau_i
14         self.r_e = r_e
15         self.r_i = r_i
16         self.slope_e = slope_e
17         self.slope_i = slope_i
18         self.theta_e = theta_e
19         self.theta_i = theta_i
20
21     # vars
22     self.input_e = bp.backend.zeros(size)
23     self.input_i = bp.backend.zeros(size)
24     self.a_e = bp.backend.ones(size) * 0.1
25     self.a_i = bp.backend.ones(size) * 0.05
26
27     self.integral = bp.odeint(self.derivative)
28
29     super(FiringRateUnit, self).__init__(size=size, **kwargs)
30
31     def update(self, _t):
32         self.a_e, self.a_i = self.integral(
33             self.a_e, self.a_i, _t,
34             self.k_e, self.r_e, self.c1, self.c2,
35             self.input_e, self.slope_e,
36             self.theta_e, self.tau_e,
37             self.k_i, self.r_i, self.c3, self.c4,
38             self.input_i, self.slope_i,
39             self.theta_i, self.tau_i)
40
41         self.input_e[:] = 0.
42         self.input_i[:] = 0.
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

```

Model parameters saved as floating point numbers.

Model variables saved as vectors of floating point numbers.

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

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

Update variables with numerical integration in vector form.

Reset external input for this time step.

2. 突触模型

当我们建模了神经元的动作电位后，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。

在本章中，我们将在[2.1节](#)介绍包括化学突触与电突触的模型，同时涵盖从简单到复杂的各种突触动力学模型。

突触模型另一个重要的地方在于突触可塑性的实现，包括突触短时程与长时程可塑性，对于学习、记忆与神经网络的计算、训练都非常重要，我们将会在[2.2节](#)中介绍突触可塑性的部分。

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的动作电位，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对神经元之间的沟通进行模拟仿真。

2.1.1 化学突触

生物背景

我们可以从图2-1这个生物突触的图中看到神经元之间信息传递的过程。当突触前神经元的动作电位传递到轴突的末端（terminal），它会往突触间隙释放神经递质（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，这种改变成为突触后电位（PSP）。根据神经递质种类的不同，突触后电位可以是兴奋性或是抑制的。例如谷氨酸（Glutamate）就是一种重要兴奋性的神经递质，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（ Mg^{2+} ）堵住，无法对谷氨酸做出反应。由于镁离子对电压敏感，当AMPA导致突触后电位改变到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。

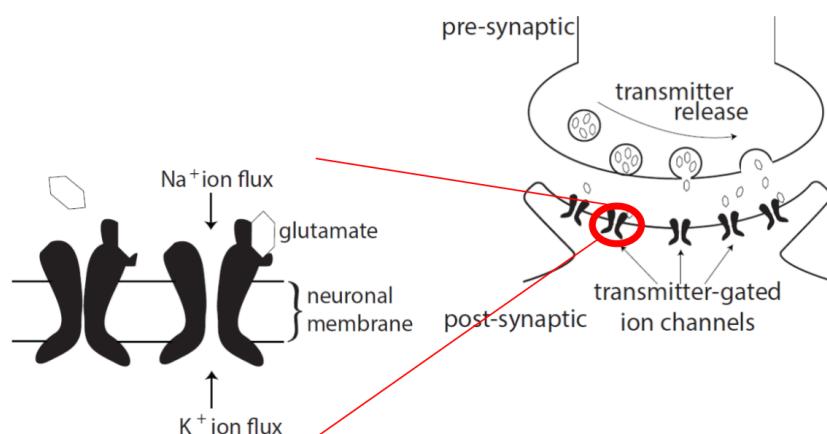


图 2-1 生物突触 (引自 Gerstner et al., 2014 [1](#))

为了简便地建模从神经递质释放到引起突触后电位的这个过程，我们可以把神经递质释放、递质与受体结合、受体引起的变化这些过程概括为突触前神经元的动作电位变化如何引起突触后神经元膜上的离子通道变化，即用门控变量 s 来描述每当突

触前神经元产生动作电位的时候，有多少比例的离子通道会被打开。我们首先来看看AMPA的例子。

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。

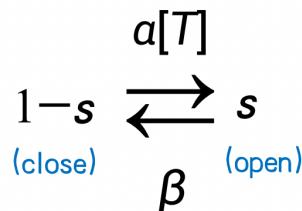


图2-2 离子通道动力学的马尔可夫过程

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； β 表示从 s 到 $(1 - s)$ 的转移概率。

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。

1.1 生物背景

```

1 import brainpy as bp
2
3
4 class AMPA(bp.TwoEndConn):           → bp.TwoEndConn class:
5     target_backend = ['numpy', 'numba']   Connections between two neuron groups
6
7     @staticmethod
8     def derivative(s, t, TT, alpha, beta):
9         ds = alpha * TT * (1 - s) - beta * s    }  $\frac{ds}{dt} = \alpha[T](1-s) - \beta s$ 
10        return ds
11
12     def __init__(self, pre, post, conn, alpha=0.98, beta=0.18, T=0.5,
13                  T_duration=0.5, **kwargs):
14         # parameters
15         self.alpha = alpha
16         self.beta = beta
17         self.T = T
18         self.T_duration = T_duration    } Regard [T] as a constant T, last for T_duration
19
20         # connections
21         self.conn = conn(pre.size, post.size)
22         self.pre_ids, self.post_ids = conn.requires('pre_ids', 'post_ids')   } Specify how the
23         self.size = len(self.pre_ids)                                         presynaptic and
24
25         # variables
26         self.s = bp.ops.zeros(self.size)
27         self.t_last_pre_spike = -1e7 * bp.ops.ones(self.size)                postsynaptic neuron
28
29         self.int_s = bp.odeint(f=self.derivative, method='exponential_euler')
30         super(AMPA, self).__init__(pre=pre, post=post, **kwargs)
31

```

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

pre ids	0	0	0	1
post ids	3	5	7	0
syn ids	0	1	2	3

然后我们在 `update` 函数中更新 `s`。

```

32     def update(self, _t):
33         for i in range(self.size):           → For each single synapse
34             pre_id = self.pre_ids[i]
35             post_id = self.post_ids[i]
36
37             if self.pre.spike[pre_id]:
38                 self.t_last_pre_spike[pre_id] = _t
39                 TT = ((_t - self.t_last_pre_spike[pre_id])           } TT denotes [T], TT=T if pre neuron
40                     < self.T_duration) * self.T                         spikes within T_duration,
41             self.s[i] = self.int_s(self.s[i], _t, TT, self.alpha, self.beta)   otherwise TT=0.
42

```

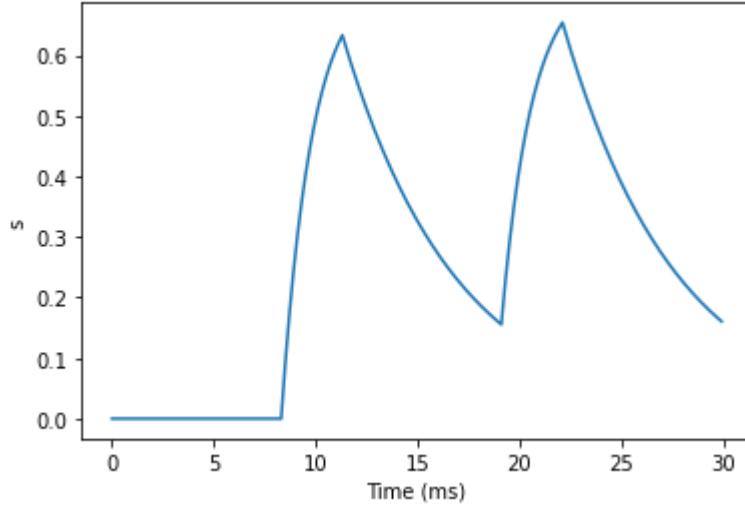
我们已经定义好了一个AMPA类，现在可以画出 `s` 随时间变化的图了。我们首先写一个 `run_syn` 函数来方便之后运行更多的突触模型，然后把AMPA类和需要自定义的变量传入这个函数来运行并画图。

```

43
44 import brainmodels as bm
45
46 bp.backend.set(backend='numba', dt=0.1)           → Set numba backend
47
48
49 def run_syn(syn_model, **kwargs):           → A method to run synapse
50     neu1 = bm.neurons.LIF(2, monitors=['V'])   } Get two LIF neuron groups from
51     neu2 = bm.neurons.LIF(3, monitors=['V'])   } brainmodels package
52
53     syn = syn_model(pre=neu1, post=neu2, conn=bp.connect.All2All(),
54                      monitors=['s'], **kwargs)           } Specify pre and post neurons and
55
56     net = bp.Network(neu1, syn, neu2)           } there connections. Here we use all to
57     net.run(30., inputs=(neu1, 'input', 35.))    all connections provided by BrainPy.
58     bp.visualize.line_plot(net.ts, syn.mon.s, ylabel='s', show=True)
59
60
61 run_syn(AMPA, T_duration=3.)           → Run AMPA synapse and
62                                         set parameter
63                                         T_duration to be 3.

```

运行以上代码，我们就会看到以下的结果：



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，而随着膜电位的变化，镁离子又会移开，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：

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

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

在此公式中， g_∞ 的值随着镁离子浓度增加而减小。而随着电压 V 增加， g_∞ 越来越不受镁离子的影响，建模了镁离子随电压增加而离开的效果。 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：

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

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

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。

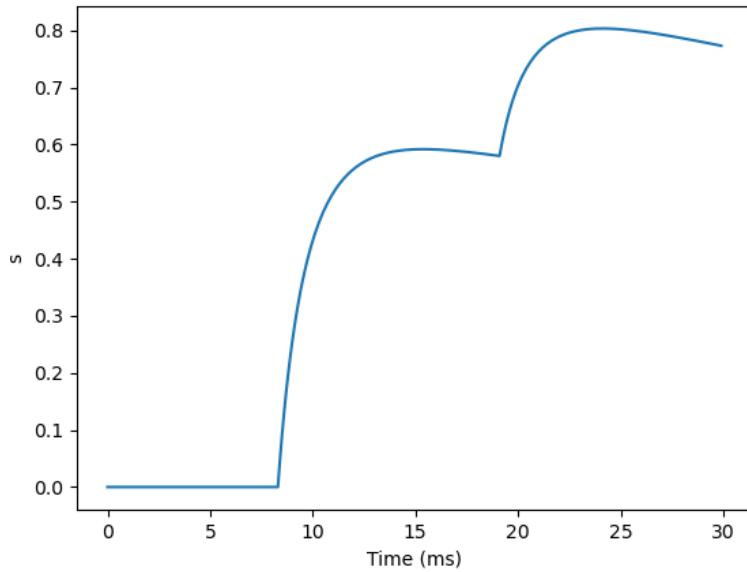
接下来我们用BrainPy来实现NMDA模型，代码如下。

1.1 生物背景

```
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)           }    $\frac{ds}{dt} = -s/\tau_{decay} + ax(1-s)$ 
10         dxdt = x / tau_rise                                }    $\frac{dx}{dt} = -x/\tau_{rise}$ 
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])   }    $g_\infty = (1 + e^{-\alpha V})^{-1}$ 
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)   }   I = g(V-E)
58             self.post.input[post_id] -= I_syn
59
```

由于前面我们已经定义了 `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：

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

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

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

$[R]$ 的动力学类似于AMPA模型中 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

```

1  class GABAAb(bp.TwoEndConn):
2      target_backend = ['numpy', 'numba']
3
4      @staticmethod
5      def derivative(R, G, t, k3, TT, k4, k1, k2):
6          dRdt = k3 * TT * (1 - R) - k4 * R
7          dGdt = k1 * R - k2 * G
8          return dRdt, dGdt
9
10     def __init__(self, pre, post, conn, delay=0., g_max=0.02, E=-95.,
11                  k1=0.18, k2=0.034, k3=0.09, k4=0.0012, kd=100., T=0.5,
12                  T_duration=0.3, **kwargs):
13         # params
14         self.g_max = g_max
15         self.E = E
16         self.k1 = k1
17         self.k2 = k2
18         self.k3 = k3
19         self.k4 = k4
20         self.kd = kd
21         self.T = T
22         self.T_duration = T_duration ] Regard [T] as a constant T, last for T_duration
23
24         # connns
25         self.conn = conn(pre.size, post.size)
26         self.pre_ids, self.post_ids = conn.requires('pre_ids', 'post_ids')
27         self.size = len(self.pre_ids)
28
29         # data
30         self.R = bp.ops.zeros(self.size)
31         self.G = bp.ops.zeros(self.size)
32         self.t_last_pre_spike = bp.ops.ones(self.size) * -1e7
33         self.s = 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         super(GABAAb, self).__init__(pre=pre, post=post, **kwargs)
39
40     def update(self, _t):
41         for i in range(self.size):
42             pre_id = self.pre_ids[i]
43             post_id = self.post_ids[i]
44
45             if self.pre.spike[pre_id]:
46                 self.t_last_pre_spike[i] = _t
47                 TT = ((_t - self.t_last_pre_spike[i]) < self.T_duration) * self.T ] TT denotes [T], TT=T
48
49                 self.R[i], G = self.integral(self.R[i], self.G[i], _t, self.k3,
50                                              TT, self.k4, self.k1, self.k2)
51                 self.s[i] = G ** 4 / (G ** 4 + self.kd) ——————> s = [G]^4 / ([G]^4 + K_d)
52                 self.G[i] = G
53
54                 self.g.push(i, self.g_max * self.s[i]) ——————> g = —s
55                 I_syn = self.g.pull(i) * (self.post.V[post_id] - self.E) ] I = g(V-E)
56                 self.post.input[post_id] -= I_syn
57

```

由于GABA_B也是非常缓慢的模型，这里我们不再用前面写的只有30ms模拟的 run_syn 函数，而是先给20ms的输入，接着看剩余1000ms在没有外界输入情况下的衰减。

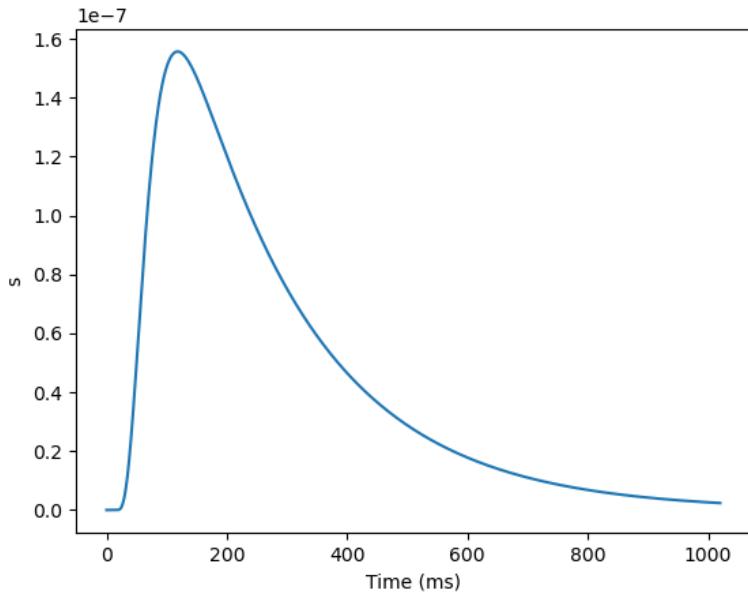
```

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

```



从结果可以看到，GABA_B的衰减确实非常慢，在移除外界输入之后几百ms内都还在衰减。

指数及Alpha模型

由于许多突触模型都有类似AMPA突触那样先上升后下降的动力学特征，有时候我们建模不需要具体对应到生物学上的突触，因此，有人提出了一些抽象的突触模型。这里，我们会介绍四种这类抽象模型在BrainPy上的实现。这些模型在 `BrainModels` 中也有现成的提供。

(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：

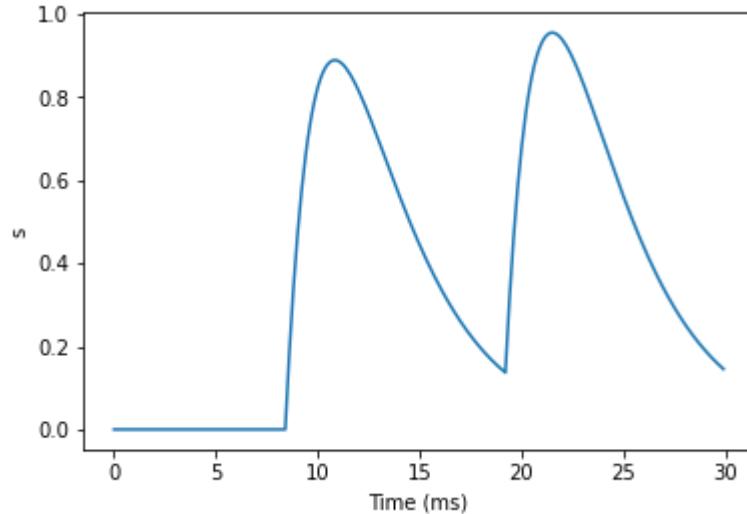
1.1 生物背景

```

1  class Two_exponentials(bp.TwoEndConn):
2      target_backend = ['numpy', 'numba']
3
4      @staticmethod
5      def derivative(s, x, t, tau1, tau2):
6          dxdt = 
$$\frac{dx}{dt} = \frac{-(\tau_1 + \tau_2)x - s}{(\tau_1\tau_2)}$$

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]
35
36             if (pre spike), then x = x + 1
37
38 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：

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

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

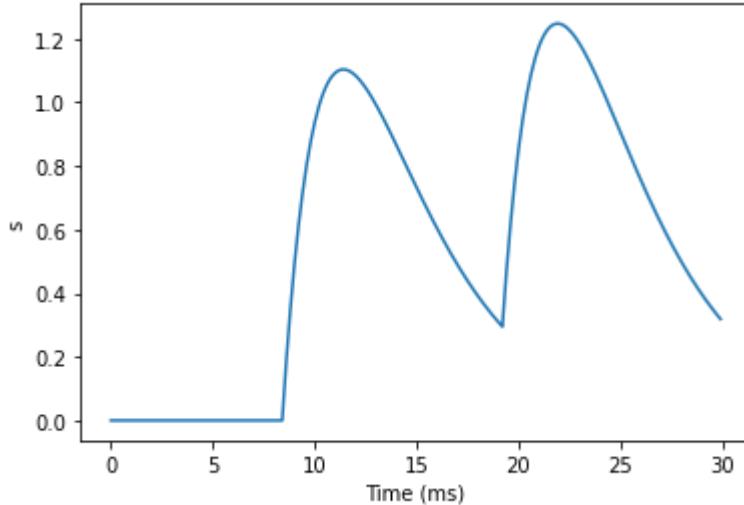
if (fire), then $x \leftarrow x + 1$

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

```

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

```



(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：

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

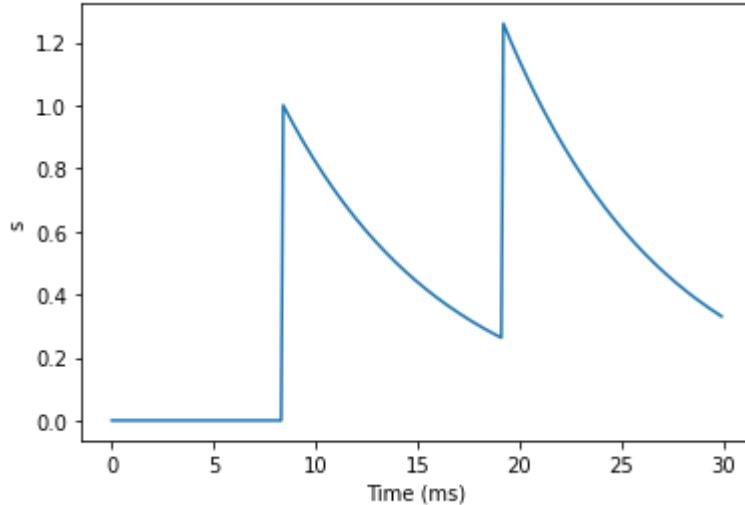
if (fire), then $s \leftarrow s + 1$

代码实现如下：

```

1  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：

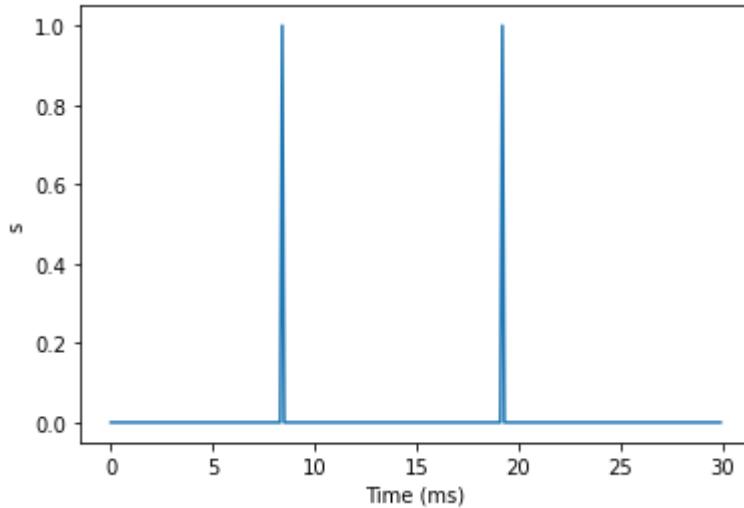
$$\text{if (fire), then } s \leftarrow s + 1$$

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)

```



基于电流的和基于电导的突触

目前为止，我们都在介绍门控变量 s 的动力学，现在让我们来看看突触电流如何变化。我们用 I 表示通过突触的电流，从 s 到 I 的关系有有两种不同的方法来建模，分别为 **基于电流 (current-based)** 与 **基于电导 (conductance-based)**。两者的主要区别在于突触电流是否受突触后神经元膜电位的影响。

(1) 基于电流 (Current-based)

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

```

(2) 基于电导 (Conductance-based)

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-65时，减去比它更低的 E ，例如-75，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0。

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) —— set delay time
7                                         before changing the
8                                         conductance
9     def update(self, _t):
10        for i in range(self.size):
11            # ...
12            self.g.push(i, self.g_max * self.s[i]) ——  $g = \bar{g}s$ 
13            self.post.input[post_id] -= self.g.pull(i) * (self.post.V[post_id] - self.E) ——  $I = g(V - E)$ 
14
15

```

2.1.2 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。

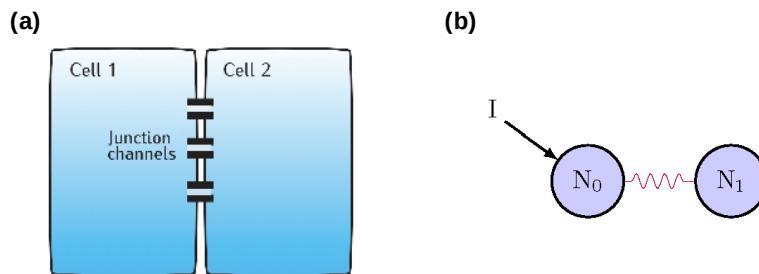


图2-3 (a) 神经元间的缝隙连接. (b) 等效模型.
(引自 Sterratt et al., 2011²)

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。

```
6  class Gap_junction(bp.TwoEndConn):
7      target_backend = ['numpy', 'numba']
8
9      def __init__(self, pre, post, conn, delay=0., k_spikelet=0.1,
10                  post_refractory=False, **kwargs):
11          self.delay = delay
12          self.k_spikelet = k_spikelet
13          self.post_has_refractory = post_refractory
14
15          # connections
16          self.conn = conn(pre.size, post.size)
17          self.pre_ids, self.post_ids = conn.requires('pre_ids', 'post_ids')
18          self.size = len(self.pre_ids)
19
20          # variables
21          self.w = bp.ops.ones(self.size)
22          self.spikelet = self.register_constant_delay('spikelet', size=self.size,
23                                                       delay_time=self.delay)
23
24
25          super(Gap_junction, self).__init__(pre=pre, post=post, **kwargs)
26
27      def update(self, _t):
28          for i in range(self.size):
29              pre_id = self.pre_ids[i]
30              post_id = self.post_ids[i]
31
32              self.post.input[post_id] += self.w[i] * (self.pre.V[pre_id] -
33                                              self.post.V[post_id]) ] Ipost = w (Vpre - Vpost)
34
35              self.spikelet.push(i, self.w[i] * self.k_spikelet *
36                                  self.pre.spikes[pre_id])
37
38              out = self.spikelet.pull(i)
39              if self.post_has_refractory:
40                  self.post.V[post_id] += out * (1. -
41                                              self.post.refractory[post_id])
42              else:
43                  self.post.V[post_id] += out
44
```

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
import numpy as np

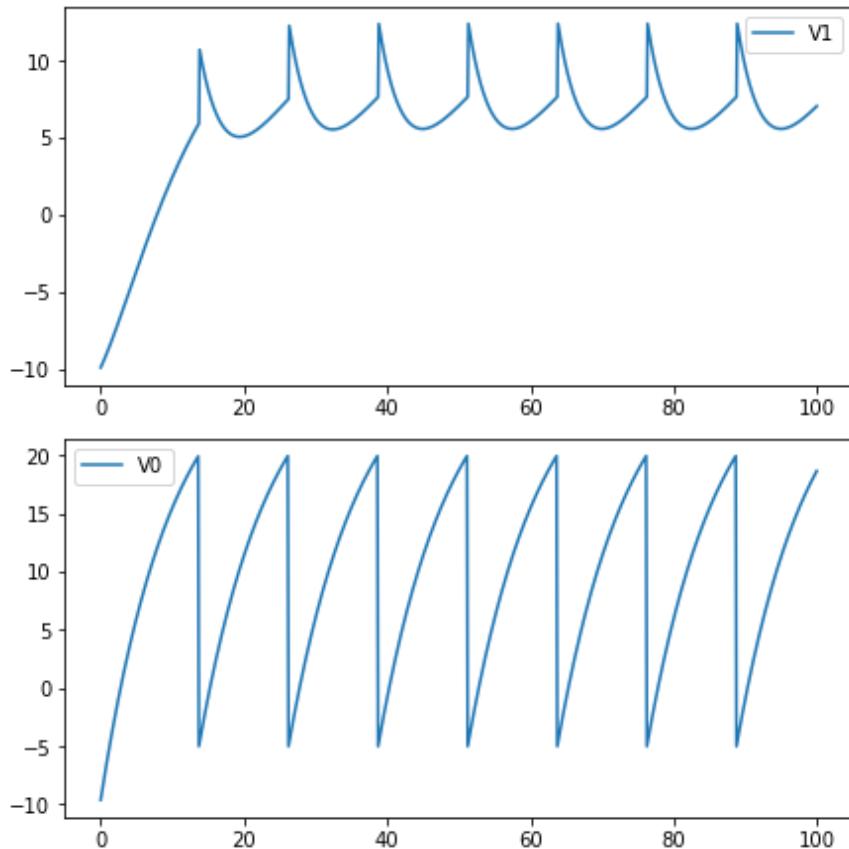
neu0 = bm.neurons.LIF(2, monitors=['V'], t_refractory=0)
neu0.V = np.ones(neu0.V.shape) * -10.
neu1 = bm.neurons.LIF(3, monitors=['V'], t_refractory=0)
neu1.V = np.ones(neu1.V.shape) * -10.
syn = Gap_junction(pre=neu0, post=neu1, conn=bp.connect.All2All(),
                     k_spikelet=5.)
syn.w =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

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

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

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

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



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。

可以看到，当 V_0 在阈值以下上升时， V_1 也跟着上升；而当 V_0 达到阈值产生动作电位时， V_1 有一个快速的上升（spikelet）以后马上降到一个更低的值。

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 modeling 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。

可塑性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。

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2-1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）。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。

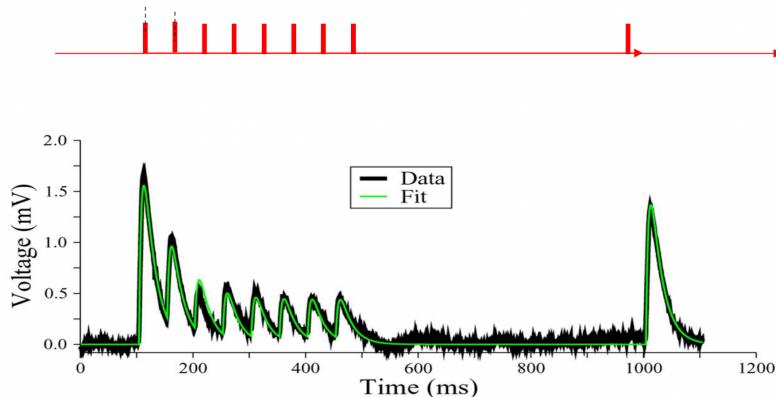


图2-1 突触短时程可塑性 (改编自 Gerstner et al., 2014¹)

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。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：

$$\begin{aligned} \frac{dI}{dt} &= -\frac{I}{\tau} \\ \frac{du}{dt} &= -\frac{u}{\tau_f} \\ \frac{dx}{dt} &= \frac{1-x}{\tau_d} \\ \text{if (pre fire), then } &\begin{cases} u^+ = u^- + U(1-u^-) \\ I^+ = I^- + Au^+x^- \\ x^+ = x^- - u^+x^- \end{cases} \end{aligned}$$

1.1 生物背景

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），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），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起主导作用。

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`。

```

1  class STP(bp.TwoEndConn):
2      target_backend = ['numpy', 'numba']
3
4      @staticmethod
5      def derivative(s, u, x, t, tau, tau_d, tau_f):
6          dsdt = -s / tau
7          dudt = -u / tau_f
8          dxdt = (1 - x) / tau_d
9          return dsdt, dudt, dxdt
10
11     def __init__(self, pre, post, conn, delay=0., U=0.15, tau_f=1500.,
12                  tau_d=200., tau=8., **kwargs):
13         # parameters
14         self.tau_d = tau_d
15         self.tau_f = tau_f
16         self.tau = tau
17         self.U = U
18         self.delay = delay
19
20         # connections
21         self.conn = conn(pre.size, post.size)
22         self.pre_ids, self.post_ids = conn.requires('pre_ids', 'post_ids')
23         self.size = len(self.pre_ids)
24
25         # variables
26         self.s = bp.ops.zeros(self.size)
27         self.x = bp.ops.ones(self.size)
28         self.u = bp.ops.zeros(self.size)
29         self.w = bp.ops.ones(self.size)
30         self.I_syn = self.register_constant_delay('I_syn', size=self.size,
31                                                 delay_time=delay)
31
32         self.integral = bp.odeint(f=self.derivative, method='exponential_euler')
33
34     super(STP, self).__init__(pre=pre, post=post, **kwargs)
35
36
37     def update(self, _t):
38         for i in range(self.size):
39             pre_id = self.pre_ids[i]
40
41             self.s[i], u, x = self.integral(self.s[i], self.u[i], self.x[i], _t,
42                                              self.tau, self.tau_d, self.tau_f)
43
44             if self.pre.spike[pre_id] > 0:
45                 u += self.U * (1 - self.u[i])
46                 self.s[i] += self.w[i] * u * self.x[i]
47                 x -= u * self.x[i]
48             self.u[i] = u
49             self.x[i] = x
50
51             # output
52             post_id = self.post_ids[i]
53             self.I_syn.push(i, self.s[i])
54             self.post.input[post_id] += self.I_syn.pull(i)
55

```

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']`。

1.1 生物背景

```
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)

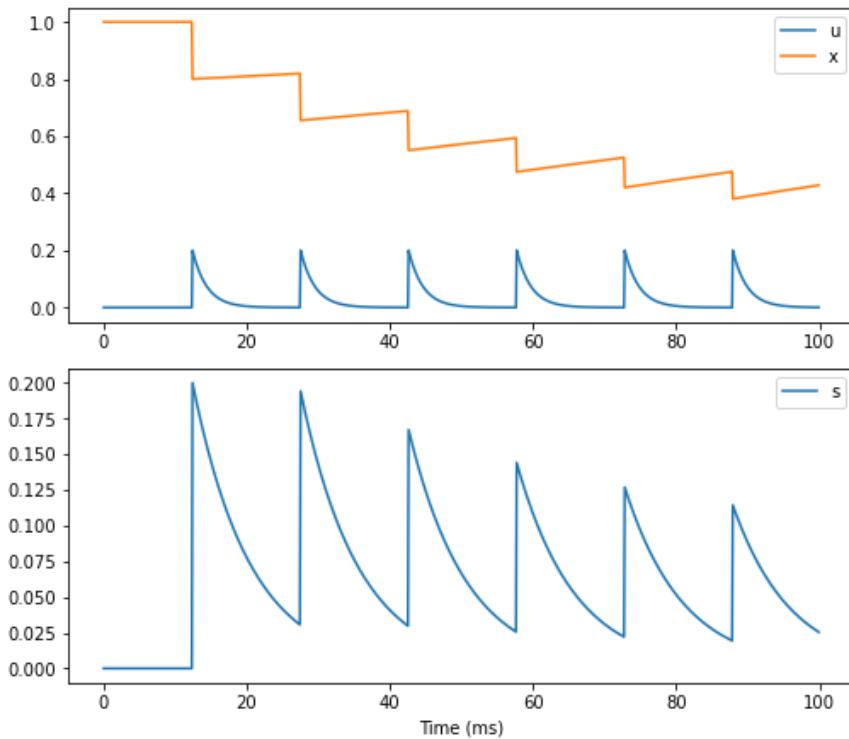
    fig.add_subplot(gs[0, 0])
    plt.plot(net.ts, syn.mon.u[:, 0], label='u')
    plt.plot(net.ts, syn.mon.x[:, 0], label='x')
    plt.legend()

    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。

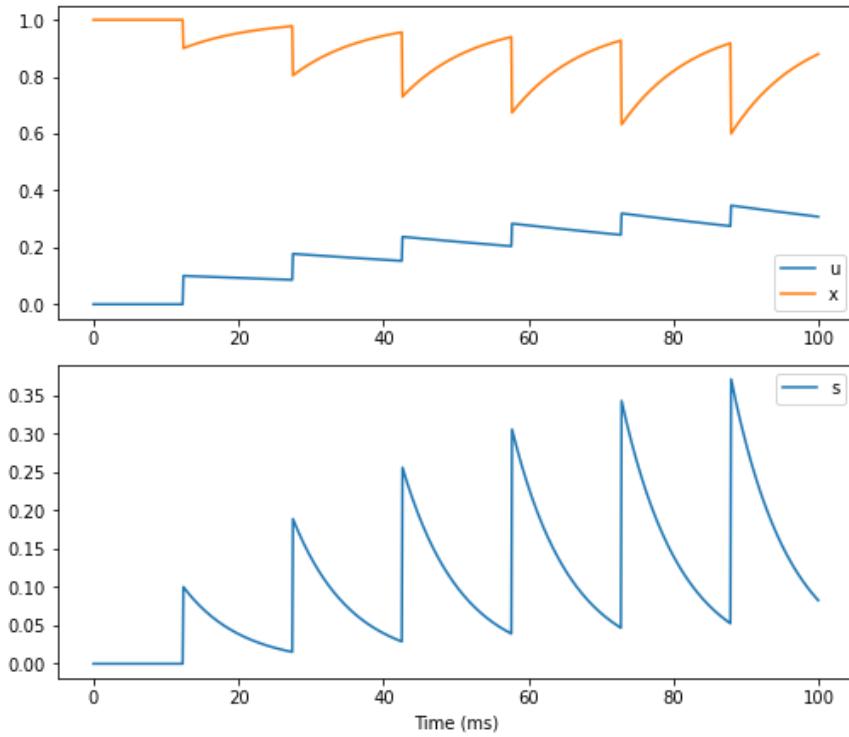
```
run_stp(U=0.2, tau_d=150., tau_f=2.)
```



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 占主导。

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.)
```

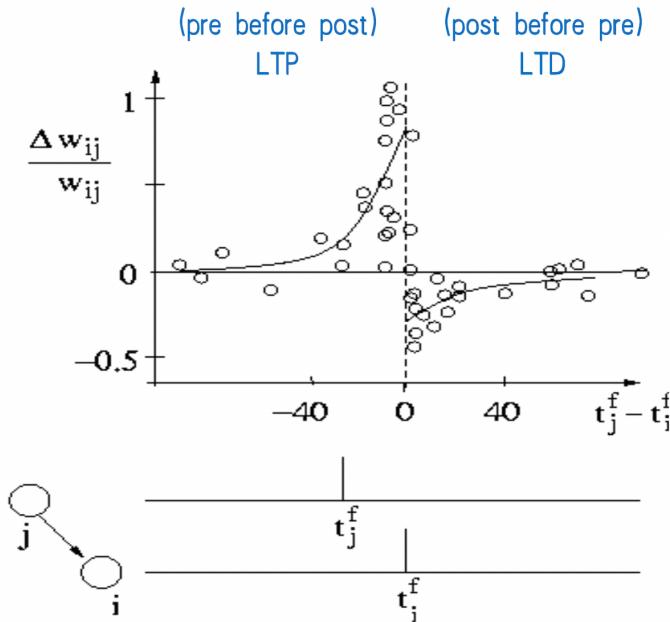


结果图显示，当 $\tau_f > \tau_d$ 时， x 每次用掉后很快又补充回去了，这表示突触前神经元总是有足够的神经递质可用。同时， u 的衰减非常缓慢，即释放神经递质的概率越来越高，从 s 的动力学可以看出STF占主导地位。

2.2.2 突触长时程可塑性

脉冲时间依赖可塑性 (STDP)

图2-2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2-2 脉冲时间依赖可塑性 (改编自 Bi & Poo, 2001²)

STDP的计算模型如下：

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

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

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

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

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

```

1  class STDP(bp.TwoEndConn):
2      target_backend = ['numpy', 'numba']
3
4      @staticmethod
5      def derivative(s, A_s, A_t, t, tau, tau_s, tau_t):
6          dsdt = -s / tau
7          dAsdt = -A_s / tau_s
8          dAtdt = -A_t / tau_t
9          return dsdt, dAsdt, dAtdt
10
11     def __init__(self, pre, post, conn, delay=0., delta_A_s=0.5,
12                  delta_A_t=0.5, w_min=0., w_max=20., tau_s=10., tau_t=10.,
13                  tau=10., **kwargs):
14         # parameters
15         self.tau_s = tau_s
16         self.tau_t = tau_t
17         self.tau = tau
18         self.delta_A_s = delta_A_s
19         self.delta_A_t = delta_A_t
20         self.w_min = w_min
21         self.w_max = w_max
22         self.delay = delay
23
24         # connections
25         self.conn = conn(pre.size, post.size)
26         self.pre_ids, self.post_ids = self.conn.requires('pre_ids', 'post_ids')
27         self.size = len(self.pre_ids)
28
29         # variables
30         self.s = bp.ops.zeros(self.size)
31         self.A_s = bp.ops.zeros(self.size)
32         self.A_t = bp.ops.zeros(self.size)
33         self.w = bp.ops.ones(self.size) * 1.
34         self.I_syn = self.register_constant_delay('I_syn', size=self.size,
35                                                 delay_time=delay)
36         self.integral = bp.odeint(f=self.derivative, method='exponential_euler')
37
38     super(STDP, self).__init__(pre=pre, post=post, **kwargs)
39
40     def update(self, _t):
41         for i in range(self.size):
42             pre_id = self.pre_ids[i]
43             post_id = self.post_ids[i]
44
45             self.s[i], A_s, A_t = self.integral(self.s[i], self.A_s[i],
46                                               self.A_t[i], _t, self.tau,
47                                               self.tau_s, self.tau_t)
48
49             w = self.w[i]
50             if self.pre.spike[pre_id] > 0:
51                 self.s[i] += w
52                 A_s += self.delta_A_s
53                 w -= A_t
54
55             if self.post.spike[post_id] > 0:
56                 A_t += self.delta_A_t
57                 w += A_s
58
59             self.A_s[i] = A_s
60             self.A_t[i] = A_t
61
62             self.w[i] = bp.ops.clip(w, self.w_min, self.w_max) ————— limit w
63
64             # output
65             self.I_syn.push(i, self.s[i])
66             self.post.input[post_id] += self.I_syn.pull(i)
67

```

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 $t = 5\text{ms}$ 时刻给突触前神经元第一段电流（每一段强度为 $30 \mu\text{A}$ ，并持续 15ms ，保证LIF模型会产生一个脉冲），然后在 $t = 10\text{ms}$ 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 $t_{post} = t_{pre} + 5$ 。接下来我们设置一个较长的间隔，然后把刺激顺序调整为 $t_{post} = t_{pre} - 3$ 。

1.1 生物背景

```
duration = 300.
(I_pre, _) = bp.inputs.constant_current([(0, 5), (30, 15),    # pre at 5ms
                                         (0, 15), (30, 15),
                                         (0, 15), (30, 15),
                                         (0, 98), (30, 15),  # switch order: t_interval=98ms
                                         (0, 15), (30, 15),
                                         (0, 15), (30, 15),
                                         (0, duration-155-98)])
(I_post, _) = bp.inputs.constant_current([(0, 10), (30, 15), # post at 10
                                         (0, 15), (30, 15),
                                         (0, 15), (30, 15),
                                         (0, 90), (30, 15), # switch order: t_interval=98-8=90(ms)
                                         (0, 15), (30, 15),
                                         (0, 15), (30, 15),
                                         (0, duration-160-90)])
```

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。

1.1 生物背景

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

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

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

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

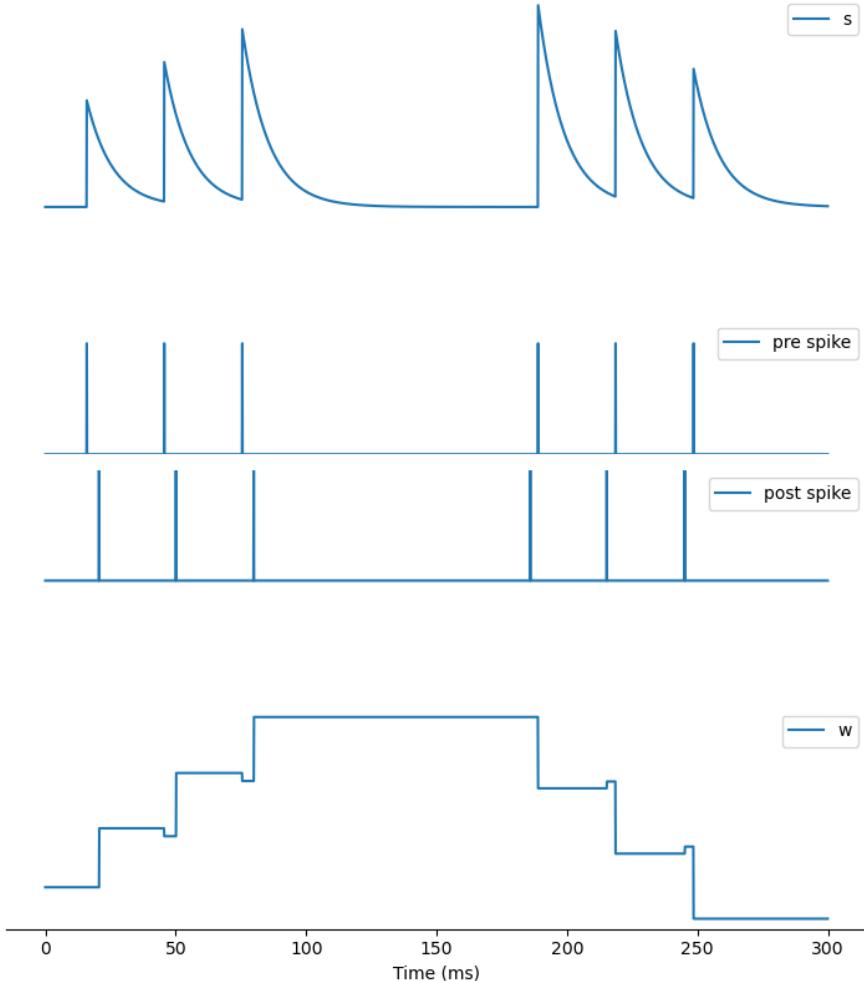
ax=fig.add_subplot(gs[0, 0])
plt.plot(net.ts, syn.mon.s[:, 0], label="s")
hide_spines(ax)

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

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

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

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



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。而这个同步不需要在意两个神经元前后发放的次序，因此可以忽略具体的发放时间，简化为发放率模型。我们首先看赫布学习律的一般形式，对于 j 到 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：

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

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。上式给出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):           bp.TwoEndConn class:
2      target_backend = ['numpy', 'numba']   Learning rule occur between two neuron
3
4      @staticmethod
5      def derivative(w, t, gamma, r_pre, r_post):
6          dwdt = gamma * (r_post * r_pre - r_post * r_post * w)   }   dw/dt = γ(r_i r_j - w r_i^2)
7          return dwdt
8
9      def __init__(self, pre, post, conn, delay=0,
10                  gamma=0.005, w_max=1., w_min=0.,
11                  **kwargs):
12          # params
13          self.gamma = gamma
14          self.w_max = w_max
15          self.w_min = w_min
16          # no delay in firing rate models
17
18          # connns
19          self.conn = conn(pre.size, post.size)
20          self.pre_ids, self.post_ids = self.conn.requires('pre_ids', 'post_ids')
21          self.size = len(self.pre_ids)
22
23          # data
24          self.w = bp.ops.ones(self.size) * 0.05
25
26          self.integral = bp.odeint(f=self.derivative)
27          super(Oja, self).__init__(pre=pre, post=post, **kwargs)
28
29
30      def update(self, _t):
31          post_r = bp.ops.zeros(self.post.size[0])
32          for i in range(self.size):
33              pre_id = self.pre_ids[i]
34              post_id = self.post_ids[i]                                rate model: r_i = Σ_j w_ij r_j
35              add = self.w[i] * self.pre.r[pre_id]    → let add_j = w_ij r_j
36              post_r[post_id] += add                → then r_i = Σ_j add_j
37              self.w[i] = self.integral(
38                  self.w[i], _t, self.gamma,
39                  self.pre.r[pre_id], self.post.r[post_id]) → use self.post.r, not post_r, see next line
40              self.post.r = post_r → r_i = Σ_j add_j
41
42          Note that the effects of pre groups
43          are simultaneously, so we update
44          post_r after the for loop

```

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。

```

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

```

我们打算实现如图2-3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。

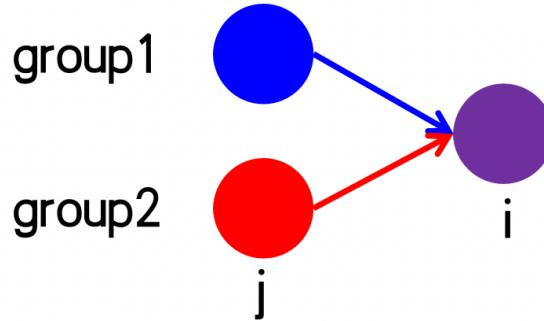


图2-3 神经元的连接

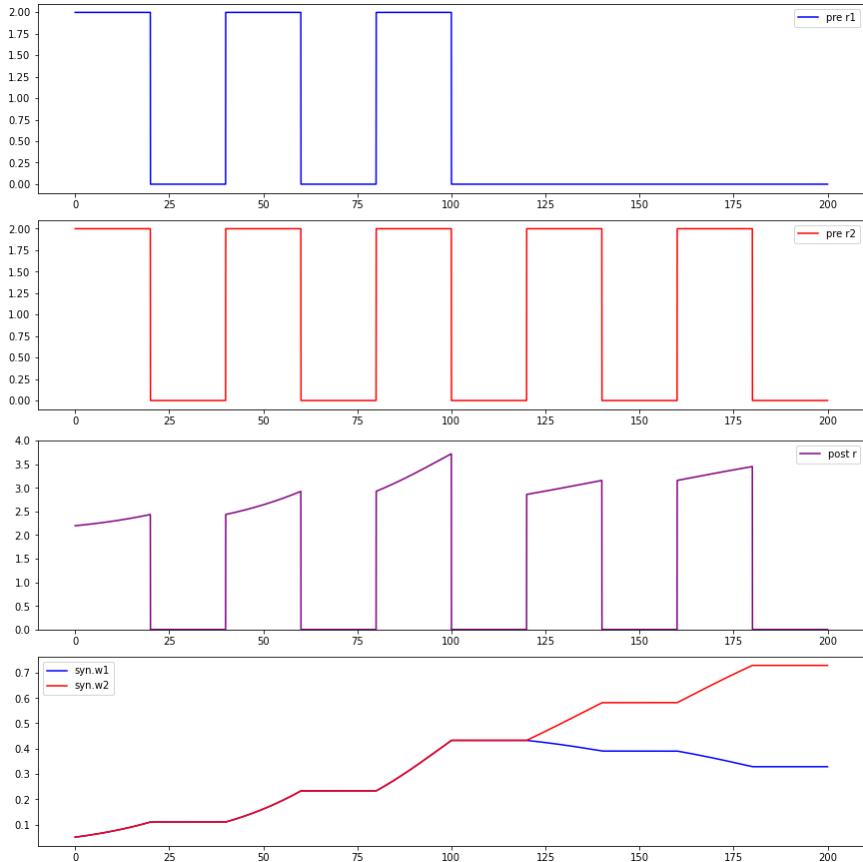
```

51 # create input
52 current1, _ = bp.inputs.constant_current(
53     [(2., 20.), (0., 20.)] * 3 + [(0., 20.), (0., 20.)] * 2)
54 current2, _ = bp.inputs.constant_current([(2., 20.), (0., 20.)] * 2)
55 current3, _ = bp.inputs.constant_current([(2., 20.), (0., 20.)] * 5)
56 current_pre = np.vstack((current1, current2))
57 current_post = np.vstack((current3, current3))
58
59 # simulate
60 neu_pre = neu(2, monitors=['r'])
61 neu_post = neu(2, monitors=['r'])
62 syn = Oja(pre=neu_pre, post=neu_post, conn=bp.connect.All2All(), monitors=['w'])
63 net = bp.Network(neu_pre, syn, neu_post)
64 net.run(duration=200., inputs=[(neu_pre, 'r', current_pre.T, '='),
65                               (neu_post, 'r', current_post.T)])
66
67 # plot
68 fig, gs = bp.visualize.get_figure(4, 1)
69
70 fig.add_subplot(gs[0, 0])
71 plt.plot(net.ts, neu_pre.mon.r[:, 0], 'b', label='pre r1') —————> index[:, 0], group 1
72 plt.legend()
73
74 fig.add_subplot(gs[1, 0])
75 plt.plot(net.ts, neu_pre.mon.r[:, 1], 'r', label='pre r2') —————> index[:, 0], group 2
76 plt.legend()
77
78 fig.add_subplot(gs[2, 0])
79 plt.plot(net.ts, neu_post.mon.r[:, 0], color='purple', label='post r')
80 plt.ylim([0, 4])
81 plt.legend()
82
83 fig.add_subplot(gs[3, 0])
84 plt.plot(net.ts, syn.mon.w[:, 0], 'b', label='syn.w1')
85 plt.plot(net.ts, syn.mon.w[:, 1], 'r', label='syn.w2')
86 plt.legend()
87 plt.show()

```

Annotations for the code:

- current1: input to pre-group1
- current2: input to pre-group2
- current3: input to post group, the same as current2.(fire together)
- simulation is like synapse model. Here we give inputs to both pre-group and post-group.



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2-4）。图2-4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。

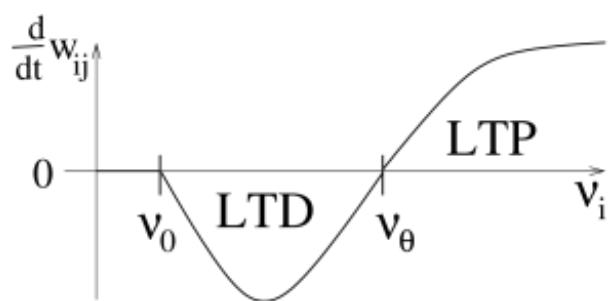


图2-4 BCM法则 (引自 Gerstner et al., 2014¹)

我们将实现和Oja法则相同的连接方式（图2-3），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      }    $\frac{dw}{dt} = \eta r_i(r_i - r_\theta)r_j$ 
7          return dwdt
8
9      def __init__(self, pre, post, conn, lr=0.005, w_max=2., w_min=0., **kwargs):
10         # parameters
11         self.lr = lr
12         self.w_max = w_max
13         self.w_min = w_min
14         self.dt = bp.backend._dt
15
16         # connections
17         self.conn = conn(pre.size, post.size)
18         self.pre_ids, self.post_ids = self.conn.requires('pre_ids', 'post_ids')
19         self.size = len(self.pre_ids)
20
21         # variables
22         self.w = bp.ops.ones(self.size)
23         self.sum_post_r = bp.ops.zeros(post.size[0])
24
25         self.int_w = bp.odeint(f=self.derivative, method='rk4')
26
27         super(BCM, self).__init__(pre=pre, post=post, **kwargs)
28
29     def update(self, _t):
30         # update threshold
31         self.sum_post_r += self.post.r      }    $r_\theta = \frac{\sum_t dt r_i}{T}$ 
32         r_th = self.sum_post_r / (_t / self.dt + 1)      }   self.dt+1 here since
33
34         # update w and post_r
35         post_r = bp.ops.zeros(self.post.size[0])
36
37         for i in range(self.size):
38             pre_id = self.pre_ids[i]
39             post_id = self.post_ids[i]
40
41             post_r[post_id] += self.w[i] * self.pre.r[pre_id]      }   rate model:  $r_i = \sum_j w_{ij} r_j$ 
42
43             w = self.int_w(self.w[i], _t, self.lr, self.pre.r[pre_id],
44                            self.post.r[post_id], r_th[post_id])      }   let  $add_j = w_{ij}r_j$ 
45
46             self.w[i] = bp.ops.clip(w, self.w_min, self.w_max)      }   then  $r_i = \sum_j add_j$ 
47
48             self.post.r = post_r      }   limit w
49
50             ...      }    $r_i = \sum_j add_j$ 
```

定义了BCM类以后，我们可以跑模拟了。

1.1 生物背景

```

n_post = 1
n_pre = 20

# group selection
group1, duration = bp.inputs.constant_current(([1.5, 1], [0, 1]) * 20)
group2, duration = bp.inputs.constant_current(([0, 1], [1, 1]) * 20)
group1 = bp.ops.vstack(((group1,) * 10))
group2 = bp.ops.vstack(((group2,) * 10))
input_r = bp.ops.vstack((group1, group2))

pre = neu(n_pre, monitors=['r'])
post = neu(n_post, monitors=['r'])
bcm = BCM(pre=pre, post=post, conn=bp.connect.All2All(),
           monitors=['w'])

net = bp.Network(pre, bcm, post)
net.run(duration, inputs=(pre, 'r', input_r.T, "="))

w1 = bp.ops.mean(bcm.mon.w[:, :10, 0], 1)
w2 = bp.ops.mean(bcm.mon.w[:, 10:, 0], 1)

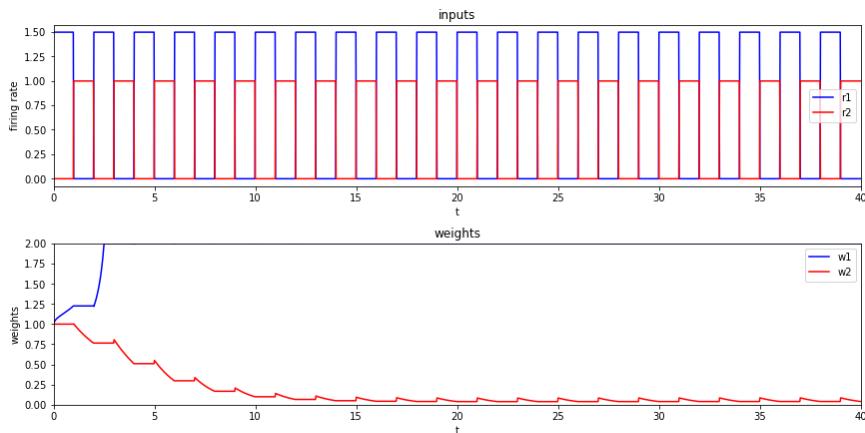
r1 = bp.ops.mean(pre.mon.r[:, :10], 1)
r2 = bp.ops.mean(pre.mon.r[:, 10:], 1)

fig, gs = bp.visualize.get_figure(2, 1, 3, 12)
fig.add_subplot(gs[1, 0], xlim=(0, duration), ylim=(0, w_max))
plt.plot(net.ts, w1, 'b', label='w1')
plt.plot(net.ts, w2, 'r', label='w2')
plt.title("weights")
plt.ylabel("weights")
plt.xlabel("t")
plt.legend()

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

plt.show()

```



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. 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种重要的网络模型：脉冲神经网络和发放率神经网络。

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；而发放率网络则将一群神经元简化为单个的发放率单元，并以一个发放率单元中的神经元群作为计算的最小单位。

3.1 脉冲神经网络

3.2 发放率神经网络

3.1 脉冲神经网络

3.1.1 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 and 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。

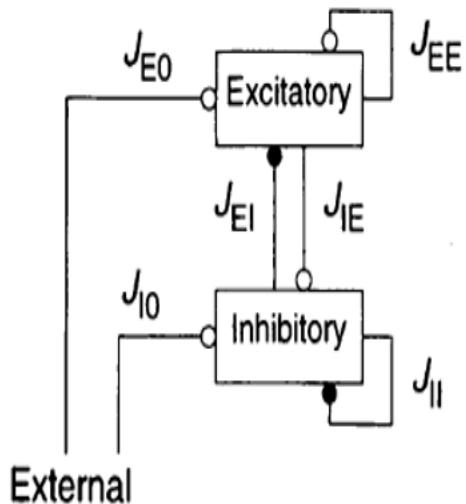


Fig.3-1 Structure of E/I balanced network | Vreeswijk and Sompolinsky, 1996

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
12
13
14  neu_E = brainmodels.neurons.LIF(N_E, monitors=['spike'])
15  neu_I = brainmodels.neurons.LIF(N_I, monitors=['spike'])
16  neu_E.V = V_rest + np.random.random(N_E) * (V_th - V_rest)
17  neu_I.V = V_rest + np.random.random(N_I) * (V_th - V_rest)
18
19
20  syn_E2E = brainmodels.synapses.Exponential(
21      pre=neu_E, post=neu_E,
22      conn=bp.connect.FixedProb(prob=prob))
23  syn_E2I = brainmodels.synapses.Exponential(
24      pre=neu_E, post=neu_I,
25      conn=bp.connect.FixedProb(prob=prob))
26  syn_I2E = brainmodels.synapses.Exponential(
27      pre=neu_I, post=neu_E,
28      conn=bp.connect.FixedProb(prob=prob))
29  syn_I2I = brainmodels.synapses.Exponential(
30      pre=neu_I, post=neu_I,
31      conn=bp.connect.FixedProb(prob=prob))
32
33
34  JE = 1 / np.sqrt(prob * N_E)
35  JI = 1 / np.sqrt(prob * N_I)
36  syn_E2E.w = JE
37  syn_E2I.w = JE
38  syn_I2E.w = -JI
39  syn_I2I.w = -JI

```

Import LIF neuron model from brainmodels.
Build excitatory neuron group and inhibitory neuron group.

Import exponential synapse model from brainmodels.
Build E2E, E2I, I2E, I2I synapse connections between neuron groups.

$$weight\ of\ connection = J_E = J_I = \frac{1}{\sqrt{p N_E}} = \frac{1}{\sqrt{p N_I}}$$

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。

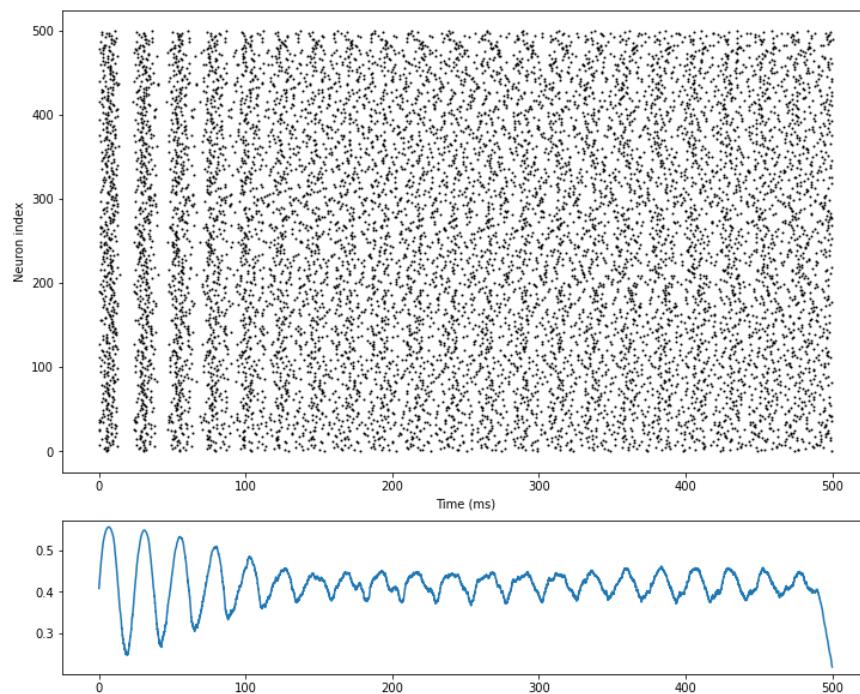
```

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.visualization.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

```

Simulate the network for 500ms,
give all neurons constant external
inputs with amplitude 3.

Paint the raster plot and firing_rate-t plot of
excitatory neurons in E/I balanced network.



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。

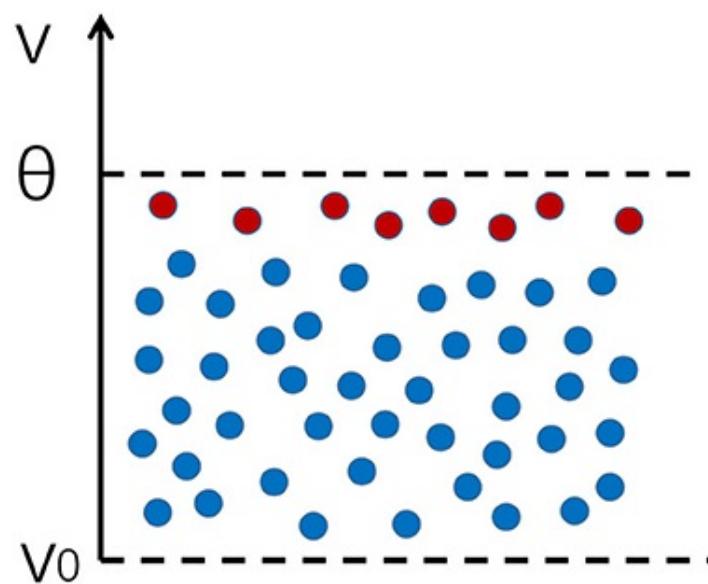


Fig.3-2 Distribution of neuron membrane potentials in E/I balanced network | Tian et al., 2020

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。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。

3.1.2 决策网络

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。

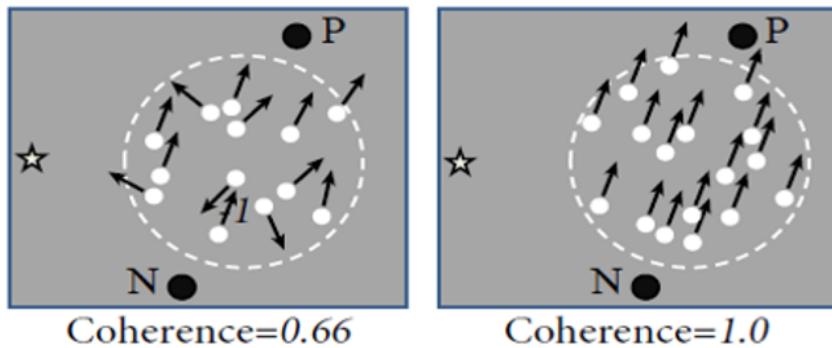


Fig.3-4 Experimental Diagram

Wang (2002) 提出了本节所述的决策网络，希望建模在视觉运动区分实验中猕猴大脑新皮层的决策回路的活动。

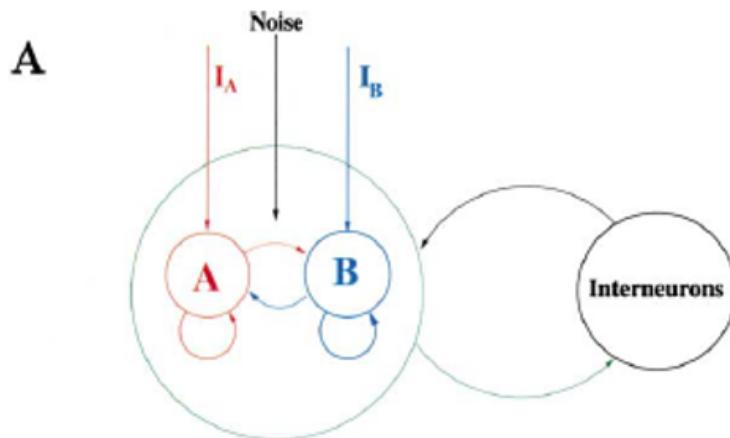


Fig.3-5 structure of decision making network

如图3-5所示，网络同样基于兴奋-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。

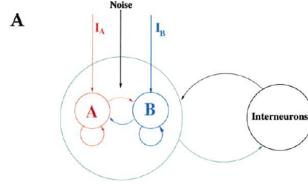
为了简化模型，实验被设定为一个二选一的任务：在兴奋性神经元群中，特别地标出两个选择性子神经元群A和B，其他的兴奋性神经元称为非选择性神经元，用下标_{non}表示。A群和B群的数目均为兴奋性神经元的0.15倍（ $N_A = N_B = 0.15 N_E$ ），它们分别代表着两个相反的运动方向，可以视作随机点要么向左，要么向右，没有第三个方向，网络的决策结果也必须在这两个子神经元群中产生。非选择性神经元的数目为 $N_{non} = (1 - 2 * 0.15) N_E$ 。

1.1 生物背景

```

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

```



决策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群之间作出决策，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决策。

Sheet 3-1 Weight of synapse connections between E-neurons

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

1.1 生物背景

```

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, 0:N_A] = w_pos
262     weight[i, 0:N_A] = w_neg
263 for i in range(N_A+N_B, N_E):
264     weight[i, 0:N_A+N_B] = w_neg
265 print(f"Check constraints: Weight sum {weight.sum(axis=0)[0]} \
266 should be equal to N_E = {N_E}")

```

Define structured weights for E2E connns.
Strong connection inside selective group,
weak connection for other synapses.

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

决策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）。

```

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) \ } Generate spikes with a given Poisson frequency.
502             < self.freqs * self.dt / 1000.
503
504 # poisson_freq = 2400Hz
505 neu_poisson_A = PoissonInput(N_A, freqs=poisson_freq, dt=dt)
506 neu_poisson_B = PoissonInput(N_B, freqs=poisson_freq, dt=dt)
507 neu_poisson_non = PoissonInput(N_non, freqs=poisson_freq, dt=dt) } Define neurons that generate Poisson
508 neu_poisson_I = PoissonInput(N_I, freqs=poisson_freq, dt=dt) background inputs for every neuron in
509
510 syn_back2A_AMPA = AMPA(pre=neu_poisson_A, post=neu_A,
511                         conn=bp.connect.One2One())
512 syn_back2B_AMPA = AMPA(pre=neu_poisson_B, post=neu_B,
513                         conn=bp.connect.One2One())
514 syn_back2non_AMPA = AMPA(pre=neu_poisson_non, post=neu_non,
515                         conn=bp.connect.One2One())
516
517 syn_back2I_AMPA = AMPA(pre=neu_poisson_I, post=neu_I,
518                         conn=bp.connect.One2One())

```

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

- 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决策。

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

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

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

1.1 生物背景

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。

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

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

```

529 ## def stimulus input
530 # Note: inputs only given to A and B group
531 mu_0 = 40.
532 coherence = 25.6
533 rou_A = mu_0 / 100.
534 rou_B = mu_0 / 100.
535 mu_A = mu_0 + rou_A * coherence
536 mu_B = mu_0 - rou_B * coherence
537 print(f"coherence = {coherence}, mu_A = {mu_A}, mu_B = {mu_B}")
538
539 class PoissonStim(bp.NeuGroup):
540 """
541 from time <t_start> to <t_end> during the simulation, the neuron
542 generates a poisson spike with frequency <self.freq>. however,
543 the value of <self.freq> changes every <t_interval> ms and obey
544 a Gaussian distribution defined by <mean_freq> and <var_freq>.
545 """
546 target_backend = 'general'
547
548 def __init__(self, size, dt=0., t_start=0., t_end=0., t_interval=0.,
549             mean_freq=0., var_freq=20., **kwargs):
550     self.dt = dt
551     self.stim_start_t = t_start
552     self.stim_end_t = t_end
553     self.stim_change_freq_interval = t_interval
554     self.mean_freq = mean_freq
555     self.var_freq = var_freq
556
557     self.freq = 0.
558     self.t_last_change_freq = -1e7
559     self.spike = bp.ops.zeros(size, dtype=bool)
560
561     super(PoissonStim, self).__init__(size=size, **kwargs)
562
563 def update(self, _t):
564     if self.stim_start_t < _t < self.stim_end_t:
565         if self.stim_change_freq_interval <= _t - self.t_last_change_freq:
566             self.freq = np.random.normal(self.mean_freq, self.var_freq)
567             self.freq = max(self.freq, 0)
568             self.t_last_change_freq = _t
569             self.spike = np.random.random(self.size) < (self.freq * self.dt / 1000)
570     else:
571         self.freq = 0.
572         self.spike[:] = False
573
574
575 neu_input2A = PoissonStim(N_A, dt=dt, t_start=pre_period,
576                           t_end=pre_period + stim_period,
577                           t_interval=50., mean_freq=mu_A, var_freq=10.,
578                           monitors=['freq'])
579 neu_input2B = PoissonStim(N_B, dt=dt, t_start=pre_period,
580                           t_end=pre_period + stim_period,
581                           t_interval=50., mean_freq=mu_B, var_freq=10.,
582                           monitors=['freq'])
583
584 syn_input2A_AMPA = AMPA(pre=neu_input2A, post=neu_A,
585                           conn=bp.connect.OneToOne())
586
587 syn_input2B_AMPA = AMPA(pre=neu_input2B, post=neu_B,
588                           conn=bp.connect.OneToOne())
589

```

Compute basic Poisson frequencies
for stimuli given to neuron group A
and B.
 $\mu_A = \mu_0 + \rho_A * c$
 $\mu_B = \mu_0 - \rho_B * c$

Save model parameters and variables.

Stimulus are only generated in simulation
between time period [stim_start_t, stim_end_t].

Generate new Poisson frequency
obeys Gaussian distribution for
this neuron every 50ms.
 $f_A \sim N(\mu_A, \delta^2)$
 $f_B \sim N(\mu_B, \delta^2)$

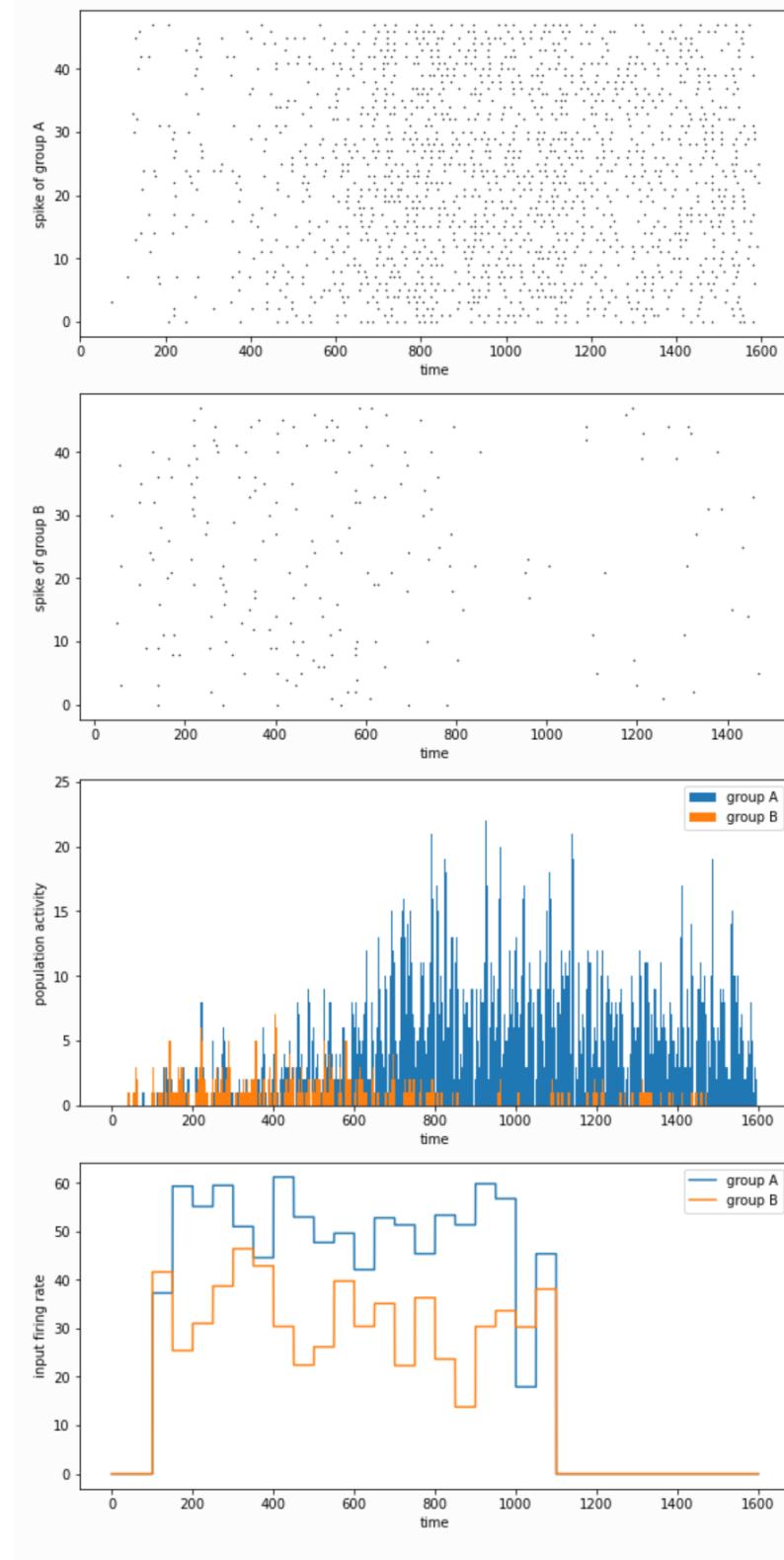
Generate Poisson inputs with the
Poisson frequency of this neuron.

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

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

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。

1.1 生物背景



3.2 放率神经网络

3.2.1 决择模型

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放率 (firing rate) 的简化模型 (Wong & Wang, 2006¹)。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，使用一群神经元的放率来表示整个神经元群的状态，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。

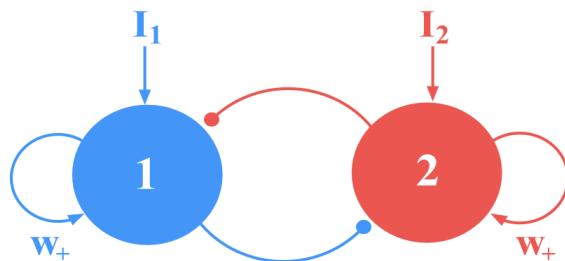


图3-1 简化的抉择模型 (引自 Wong & Wang, 2006¹)

基于放率的抉择模型如图3-1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3-1的模型结构给出：

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

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

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。

```

1  from collections import OrderedDict
2  import brainpy as bp
3  bp.backend.set(backend='numba', dt=0.1)
4
5  class Decision(bp.NeuGroup):
6      target_backend = ['numpy', 'numba']
7
8      @staticmethod
9      def derivative(s1, s2, t, I, coh,
10                     JAext, J_rec, J_inh, I_0,
11                     a, b, d, tau_s, gamma):
12          I1 = JAext * I * (1. + coh)   ] I1 = J_Aext * I * (1. + coh)
13          I2 = JAext * I * (1. - coh)   ] I2 = J_Aext * I * (1. - coh)
14
15          I_syn1 = J_rec * s1 - J_inh * s2 + I_0 + I1
16          r1 = (a * I_syn1 - b) / (1. - bp.ops.exp(-d * (a * I_syn1 - b))) } I_syn1 = J11 S1 - J12 S2 + I0 + I1
17          ds1dt = - s1 / tau_s + (1. - s1) * gamma * r1 } r1 = a I_syn1 - b
18
19          I_syn2 = J_rec * s2 - J_inh * s1 + I_0 + I2
20          r2 = (a * I_syn2 - b) / (1. - bp.ops.exp(-d * (a * I_syn2 - b))) } I_syn2 = J22 S2 - J21 S1 + I0 + I2
21          ds2dt = - s2 / tau_s + (1. - s2) * gamma * r2 } r2 = a I_syn2 - b
22
23          return ds1dt, ds2dt } ds2dt = - S2 / tau_s + (1 - S2) * gamma * r2
24
25      def __init__(self, size, coh, JAext=.00117, J_rec=.3725, J_inh=.1137,
26                  I_0=.3297, a=270., b=108., d=0.154, tau_s=.06, gamma=0.641,
27                  **kwargs):
28          # parameters
29          self.coh = coh
30          self.JAext = JAext
31          self.J_rec = J_rec
32          self.J_inh = J_inh
33          self.I0 = I_0
34          self.a = a
35          self.b = b
36          self.d = d
37          self.tau_s = tau_s
38          self.gamma = gamma
39
40          # variables
41          self.s1 = bp.ops.ones(size) * .06
42          self.s2 = bp.ops.ones(size) * .06
43          self.input = bp.ops.zeros(size)
44
45          self.integral = bp.odeint(f=self.derivative, method='rk4', dt=0.01) }
46
47          super(Decision, self).__init__(size=size, **kwargs)
48
49      def update(self, _t):
50          for i in range(self.size):
51              self.s1[i], self.s2[i] = self.integral(self.s1[i], self.s2[i], _t,
52                                                    self.input[i], self.coh,
53                                                    self.JAext, self.J_rec,
54                                                    self.J_inh, self.I0,
55                                                    self.a, self.b, self.d,
56                                                    self.tau_s, self.gamma)
57
58          self.input[i] = 0.
```

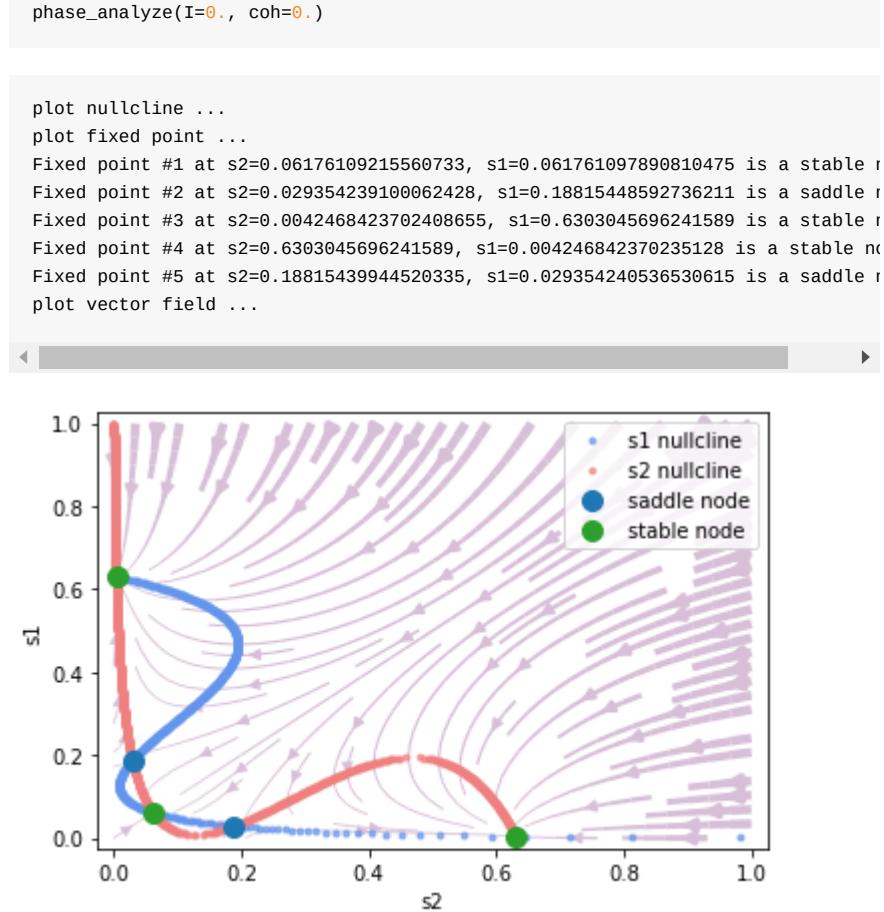
接下来，我们想要看模型在不同输入情况下的动力学，因此，我们先定义一个对抉择模型做相平面分析的方法，可以让我们改变 I （即外界输入强度 μ_0 ）和 coh （即输入的一致性 c' ），而固定了参数的值等。

```

60 def phase_analyze(I, coh):
61     decision = Decision(1, coh=coh)
62
63     phase = bp.analysis.PhasePlane(decision.integral,           Functions defining the differential equations
64                                     target_vars=OrderedDict(s2=[0., 1.],           (from decision model)
65                                     s1=[0., 1.]),                                Variables to be showed
66                                     fixed_vars=None,                            in phase plane.
67                                     pars_update=dict(I=I, coh=coh,
68                                         JAext=.00117, J_rec=.3725,
69                                         J_inh=.1137, I_0=.3297,
70                                         a=270., b=108., d=0.154,
71                                         tau_s=.06, gamma=0.641),
72                                     numerical_resolution=.001,
73                                     options={'escape_sympy_solver': True})      Specify parameters.
74
75     phase.plot_nullcline()
76     phase.plot_fixed_point()
77     phase.plot_vector_field(show=True)            We use numerical solution
                                                 rather than analytic solution
                                                 because the equations are
                                                 so complex. Thus, we don't
                                                 need to use the sympy
                                                 solver.

```

现在让我们来看看当没有外界输入，即 $\mu_0 = 0$ 时的动力学。



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。

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。

1.1 生物背景

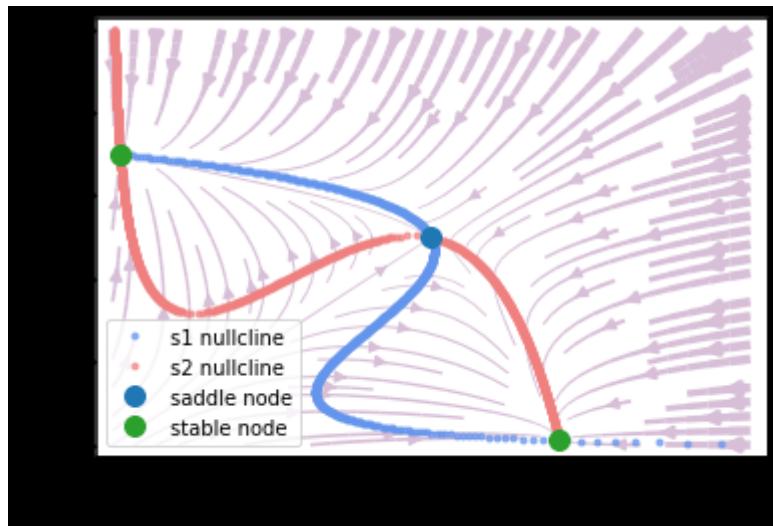
现在让我们看看当我们把外部输入强度固定为30时，在不同一致性（coherence）下的相平面。

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

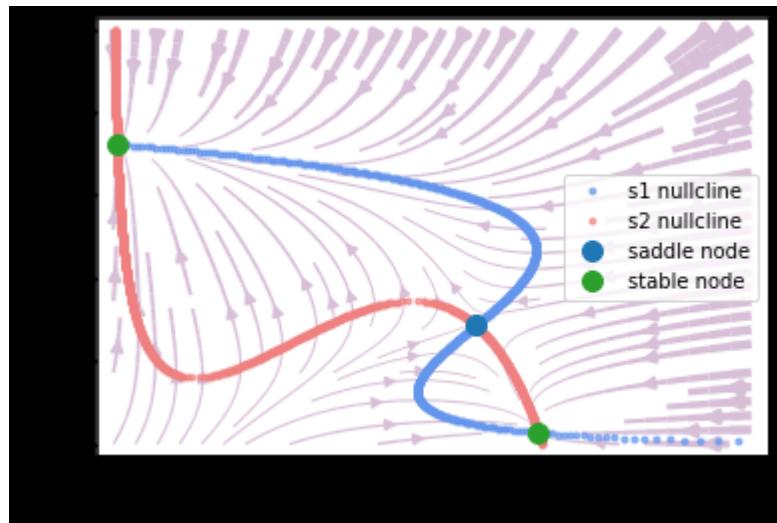
# coherence = 51.2%
print("coherence = 51.2%")
phase_analyze(I=30., coh=0.512)

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

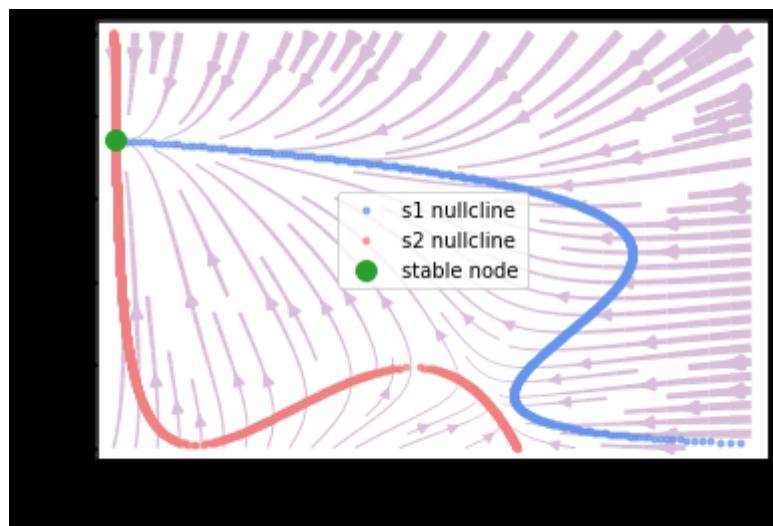
```
coherence = 0%
plot nullcline ...
plot fixed point ...
Fixed point #1 at s2=0.6993504413889349, s1=0.011622049526766405 is a stable node
Fixed point #2 at s2=0.49867489858358865, s1=0.49867489858358865 is a saddle node
Fixed point #3 at s2=0.011622051540013889, s1=0.6993504355529329 is a stable node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



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



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



3.2.2 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)。一维CANN的结构如下：

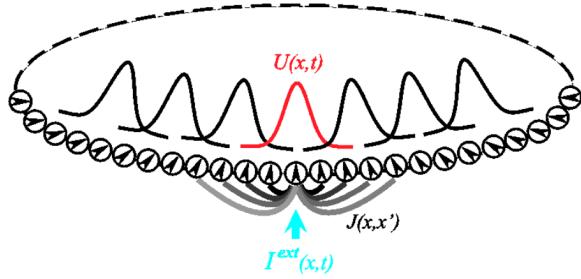


图3-2 连续吸引子神经网络 (引自 Wu et al., 2008²)

神经元群的突触总输入 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 表示神经元群的参数空间位点， $r(x', t)$ 为神经元群(x')的发放率，由以下公式给出：

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

而神经元群(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 } Distances on the ring:
33        self.dx = self.z_range / num # The stimulus density }  $\rho \int dx'$   

34 }  $z_{\text{range}} \text{ denotes the range of } x$ 
35
36 @staticmethod
37 @bp.odeint(method='rk4', dt=0.05)
38 def int_u(u, t, conn, k, tau, Iext):
39     r1 = np.square(u)
40     r2 = 1.0 + k * np.sum(r1)
41     r = r1 / r2
42     Irec = np.dot(conn, r)
43     du = (-u + Irec + Iext) / tau
44     return du
45
46 def dist(self, d):
47     d = np.remainder(d, self.z_range)
48     d = np.where(d > 0.5 * self.z_range, d - self.z_range, d) } Distances on the ring:  

49     return d }  $x$  is the position of the ring, so we have:  

50
51 def make_conn(self, x):
52     assert np.ndim(x) == 1
53     x_left = np.reshape(x, (-1, 1))
54     x_right = np.repeat(x.reshape((1, -1)), len(x), axis=0)
55     d = self.dist(x_left - x_right) }  $d = |x - x'|$   

56     Jxx = self.J0 * np.exp(-0.5 * np.square(d / self.a)) / ( } Get a matrix of  $d$  for  

57                                         np.sqrt(2 * np.pi) * self.a) } all positions
58     return Jxx }  $J(x, x') = \frac{\exp(-0.5(\frac{d}{a})^2)}{\sqrt{2\pi}a}$ 
59
60 def get_stimulus_by_pos(self, pos):
61     return self.A * np.exp(-0.25 * np.square(self.dist(self.x - pos) / self.a)) }  $I_{ek} = A \exp\left(\frac{|x - z(t)|^2}{4a^2}\right)$ 
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.

```

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。

我们用函数 `get_stimulus_by_pos` 来根据参数空间位点 `pos` 处理外界输入，可供用户在使用时调用获取所需的输入电流大小。例如在简单的群编码（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

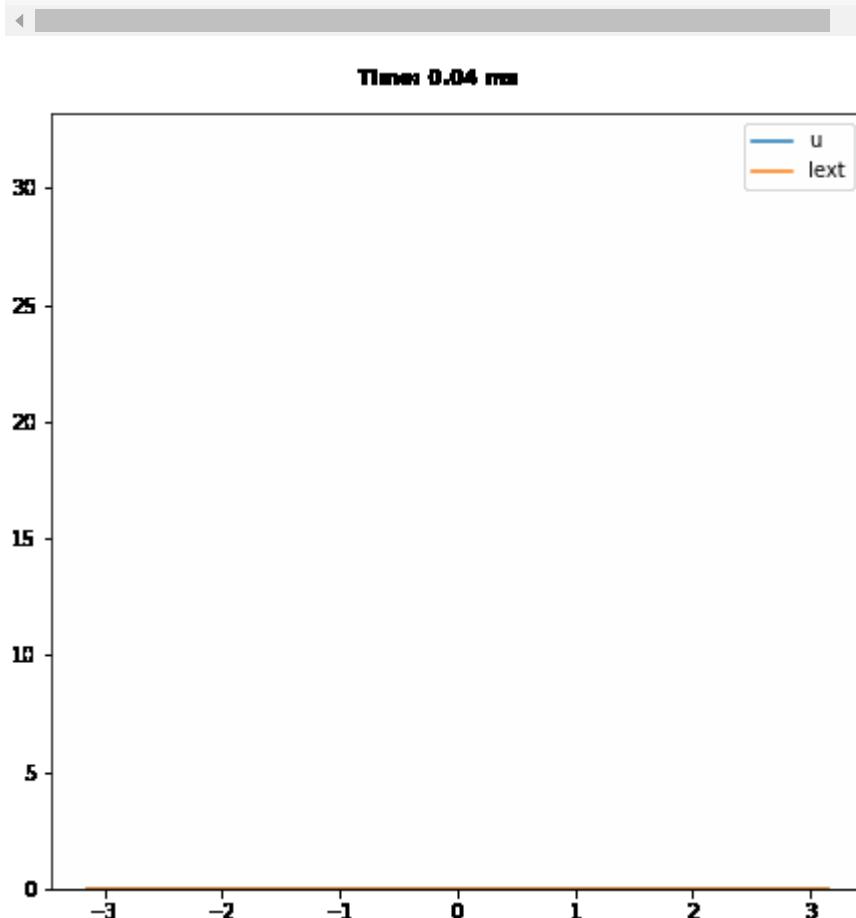
```
cann = CANN1D(num=512, k=0.1, monitors=['u'])

I1 = cann.get_stimulus_by_pos(0.)
Itext, duration = bp.inputs.constant_current([(0., 1.), (I1, 8.), (0., 8.)])
cann.run(duration=duration, inputs=('input', Itext))
```

由于在之后的运行中，画结果图的代码是一样的，我们写一个 `plot_animate` 的函数来调用 `bp.visualize.animate_1D`。

```
# 定义函数
def plot_animate(frame_step=5, frame_delay=50):
    bp.visualize.animate_1D(dynamical_vars=[{'ys': cann.mon.u, 'xs': cann.x,
                                              'legend': 'u'}, {'ys': Itext,
                                              'xs': cann.x, 'legend': 'Itext'}],
                           frame_step=frame_step, frame_delay=frame_delay,
                           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。

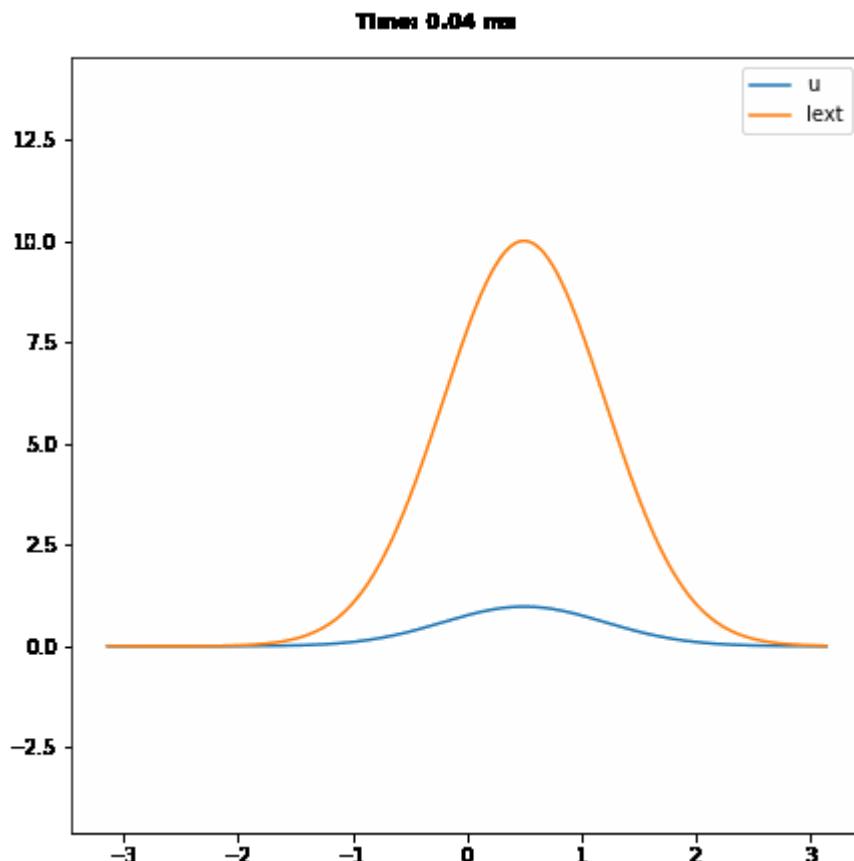
现在我们给外界输入加上随机噪声，看看 u 的形状如何变化。

1.1 生物背景

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

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

plot_animate()
```



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可以进行模版匹配。

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，即平滑追踪。

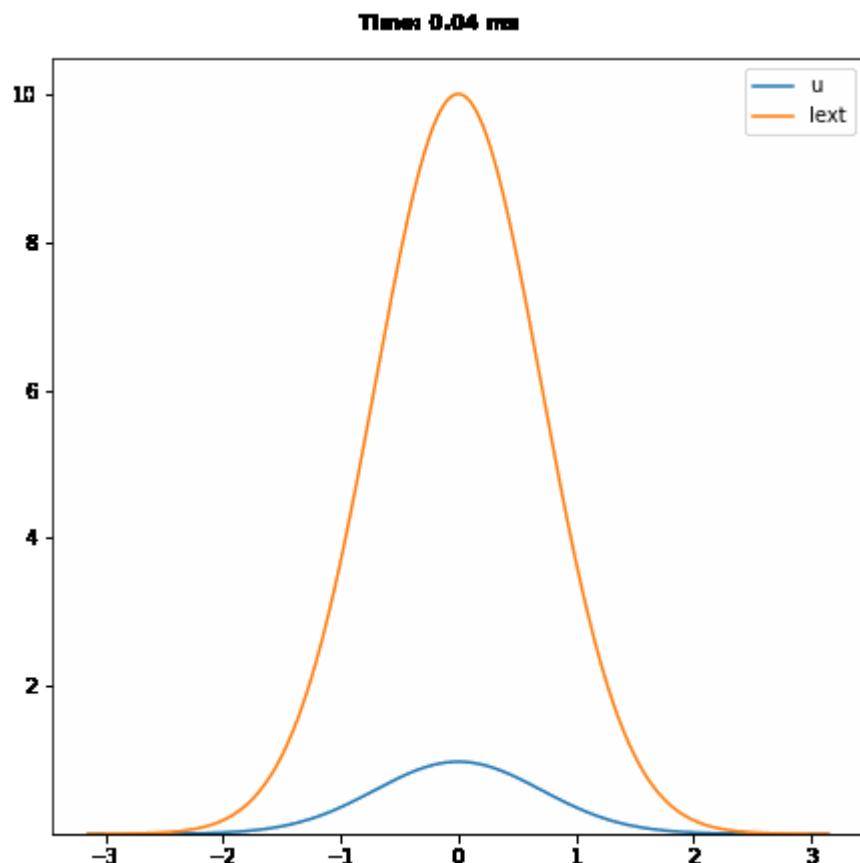
```

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))
Iext = cann.get_stimulus_by_pos(position)
cann.run(duration=dur1 + dur2 + dur3, inputs=('input', Iext))

plot_animate()

```



参考资料

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