Hodgkin-Huxley

brain dynamics programming

张天秋
Peking University

目录 CONTENTS

北京大学 PEKING UNIVERSITY

01

Dynamics Programming Basics

Integrators

$$\frac{dx}{dt} = f_1(x, t, y, p_1)$$
$$\frac{dy}{dt} = g_1(x, t, y, p_2)$$

Variables

Parameters

```
1 def diff(x, y, t, p1, p2):
2     dx = f1(x, t, y, p1)
3     dy = g1(y, t, x, p2)
4     return dx, dy
```

Decorators: brainpy.odeint()

```
1 @bp.odeint
2 def diff(x, y, t, p1, p2):
3     dx = f1(x, t, y, p1)
4     dy = g1(y, t, x, p2)
5     return dx, dy
```

```
1 @bp.odeint(method='euler', dt=0.01)
2 def diff(x, y, t, p1, p2):
3     dx = f1(x, t, y, p1)
4     dy = g1(y, t, x, p2)
5     return dx, dy
```

Examples

FitzHugh-Nagumo equation

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

$$\dot{v} = v - \frac{v^3}{3} - w + I_{\text{ext}}.$$

```
@bp.odeint(method='Euler', dt=0.01)
def integral(V, w, t, Iext, a, b, tau):
    dw = (V + a - b * w) / tau
    dV = V - V * V * V / 3 - w + Iext
    return dV, dw
```

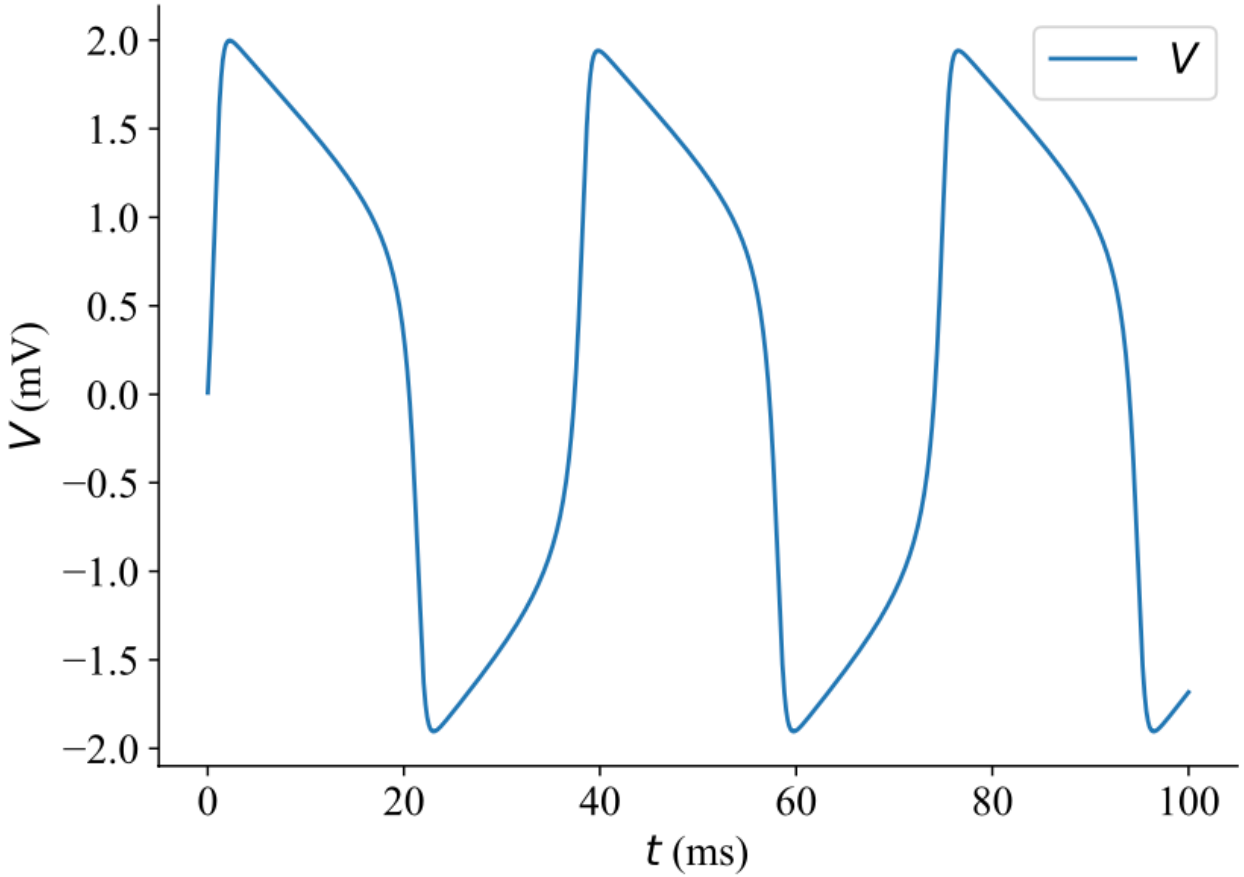
JointEq

In a dynamical system, there may be multiple variables that change dynamically over time. Sometimes these variables are interrelated, and updating one variable requires other variables as inputs. For better integration accuracy, we recommend that you use **brainpy.JointEq** to jointly solve interrelated differential equations.

```
a, b = 0.02, 0.20
dV = lambda V, t, w, Iext: 0.04 * V * V + 5 * V + 140 - w + Iext    # 第一个方程
dw = lambda w, t, V: a * (b * V - w)                                # 第二个方程
joint_eq = bp.JointEq(dV, dw)                                       # 联合微分方程
integral2 = bp.odeint(joint_eq, method='rk2')                       # 定义该联合微分方程的数值积分方法
```

Integrators

```
# 声明积分运行器
runner = bp.integrators.IntegratorRunner(
    integral,
    monitors=['V'],
    inits=dict(V=0., w=0.),
    args=dict(a=a, b=b, tau=tau, Iext=Iext),
    dt=0.01
)

# 使用积分运行器来进行模拟100ms，结合步长dt=0.01
runner.run(100.)

plt.plot(runner.mon.ts, runner.mon.V)
plt.show()
```



Dynamical System

A dynamical model often contains not only a continuous integration part, but also a discontinuous update step. For this reason, BrainPy provides a generic **DynamicalSystem** class to define various types of dynamical models.

BrainPy supports modelings in brain simulation and brain-inspired computing.

All these supports are based on one common concept: **Dynamical System** via `brainpy.DynamicalSystem`.

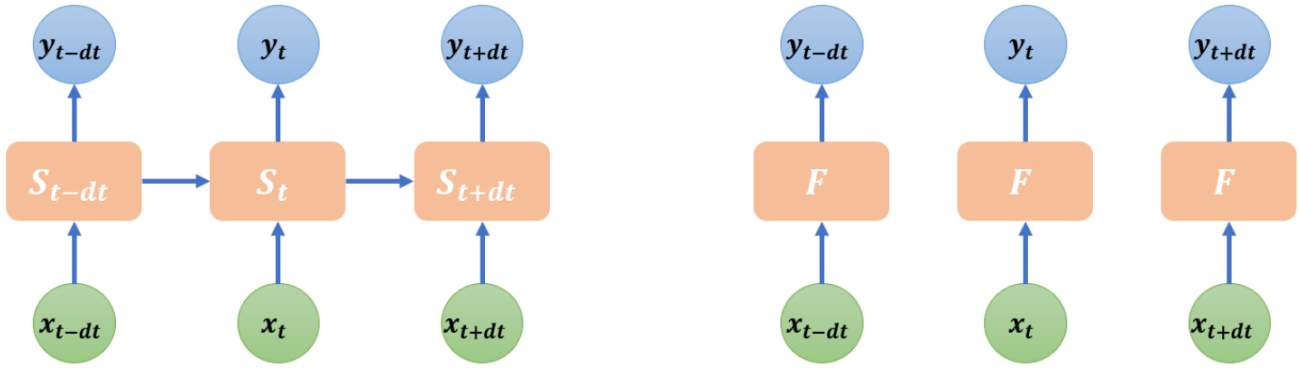
Therefore, it is essential to understand:

1. what is `brainpy.DynamicalSystem`?
2. how to define `brainpy.DynamicalSystem`?
3. how to run `brainpy.DynamicalSystem`?

What is DynamicalSystem

A `DynamicalSystem` defines the updating rule of the model at single time step.

1. For models with state, `DynamicalSystem` defines the state transition from t to $t + dt$, i.e., $S(t + dt) = F\left(S(t), x, t, dt\right)$, where S is the state, x is input, t is the time, and dt is the time step. This is the case for recurrent neural networks (like GRU, LSTM), neuron models (like HH, LIF), or synapse models which are widely used in brain simulation.
2. However, for models in deep learning, like convolution and fully-connected linear layers, `DynamicalSystem` defines the input-to-output mapping, i.e., $y = F\left(x, t\right)$.



Model with state

Model without state

How to define DynamicalSystem

北京大学 PEKING UNIVERSITY

```
class YourDynamicalSystem(bp.DynamicalSystem):
    def update(self, x):
        pass
```

Instead of input x, there are shared arguments across all nodes/layers in the network:

- the current time `t`, or
- the current running index `i`, or
- the current time step `dt`, or
- the current phase of training or testing `fit=True/False`.

How to define DynamicalSystem

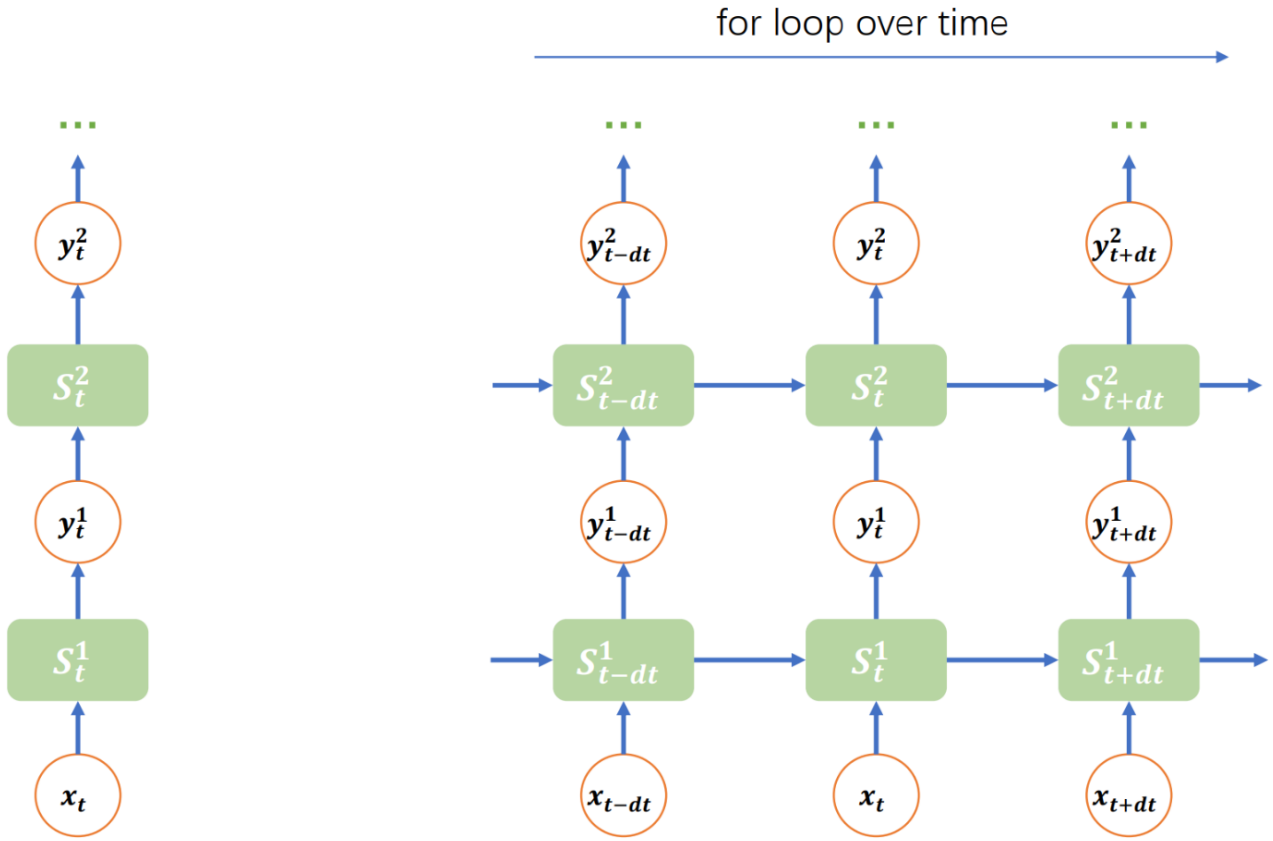
北京大学 PEKING UNIVERSITY

Here, it is necessary to explain the usage of `bp.share`.

- `bp.share.save( )`: The function saves shared arguments in the global context. User can save shared arguments in tow ways, for example, if user want to set the current time `t=100`, the current time step `dt=0.1`,the user can use `bp.share.save("t",100,"dt",0.1)` or `bp.share.save(t=100,dt=0.1)`.
- `bp.share.load( )`: The function gets the shared data by the `key`, for example, `bp.share.load("t")`.
- `bp.share.clear_shargs( )`: The function clears the specific shared arguments in the global context, for example, `bp.share.clear_shargs("t")`.
- `bp.share.clear( )`: The function clears all shared arguments in the global context.

How to run DynamicalSystem

As we have stated above that `DynamicalSystem` only defines the updating rule at single time step, to run a `DynamicalSystem` instance over time, we need a for loop mechanism.



DynamicalSystem

DSRunner / DSTrainer

How to run DynamicalSystem

北京大学 PEKING UNIVERSITY

1. `brainpy.math.for_loop`

`for_loop` is a structural control flow API which runs a function with the looping over the inputs. Moreover, this API just-in-time compile the looping process into the machine code.

```
inputs = bp.inputs.section_input([0., 6.0, 0.], [100. , 200., 100.])
indices = np.arange(inputs.size)


def run(i, x):
  neu.step_run(i, x)
  return neu.V.value


vs = bm.for_loop(run, (indices, inputs), progress_bar=True)
```

How to run DynamicalSystem

北京大学 PEKING UNIVERSITY

2. `brainpy.LoopOverTime`

Different from `for_loop`, `brainpy.LoopOverTime` is used for constructing a dynamical system that automatically loops the model over time when receiving an input.

`for_loop` runs the model over time. While `brainpy.LoopOverTime` creates a model which will run the model over time when calling it.

```
net.reset_state()
looper = bp.LoopOverTime(net)
out = looper(currents)
```

How to run DynamicalSystem

北京大学 PEKING UNIVERSITY

Initializing a `DSRunner`

Generally, we can initialize a runner for dynamical systems with the format of:

```
runner = DSRunner(target=instance_of_dynamical_system,
                  inputs=inputs_for_target_DynamicalSystem,
                  monitors=interested_variables_to_monitor,
                  dyn_vars=dynamical_changed_variables,
                  jit=enable_jit_or_not,
                  progress_bar=report_the_running_progress,
                  numpy_mon_after_run=transform_into_numpy_ndarray
                  )
```

Running a `DSRunner`

After initialization of the runner, users can call `.run()` function to run the simulation. The format of function `.run()` is showed as follows:

```
runner.run(duration=simulation_time_length,
           inputs=input_data,
           reset_state=whether_reset_the_model_states,
           shared_args=shared_arguments_across_different_layers,
           progress_bar=report_the_running_progress,
           eval_time=evaluate_the_running_time
           )
```

Simulation with `DSRunner`

https://brainpy.readthedocs.io/en/latest/tutorial_simulation/simulation_dsrunner.html

Monitors

北京大学 PEKING UNIVERSITY

Initialization with a list of strings

```
# initialize monitor through a list of strings
runner1 = bp.DSRunner(target=net,
                      monitors=['E.spike', 'E.V', 'I.spike', 'I.V'],
                      inputs=[('E.input', 20.), ('I.input', 20.)],
                      jit=True)
```

Once we call the runner with a given time duration, the monitor will automatically record the variable evolutions in the corresponding models. Afterwards, users can access these variable trajectories by using .mon.[variable_name]. The default history times .mon.ts will also be generated after the model finishes its running.

```
runner1.run(100.)
bp.visualize.raster_plot(runner1.mon.ts, runner1.mon['E.spike'], show=True)
```

Monitors

北京大学 PEKING UNIVERSITY

Initialization with index specification

```
monitors=[('E.spike', [1, 2, 3]),  # monitor values of Variable at index of [1, 2, 3]
          'E.V'],  # monitor all values of Variable 'V'
```

```
The monitor shape of "E.V" is (run length, variable size) = (1000, 3200)
The monitor shape of "E.spike" is (run length, index size) = (1000, 3)
```

Explicit monitor target

```
monitors={'spike': net.E.spike, 'V': net.E.V},
```

```
The monitor shape of "V" is = (1000, 3200)
The monitor shape of "spike" is = (1000, 3200)
```

Explicit monitor target with index specification

```
monitors={'E.spike': (net.E.spike, [1, 2]),  # monitor values of Variable at index of [1, 2]
          'E.V': net.E.V},  # monitor all values of Variable 'V'
```

```
The monitor shape of "E.V" is = (1000, 3200)
The monitor shape of "E.spike" is = (1000, 2)
```

Inputs

北京大学 PEKING UNIVERSITY

`inputs` should have the format like `(target, value, [type, operation])`, where

- `target` is the target variable to inject the input.
- `value` is the input value. It can be a scalar, a tensor, or a iterable object/function.
- `type` is the type of the input value. It support two types of input: `fix` and `iter`. The first one means that the data is static; the second one denotes the data can be iterable, no matter whether the input value is a tensor or a function. The `iter` type must be explicitly stated.
- `operation` is the input operation on the target variable. It should be set as one of `{ + , - , * , / , = }`, and if users do not provide this item explicitly, it will be set to '+' by default, which means that the target variable will be updated as `val = val + input`.

Inputs

北京大学 PEKING UNIVERSITY

Static inputs

```
runner6 = bp.DSRunner(target=net,
                      monitors=['E.spike'],
                      inputs=[('E.input', 20.), ('I.input', 20.)],  # static inputs
                      jit=True)
runner6.run(100.)
bp.visualize.raster_plot(runner6.mon.ts, runner6.mon['E.spike'])
```

Iterable inputs

```
I, length = bp.inputs.section_input(values=[0, 20., 0],
                                    durations=[100, 1000, 100],
                                    return_length=True,
                                    dt=0.1)

runner7 = bp.DSRunner(target=net,
                      monitors=['E.spike'],
                      inputs=[('E.input', I, 'iter'), ('I.input', I, 'iter')],
                      jit=True)
runner7.run(length)
bp.visualize.raster_plot(runner7.mon.ts, runner7.mon['E.spike'])
```

北京大学
PEKING UNIVERSITY

02

Run a built-in HH model

Run a built-in HH model

北京大学 PEKING UNIVERSITY

```
import brainpy as bp
import brainpy.math as bm

current, length = bp.inputs.section_input(values=[0., bm.asarray([1., 2., 4., 8., 10., 15.]), 0.],
                                          durations=[10, 2, 25],
                                          return_length=True)

hh_neurons = bp.neurons.HH(current.shape[1])

runner = bp.DSRunner(hh_neurons, monitors=['V', 'm', 'h', 'n'], inputs=('input', current, 'iter'))

runner.run(length)
```

https://brainpy.readthedocs.io/en/latest/tutorial_building/overview_of_dynamic_model.html

北京大学
PEKING UNIVERSITY

03

Build a HH model from scratch

The mathematic expression of the HH model

$$
\begin{cases}
c\dfrac{\mathrm{d}V}{\mathrm{d}t} = -\bar{g}_{\mathrm{Na}} m^3 h\,(V - E_{\mathrm{Na}}) - \bar{g}_{\mathrm{K}} n^4\,(V - E_{\mathrm{K}}) - \bar{g}_{\mathrm{L}}\,(V - E_{\mathrm{L}}) + I_{\mathrm{ext}}, \\
\dfrac{\mathrm{d}n}{\mathrm{d}t} = \phi\,[\alpha_n(V)(1-n) - \beta_n(V)n] \\
\dfrac{\mathrm{d}m}{\mathrm{d}t} = \phi\,[\alpha_m(V)(1-m) - \beta_m(V)m]\,, \\
\dfrac{\mathrm{d}h}{\mathrm{d}t} = \phi\,[\alpha_h(V)(1-h) - \beta_h(V)h]\,,
\end{cases}
$$

V: the membrane potential

n: activation variable of the K^+ channel

m: activation variable of the Na^+ channel

h: inactivation variable of the Na^+ channel

$$
\alpha_n(V) = \frac{0.01(V+55)}{1-\exp\left(-\frac{V+55}{10}\right)}, \quad \beta_n(V) = 0.125\exp\left(-\frac{V+65}{80}\right),
$$

$$
\alpha_h(V) = 0.07\exp\left(-\frac{V+65}{20}\right), \quad \beta_h(V) = \frac{1}{\left(\exp\left(-\frac{V+35}{10}\right)+1\right)},
$$

$$
\alpha_m(V) = \frac{0.1\,(V+40)}{1-\exp\left(-(V+40)/10\right)}, \quad \beta_m(V) = 4\exp\left(-(V+65)/18\right).
$$

α_x and β_x: voltage-dependent transition rates

$$
\phi = Q_{10}^{(T-T_{\mathrm{base}})/10}
$$

Programming of HH model in BrainPy

北京大学 PEKING UNIVERSITY

1.Define HH Model `class`

2. Initialization
   - parameters
   - variables
   - integral function

3. Define the derivative function

4. Complete the `update()` function

Programming of HH model in BrainPy

北京大学 PEKING UNIVERSITY

- Define HH Model `class`
  - Inherit bp.dyn.NeuDyn

```python
import brainpy as bp
import brainpy.math as bm

class HH(bp.dyn.NeuDyn):
```

```python
class Dynamic(DynamicalSystem):
  """Base class to model dynamics.
```

```python
class NeuDyn(Dynamic, AutoDelaySupp):
  """Neuronal Dynamics."""
```

Programming of HH model in BrainPy

北京大学 PEKING UNIVERSITY

- Define HH Model `class`
  - Inherit bp.dyn.NeuDyn

- Initialization

```
class HH(bp.dyn.NeuDyn):
  def __init__(self, size,
               ENa=50., gNa=120.,
               EK=-77., gK=36.,
               EL=-54.387, gL=0.03,
               V_th=0., C=1.0, T=6.3):
    super(HH, self).__init__(size=size)
```

Programming of HH model in BrainPy

北京大学 PEKING UNIVERSITY

- Define HH Model `class`
  - Inherit bp.dyn.NeuDyn

- Initialization
  - Parameters

```
class HH(bp.dyn.NeuDyn):
  def __init__(self, size,
               ENa=50., gNa=120.,
               EK=-77., gK=36.,
               EL=-54.387, gL=0.03,
               V_th=0., C=1.0, T=6.3):
    super(HH, self).__init__(size=size)

    # 定义神经元参数
    self.ENa = ENa
    self.EK = EK
    self.EL = EL
    self.gNa = gNa
    self.gK = gK
    self.gL = gL
    self.C = C
    self.V_th = V_th
    self.T_base = 6.3
    self.phi = 3.0 ** ((T - self.T_base) / 10.0)
```

Programming of HH model in BrainPy

北京大学 PEKING UNIVERSITY

- Define HH Model `class`
  - Inherit bp.dyn.NeuDyn
- Initialization
  - Parameters
  - Variables

```
class HH(bp.dyn.NeuDyn):
    def __init__(self, size,
                 ENa=50., gNa=120.,
                 EK=-77., gK=36.,
                 EL=-54.387, gL=0.03,
                 V_th=0., C=1.0, T=6.3):
        super(HH, self).__init__(size=size)

        # 定义神经元参数
        self.ENa = ENa
        self.EK = EK
# 定义神经元变量
self.V = bm.Variable(-70.68 * bm.ones(self.num))
self.m = bm.Variable(0.0266 * bm.ones(self.num))
self.h = bm.Variable(0.772 * bm.ones(self.num))
self.n = bm.Variable(0.235 * bm.ones(self.num))
self.input = bm.Variable(bm.zeros(self.num))
self.spike = bm.Variable(bm.zeros(self.num, dtype=bool))
self.t_last_spike = bm.Variable(bm.ones(self.num) * -1e7)
```

Programming of HH model in BrainPy

北京大学 PEKING UNIVERSITY

- Define HH Model `class`
  - Inherit bp.dyn.NeuDyn
- Initialization
  - Parameters
  - Variables
  - Integral function

```
class HH(bp.dyn.NeuDyn):
    def __init__(self, size,
                 ENa=50., gNa=120.,
                 EK=-77., gK=36.,
                 EL=-54.387, gL=0.03,
                 V_th=0., C=1.0, T=6.3):
        super(HH, self).__init__(size=size)

        # 定义神经元参数
        self.ENa = ENa
        # 定义神经元变量
        self.V = bm.Variable(-70.68 * bm.ones(self.num))
        self.m = bm.Variable(0.0266 * bm.ones(self.num))
        self.h = bm.Variable(0.772 * bm.ones(self.num))
        self.n = bm.Variable(0.235 * bm.ones(self.num))
        self.input = bm.Variable(bm.zeros(self.num))
        self.spike = bm.Variable(bm.zeros(self.num, dtype=bool))
        self.t_last_spike = bm.Variable(bm.ones(self.num) * -1e7)
        # 定义积分函数
        self.integral = bp.odeint(f=self.derivative, method='exp_auto')
```

Programming of HH model in BrainPy

北京大学 PEKING UNIVERSITY

- Define HH Model `class`
  - Inherit bp.dyn.NeuDyn
- Initialization
  - Parameters
  - Variables
  - Integral function
- Derivative function

$$
\begin{cases}
c\dfrac{\mathrm{d}V}{\mathrm{d}t} = -\bar{g}_{\mathrm{Na}} m^3 h\,(V - E_{\mathrm{Na}}) - \bar{g}_{\mathrm{K}} n^4\,(V - E_{\mathrm{K}}) - \bar{g}_{\mathrm{L}}\,(V - E_{\mathrm{L}}) + I_{\mathrm{ext}}, \\
\dfrac{\mathrm{d}n}{\mathrm{d}t} = \phi\,[\alpha_n(V)(1-n) - \beta_n(V) n] \\
\dfrac{\mathrm{d}m}{\mathrm{d}t} = \phi\,[\alpha_m(V)(1-m) - \beta_m(V) m]\,, \\
\dfrac{\mathrm{d}h}{\mathrm{d}t} = \phi\,[\alpha_h(V)(1-h) - \beta_h(V) h]\,,
\end{cases}
$$

```python
@property
def derivative(self):
  return bp.JointEq(self.dV, self.dm, self.dh, self.dn)

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

def dm(self, m, t, V):
  alpha = 0.1 * (V + 40) / (1 - bm.exp(-(V + 40) / 10))
  beta = 4.0 * bm.exp(-(V + 65) / 18)
  dmdt = alpha * (1 - m) - beta * m
  return self.phi * dmdt

def dh(self, h, t, V):
  alpha = 0.07 * bm.exp(-(V + 65) / 20.)
  beta = 1 / (1 + bm.exp(-(V + 35) / 10))
  dhdt = alpha * (1 - h) - beta * h
  return self.phi * dhdt

def dn(self, n, t, V):
  alpha = 0.01 * (V + 55) / (1 - bm.exp(-(V + 55) / 10))
  beta = 0.125 * bm.exp(-(V + 65) / 80)
  dndt = alpha * (1 - n) - beta * n
  return self.phi * dndt
```

Programming of HH model in BrainPy

北京大学 PEKING UNIVERSITY

- Define HH Model `class`
  - Inherit bp.dyn.NeuDyn
- Initialization
  - Parameters
  - Variables
  - Integral function
- Derivative function
- Update function

```
def update(self, x=None):
  t = bp.share.load('t')
  dt = bp.share.load('dt')
  #计算更新后的值
  V, m, h, n = self.integral(self.V, self.m, self.h, self.n, t, self.input, dt=dt)

  #判断是否发生动作电位
  self.spike.value = bm.logical_and(self.V < self.V_th, V >= self.V_th)
  self.t_last_spike.value = bm.where(self.spike, t, self.t_last_spike)

  # 更新变量的值
  self.V.value = V
  self.m.value = m
  self.h.value = h
  self.n.value = n

  #重置输入
  self.input[:] = 0.
```

Simulation

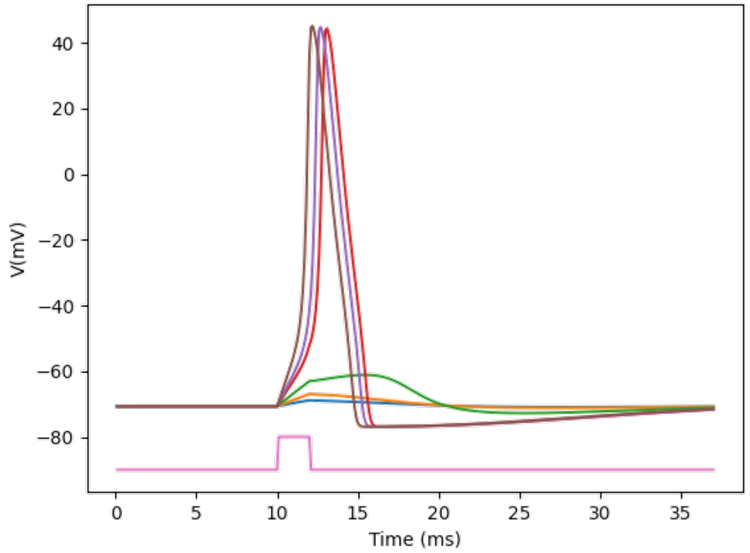
北京大学 PEKING UNIVERSITY

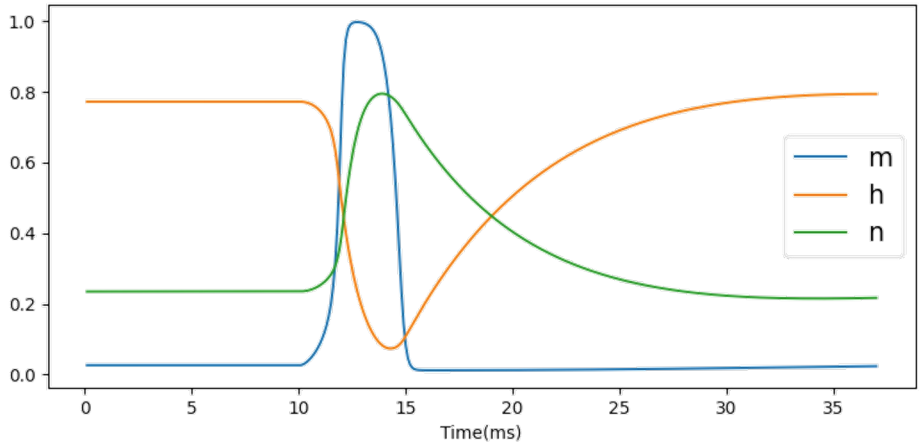
```python
current, length = bp.inputs.section_input(values=[0., bm.asarray([1., 2., 4., 8., 10., 15.]), 0.],
                                          durations=[10, 2, 25],
                                          return_length=True)

hh_neurons = HH(current.shape[1])

runner = bp.DSRunner(hh_neurons, monitors=['V', 'm', 'h', 'n'], inputs=('input', current, 'iter'))

runner.run(length)
```

```
Predict 370 steps: : 100% 370/370 [00:00<00:00, 6.38it/s]
```

Visualization

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

bp.visualize.line_plot(runner.mon.ts, runner.mon.V, ylabel='V (mV)', plot_ids=np.arange(current.shape[1]))

plt.plot(runner.mon.ts, bm.where(current[:, -1]>0, 10, 0) - 90.)

plt.figure()

plt.plot(runner.mon.ts, runner.mon.m[:, -1])
plt.plot(runner.mon.ts, runner.mon.h[:, -1])
plt.plot(runner.mon.ts, runner.mon.n[:, -1])
plt.legend(['m', 'h', 'n'])
plt.xlabel('Time (ms)')
```





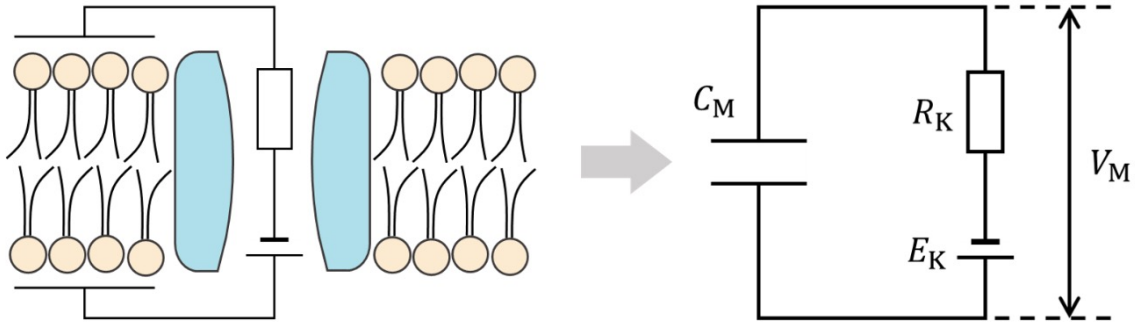
北京大学
PEKING UNIVERSITY

04

Customize a conductance-based model

Building an ion channel

As we have known, ion channels are crucial for conductance-based neuron models. So how do we model an ion channel? Let's take a look at the potassium channel for instance.



The diagram above shows how a potassium channel is changed to an electric circuit. By this, we have the differential equation:

$$\begin{aligned} c_{\mathrm{M}} \frac{\mathrm{d}V_{\mathrm{M}}}{\mathrm{d}t} &= \frac{E_{\mathrm{K}} - V_{\mathrm{M}}}{R_{\mathrm{K}}} \\ &= g_{\mathrm{K}}(E_{\mathrm{K}} - V_{\mathrm{M}}), \end{aligned}$$

in which c_{M} is the membrane capacitance, $\mathrm{d}V_{\mathrm{M}}$ is the membrane potential, E_{K} is the equilibrium potential of potassium ions, and R_{K} (g_{K}) refers to the resistance (conductance) of the potassium channel. We define currents from inside to outside as the positive direction.

Building an ion channel

Fortunately, there has been a lot of work addressing this issue to formulate analytical expressions. For example, the conductance of one typical potassium channel can be written as:

$$
\begin{aligned}
g_{\mathrm{K}} &= \bar{g}_{\mathrm{K}} n^4, \\
\frac{\mathrm{d}n}{\mathrm{d}t} &= \phi[\alpha_n(V)(1-n) - \beta_n(V)n],
\end{aligned}
$$

in which $\bar{g}_{\mathrm{K}}$ refers to the maximal conductance and n, also named the gating variable, refers to the probability (proportion) of potassium channels to open. ϕ is a parameter showing the effects of temperature. In the differential equation of n, there are two parameters, $\alpha_n(V)$ and $\beta_n(V)$, that change with membrane potential:

$$
\begin{aligned}
\alpha_n(V) &= \frac{0.01(V+55)}{1-\exp(-\frac{V+55}{10})}, \\
\beta_n(V) &= 0.125\exp\left(-\frac{V+65}{80}\right).
\end{aligned}
$$

Programming an ion channel

```python
class IK(bp.dyn.IonChannel):
  def __init__(self, size, E=-77., g_max=36., phi=1., method='exp_auto'):
    super(IK, self).__init__(size)
    self.g_max = g_max
    self.E = E
    self.phi = phi

    self.n = bm.Variable(bm.zeros(size))  # variables should be packed with bm.Variable

    self.integral = bp.odeint(self.dn, method=method)

  def dn(self, n, t, V):
    alpha_n = 0.01 * (V + 55) / (1 - bm.exp(-(V + 55) / 10))
    beta_n = 0.125 * bm.exp(-(V + 65) / 80)
    return self.phi * (alpha_n * (1. - n) - beta_n * n)

  def update(self, V):
    t = bp.share.load('t')
    dt = bp.share.load('dt')
    self.n.value = self.integral(self.n, t, V, dt=dt)

  def current(self, V):
    return self.g_max * self.n ** 4 * (self.E - V)
```

Note that besides the initialzation and update function, **another function named `current()` that computes the current flow through this channel must be implemented**. Then this potassium channel model can be used as a building block for assembling a conductance-based neuron model.

Programming an ion channel

北京大学 PEKING UNIVERSITY

```python
class INa(bp.dyn.IonChannel):
  def __init__(self, size, E= 50., g_max=120., phi=1., method='exp_auto'):
    super(INa, self).__init__(size)
    self.g_max = g_max
    self.E = E
    self.phi = phi

    self.m = bm.Variable(bm.zeros(size))  # variables should be packed with bm.Variable
    self.h = bm.Variable(bm.zeros(size))

    self.integral_m = bp.odeint(self.dm, method=method)
    self.integral_h = bp.odeint(self.dh, method=method)


  def dm(self, m, t, V):
    alpha_m = 0.11 * (V + 40) / (1 - bm.exp (-(V + 40) / 10))
    beta_m = 4* bm.exp(-(V + 65) /18)
    return self.phi * (alpha_m * (1. - m) - beta_m * m)
  def dh(self, h, t, V):
    alpha_h = 0.07 * bm.exp(-(V + 65) / 20)
    beta_h = 1. / (1 + bm.exp(-(V + 35) / 10))
    return self.phi * (alpha_h * (1. - h) - beta_h * h)

  def update(self, V):
    t = bp.share.load('t')
    dt = bp.share.load('dt')
    self.m.value = self.integral_m(self.m, t, V, dt=dt)
    self.h.value = self.integral_h(self.h, t, V, dt=dt)

  def current(self, V):
    return self.g_max * self.m ** 3 * self.h * (self.E - V)
```

```python
class IL(bp.dyn.IonChannel):
  def __init__(self, size, E=-54.39, g_max=0.03):
    super(IL, self).__init__(size)
    self.g_max = g_max
    self.E = E

  def current(self, V):
    return self.g_max * (self.E - V)
  def update(self, V):
    pass
```

Build a HH model with ion channels

北京大学 PEKING UNIVERSITY

- Using customized ion channels

```
class HH(bp.dyn.CondNeuGroup):
  def __init__(self, size):
    super(HH, self).__init__(size, V_initializer=bp.init.Uniform(-80, -60.))
    self.IK = IK(size, E=-77., g_max=36.)
    self.INa = INa(size, E=50., g_max=120.)
    self.IL = IL(size, E=-54.39, g_max=0.03)
```

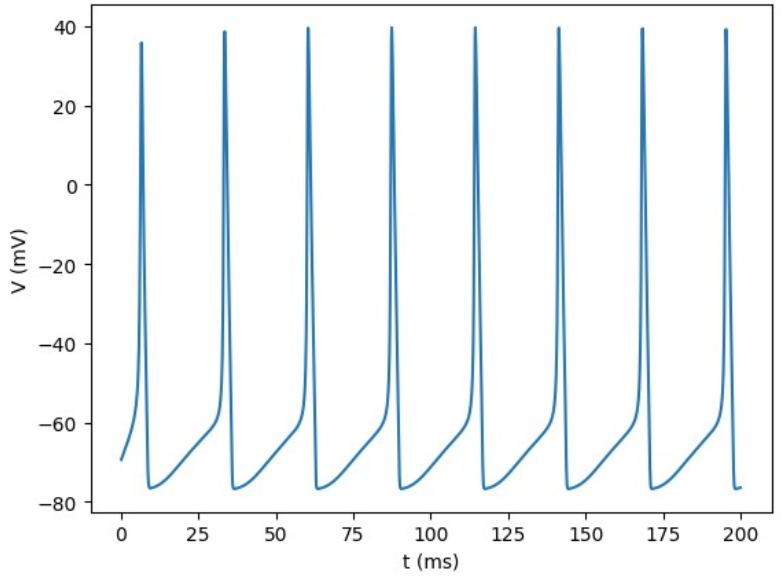
- Using built-in ion channels

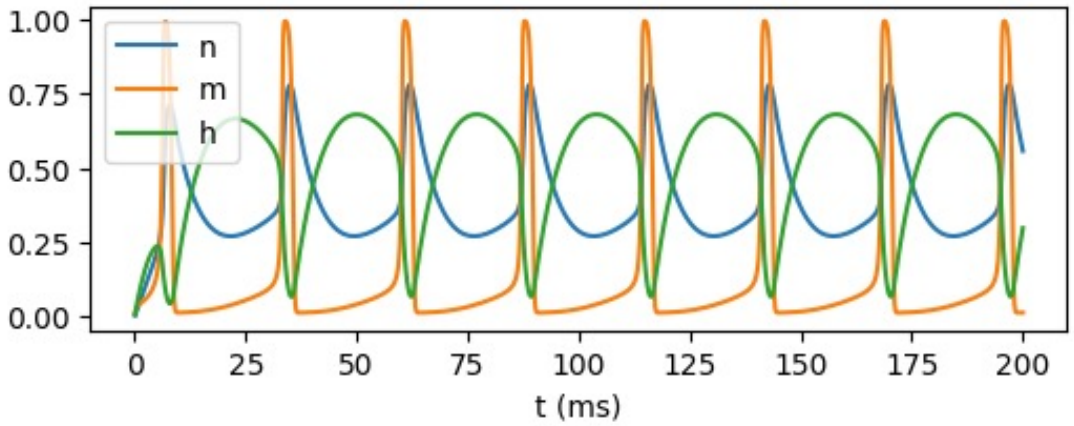
```
class HH(bp.dyn.CondNeuGroup):
  chaoming +1
  def __init__(self, size):
    super().__init__(size)

    self.INa = bp.channels.INa_HH1952(size)
    self.IK = bp.channels.IK_HH1952(size)
    self.IL = bp.channels.IL(size, E=-54.387, g_max=0.03)
```

Simulation

北京大学 PEKING UNIVERSITY

```python
neu = HH(1)

runner = bp.DSRunner(
    neu,
    monitors=['V', 'IK.n', 'INa.m', 'INa.h'],
    inputs=('input', 1.698)  # near the threshold current
)

runner.run(200)  # the running time is 200 ms

import matplotlib.pyplot as plt

plt.plot(runner.mon['ts'], runner.mon['V'])
plt.xlabel('t (ms)')
plt.ylabel('V (mV)')
plt.savefig("HH.jpg")
plt.show()

plt.figure(figsize=(6, 2))
plt.plot(runner.mon['ts'], runner.mon['IK.n'], label='n')
plt.plot(runner.mon['ts'], runner.mon['INa.m'], label='m')
plt.plot(runner.mon['ts'], runner.mon['INa.h'], label='h')
plt.xlabel('t (ms)')
plt.legend()
plt.savefig("HH_channels.jpg")

plt.show()
```





Thanks!

北京大学
PEKING UNIVERSITY