Implement a Hodgkin-Huxley neuron model with BrainPy

BrainPy Overview

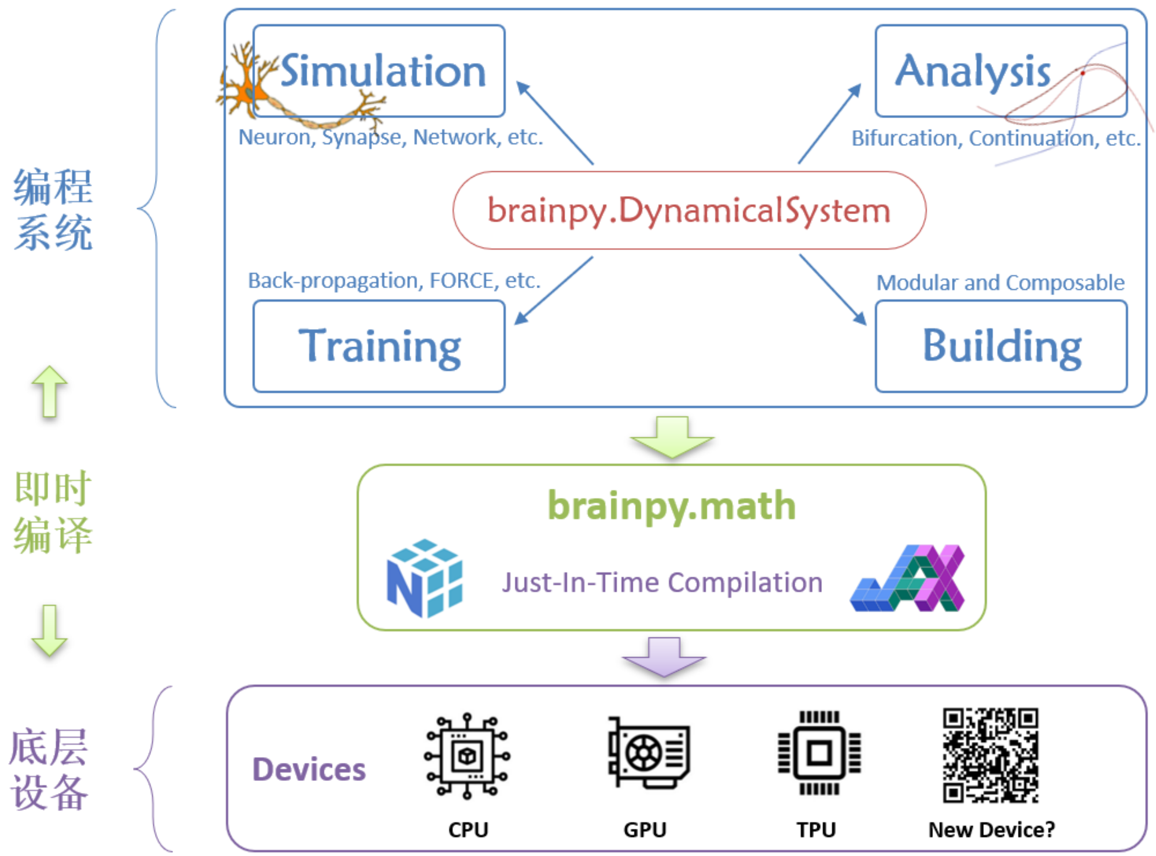
What is BrainPy?

BrainPy is a Python library designed for high-performance flexible brain modeling.

Among its key ingredients it supports:

- **General numerical solvers**
  - ☐ Ordinary differential equations
  - ☐ Stochastic differential equations
  - ☐ Delayed differential equations
  - ☐ Fractional differential equations
- **Neurodynamics simulation tools**
  - ☐ Support brain objects, such like neurons, synapses, networks, soma, dendrites, channels, and even molecular.
- **Neurodynamics analysis tools**
  - ☐ Support phase plane analysis and bifurcation analysis, continuation analysis.

A Just-In-Time compilation approach for brain modeling



Model definition, building and simulation are all done in Python.

- Easy to learn and use
- Efficient
- Flexible
- Transparent
- Extensible

```
>  pip install brain-py
```

JIT in BrainPy

brainpy.math

Just-In-Time Compilation

Numba

- Prefer loops, support Python control syntaxes

- Has poor parallel performance
- Same code cannot be used to run GPUs
- Poor performance for class objects

- Prefer large networks, and has good parallel performance
- Same code can be deployed onto CPUs, GPUs and TPUs
- Support automatic differentiation

- Not support in-place updates, like x[i] += y
- Random numbers are different from NumPy
- Do not support direct Python control flows
- Intrinsic overhead, and is not suitable to run small networks
- Only work on pure functions

- We do not implement our own JIT compilation
- Instead, we choose mature industry-level JIT compilers available right now

brinpy.math module

- flexible switch between NumPy/Numba and JAX backends

```
[2]: # switch to NumPy backend
     bp.math.use_backend('numpy')
```

```
[3]: # switch to JAX backend
     bp.math.use_backend('jax')
```

- unified numpy-like array operations

```
[6]: x = bp.math.array([[1,2], [3,4]])

     x

[6]: JaxArray(DeviceArray([[1, 2],
                           [3, 4]], dty
```

```
[9]: bp.math.repeat(x, 2, axis=1)

[9]: JaxArray(DeviceArray([[1, 1, 2, 2],
                           [3, 3, 4, 4]],
```

- unified ndarray data structure which supports in-place update
  - JaxArray
  - NumPyArray

- unified random APIs

- powerful jit() compilation which supports functions and class objects both

NumPy-like operators in BrainPy

- brainpy.base module
- brainpy.math module
  - General Functions
  - Core Functions in NumPy backend
  - Core Functions in JAX backend
  - Function Wrapper
  - Comparison Table
    - Multi-dimensional Array
    - Array Operations
    - Linear Algebra
    - Discrete Fourier Transform
    - Random Sampling
- brainpy.integrators module
- brainpy.simulation module
- brainpy.analysis module
- brainpy.dnn module
- brainpy.visualize module
- Release notes

Read the Docs v: latest

Linear Algebra

NumPy	brainpy.math.numpy	brainpy.math.jax
numpy.linalg.cholesky	brainpy.math.numpy.linalg.cholesky	brainpy.math.jax.linalg.cl
numpy.linalg.cond	brainpy.math.numpy.linalg.cond	brainpy.math.jax.linalg.co
numpy.linalg.det	brainpy.math.numpy.linalg.det	brainpy.math.jax.linalg.de
numpy.linalg.eig	brainpy.math.numpy.linalg.eig	brainpy.math.jax.linalg.e
numpy.linalg.eigh	brainpy.math.numpy.linalg.eigh	brainpy.math.jax.linalg.e
numpy.linalg.eigvals	brainpy.math.numpy.linalg.eigvals	brainpy.math.jax.linalg.e
numpy.linalg.eigvalsh	brainpy.math.numpy.linalg.eigvalsh	brainpy.math.jax.linalg.e
numpy.linalg.inv	brainpy.math.numpy.linalg.inv	brainpy.math.jax.linalg.in
numpy.linalg.lstsq	brainpy.math.numpy.linalg.lstsq	brainpy.math.jax.linalg.ls
numpy.linalg.matrix_power	brainpy.math.numpy.linalg.matrix_power	brainpy.math.jax.linalg.ma
numpy.linalg.matrix_rank	brainpy.math.numpy.linalg.matrix_rank	brainpy.math.jax.linalg.ma
numpy.linalg.multi_dot	-	-
numpy.linalg.norm	brainpy.math.numpy.linalg.norm	brainpy.math.jax.linalg.no
numpy.linalg.pinv	brainpy.math.numpy.linalg.pinv	brainpy.math.jax.linalg.p

JIT for objects in BrainPy

Any instance of **brainpy.Base** object can be just-in-time compiled into machine codes.

- A "self." accessed variable which is not an instance of ***bp.math.Variable*** will be compiled as a static constant.
- All the variables you want to change during the function call must be labeled as ***bp.math.Variable***.
- The dynamically changed variables must be in-place updated to hold their updated values.

1. **Indexing and slicing**. Like (More details

- Index: `v[i] = a`
- Slice: `v[i:j] = b`
- Slice the specific values: `v[[1, 3]] = c`
- Slice all values, `v[:] = d`, `v[...] = e`

2. **Augmented assignment**.

- `+=` (add)
- `-=` (subtract)
- `/=` (divide)
- `*=` (multiply)
- `//=` (floor divide)

```python
class LogisticRegression(bp.Base):
    def __init__(self, dimension):
        super(LogisticRegression, self).__init__()

        # parameters
        self.dimension = dimension

        # variables
        self.w = bp.math.Variable(2.0 * bp.math.ones(dimension) - 1.3)

    def __call__(self, X, Y):
        u = bp.math.dot(((1.0 / (1.0 + bp.math.exp(
            -Y * bp.math.dot(X, self.w))) - 1.0) * Y), X)
        self.w[:] = self.w - u


num_dim, num_points = 10, 20000000
num_iter = 30

points = bp.math.random.random((num_points, num_dim))
labels = bp.math.random.random(num_points)
```

```python
# numpy backend, without JIT

lr1 = LogisticRegression(num_dim)
lr1(points, labels)

import time
t0 = time.time()
for i in range(num_iter):
    lr1(points, labels)

print(f'Logistic Regression model without jit used time {time.time() - t0} s')
```

```
Logistic Regression model without jit used time 19.143301725387573 s
```

```python
# numpy backend, with JIT + parallel

lr3 = LogisticRegression(num_dim)
jit_lr3 = bp.math.jit(lr3, parallel=True)
jit_lr3(points, labels)  # first call is the compiling

t0 = time.time()
for i in range(num_iter):
    jit_lr3(points, labels)

print(f'Logistic Regression model with jit+parallel used time {time.time() - t0} s')
```

```
Logistic Regression model with jit+parallel used time 7.351796865463257 s
```

Coding HH model with ODE numerical solver

Brain modeling by using differential equations

- Neuronal activities can be described by a set of differential equations.

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

- **Basic question**: How to solve the differential equations?

$$x(t) = ?$$

- Single neuron modeling --- Hodgkin-Huxley equations

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

$$\boxed{V(t) = ?}$$

Methods to solve differential equations

- Get algebraic solution

$$\frac{dy}{dx} = x^2 - 3 \quad \Rightarrow \quad y = \frac{x^3}{3} - 3x + K$$

$$\frac{d\theta}{dt} = \frac{\sin(t+0.2)}{\theta^2} \quad \Rightarrow \quad \frac{\theta^3}{3} = -\cos(t+0.2) + K$$

- Numerical integration

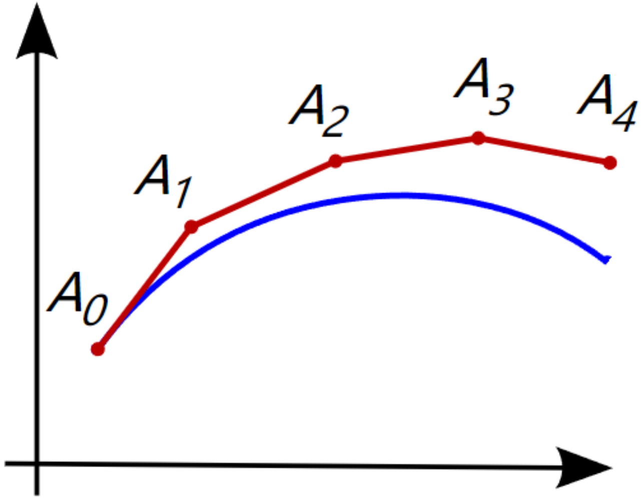
Euler's Method

$$y(t+dt) \approx y(t) + dt\,y'(t) + \frac{dt^2 y''(t)}{2!} + \frac{dt^3 y'''(t)}{3!} + \frac{dt^4 y^{\mathrm{iv}}(t)}{4!} + \ldots$$

$$y(t+dt) \approx y(t) + dt\,y'(t)$$

Solving HH neuron model by Euler method

$$m_t = m_{t-1} + \left[\alpha_m(V_{t-1})(1 - m_{t-1}) - \beta_m(V_{t-1})m_{t-1}\right] * dt$$
$$h_t = h_{t-1} + \left[\alpha_h(V_{t-1})(1 - h_{t-1}) - \beta_h(V_{t-1})h_{t-1}\right] * dt$$
$$n_t = n_{t-1} + \left[\alpha_n(V_{t-1})(1 - n_{t-1}) - \beta_n(V_{t-1})n_{t-1}\right] * dt$$
$$V_t = V_{t-1} + \left[\frac{-\bar{g}_{\mathrm{K}} n_{t-1}^4 (V_{t-1} - V_K) - \bar{g}_{\mathrm{Na}} m_{t-1}^3 h_{t-1}(V_{t-1} - V_{Na}) - \bar{g}_l(V_{t-1} - V_l) + I_{syn}}{C_m}\right] * dt$$



Define HH model with *brainpy.NeuGroup*

```python
class HH(bp.NeuGroup):
  def __init__(self, size, ENa=50., gNa=120., EK=-77., gK=36., EL=-54.387,
               gL=0.03, V_th=20., C=1.0, **kwargs):
    # 初始化父类
    super(HH, self).__init__(size=size, **kwargs)

    # 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

    # 定义神经元变量
    self.V = bm.Variable(-65. * bm.ones(self.num))  # 膜电位
    self.m = bm.Variable(0.5 * bm.ones(self.num))  # 离子通道m
    self.h = bm.Variable(0.6 * bm.ones(self.num))  # 离子通道h
    self.n = bm.Variable(0.32 * bm.ones(self.num))  # 离子通道n
    self.input = bm.Variable(bm.zeros(self.num))  # 神经元接收到的输入电流
    self.spike = bm.Variable(bm.zeros(self.num, dtype=bool))  # 神经元的发放状态
    self.t_last_spike = bm.Variable(bm.ones(self.num) * -1e7)  # 神经元上次发放的时刻
```

Initialize Parameters

Initialize Variables

```python
def update(self, _t, _dt, **kwargs):
  V, m, h, n = self.V, self.m, self.h, self.n

  # 更新下一时刻变量的值
  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
  self.m += dmdt * _dt

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

  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
  self.n += dndt * _dt

  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 + self.input) / self.C
  V = self.V + dVdt * _dt

  # 判断神经元是否产生膜电位
  self.spike[:] = bm.logical_and(self.V < self.V_th, V >= self.V_th)
  # 更新神经元发放的时间
  self.t_last_spike[:] = bm.where(self.spike, _t, self.t_last_spike)
  self.V[:] = V
  self.input[:] = 0.  # 重置神经元接收到的输入
```

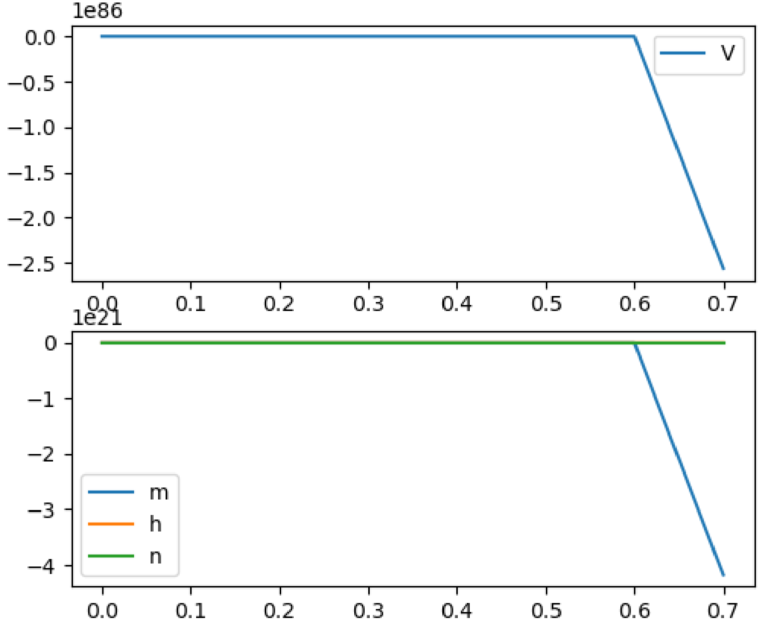
$$\frac{dm}{dt} = \alpha_m(1-m) - \beta_m$$

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

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

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

```python
def show(mon):
  plt.subplot(211)
  plt.plot(mon.ts, mon.V[:, 0], label='V')
  plt.legend()
  plt.subplot(212)
  plt.plot(mon.ts, mon.m[:, 0], label='m')
  plt.plot(mon.ts, mon.h[:, 0], label='h')
  plt.plot(mon.ts, mon.n[:, 0], label='n')
  plt.legend()
  plt.show()
```



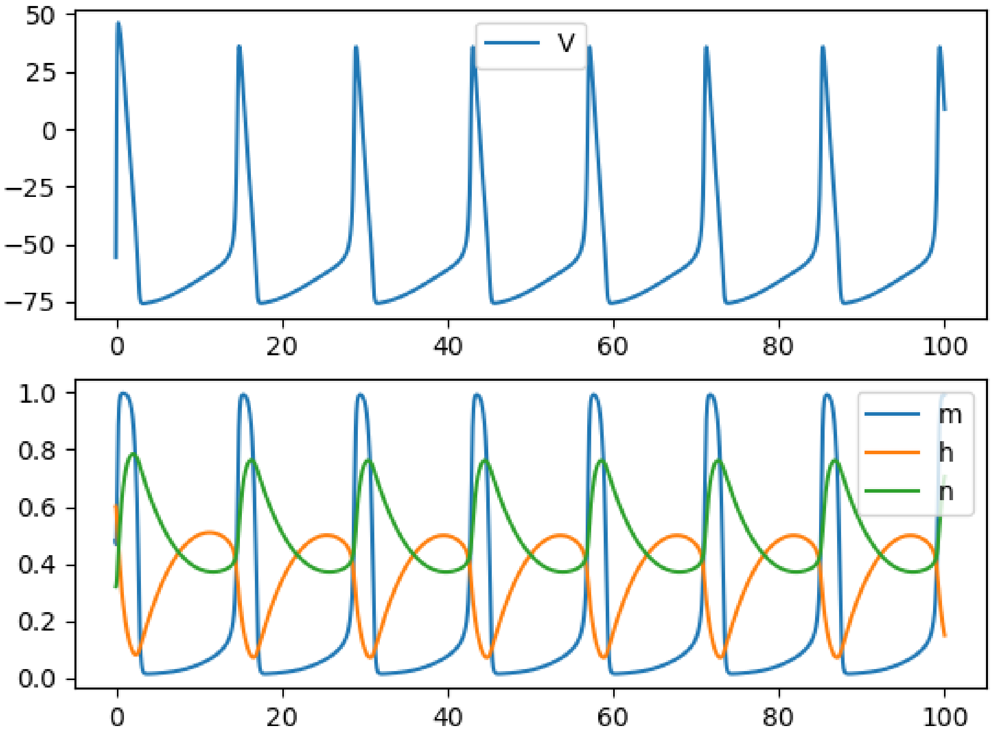

```python
hh = HHEuler(1, monitors=['V', 'm', 'n', 'h'])
hh.run(100, inputs=['input', 10], report=0.1, dt=0.1,)

show(hh.mon)
```

强大的inputs支持，(key, value, ops), 支持 +, -, *, /, = 赋值

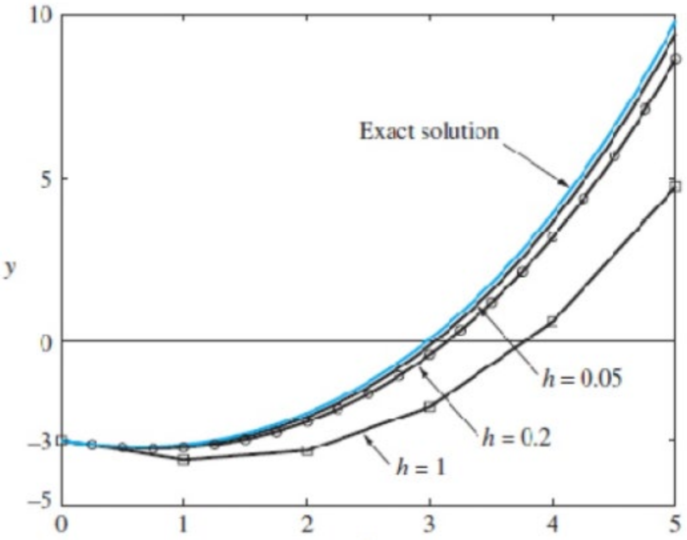
```python
hh = HHEuler(1, monitors=['V', 'm', 'n', 'h'])
hh.run(100, inputs=['input', 10], report=0.1, dt=0.01,)

show(hh.mon)
```

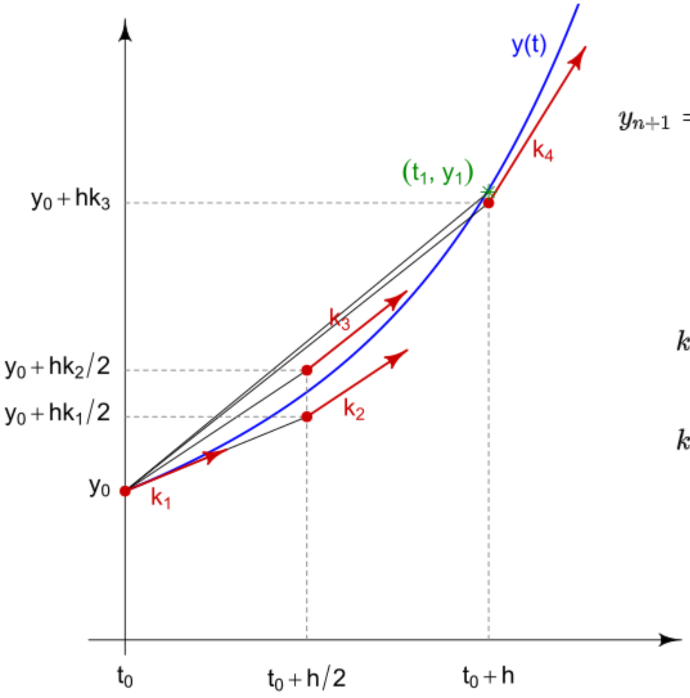


We need more high-order methods

Euler 方法



RK4 方法



$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$k_1 = f(t_n, y_n)$$

$$k_2 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right)$$

$$k_3 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right)$$

$$k_4 = f(t_n + h, y_n + hk_3)$$

Support for Ordinary Differential Equations in BrainPy

An ODE system

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

ODE as a Python function

- Can be a **scalar**
- Can be a **vector / matrix**

Variables

Parameters

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

- Can be a **system**: group of variables

```python
import numpy as np

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

Simple decorator for numerical integration

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

Numerical method

Numerical precision

Supported ODE Numerical Methods

Runge-Kutta Methods	Adaptive Runge-Kutta Methods	Other Methods
Euler	Runge–Kutta–Fehlberg 4(5)	Exponential Euler
Midpoint	Runge–Kutta–Fehlberg 1(2)	
Heun's second-order method	Dormand–Prince method	
Ralston's second-order method	Cash–Karp method	
RK2	Bogacki–Shampine method	
RK3	Heun–Euler method	
RK4		
Heun's third-order method		
Ralston's third-order method		
Third-order Strong Stability Preserving Runge-Kutta		
Ralston's fourth-order method		
Runge-Kutta 3/8-rule fourth-order method		

Define HH model with *brainpy.NeuGroup* and *brainpy.odeint*

```python
@bp.odeint(method='exponential_euler')
def integral(self, V, m, h, n, t, Iext):
  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

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

  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

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

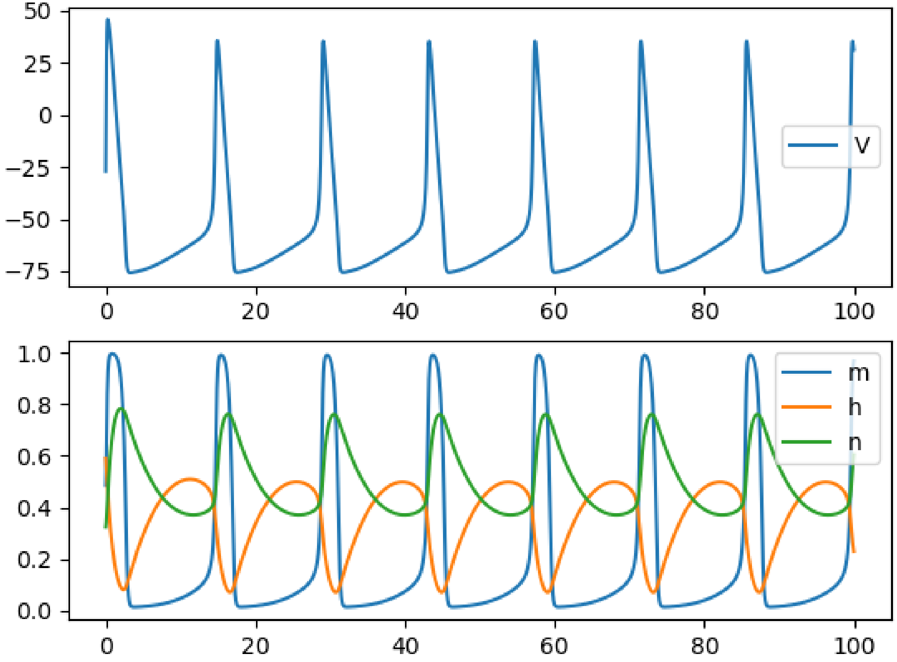
```python
def update(self, _t, _dt, **kwargs):
    # 更新下一时刻变量的值
    V, m, h, n = self.integral(self.V, self.m, self.h, self.n, _t, self.input, dt=_dt)
    # 判断神经元是否产生膜电位
    self.spike[:] = bm.logical_and(self.V < self.V_th, V >= self.V_th)
    # 更新神经元发放的时间
    self.t_last_spike[:] = bm.where(self.spike, _t, self.t_last_spike)
    self.V[:] = V
    self.m[:] = m
    self.h[:] = h
    self.n[:] = n
    self.input[:] = 0.  # 重置神经元接收到的输入
```

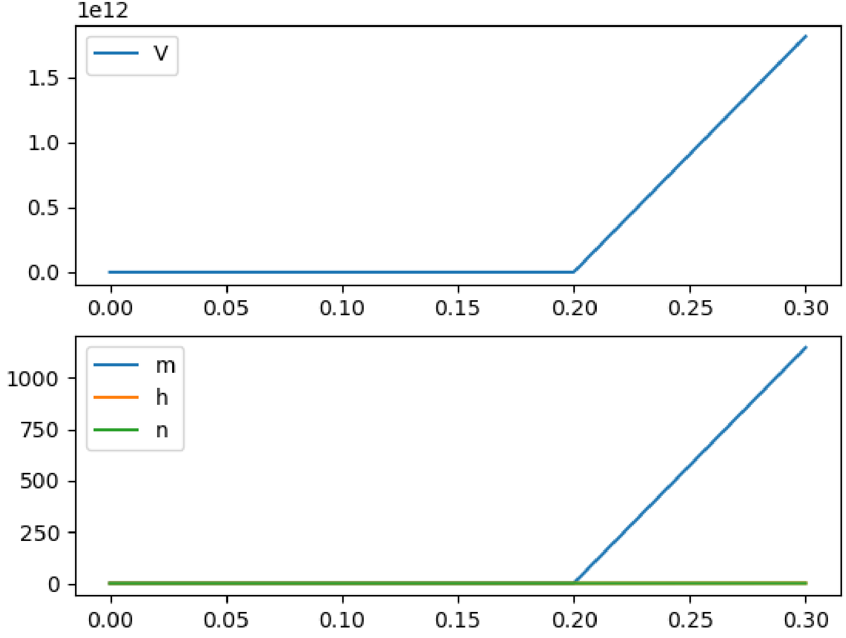
```python
hh = HH(1, monitors=['V', 'm', 'n', 'h'], method='rk4')
hh.run(100, inputs=['input', 10], report=0.1, dt=0.05,)

show(hh.mon)
```

```python
hh = HH(1, monitors=['V', 'm', 'n', 'h'], method='rk4')
hh.run(100, inputs=['input', 10], report=0.1, dt=0.1,)

show(hh.mon)
```





指数Euler方法

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

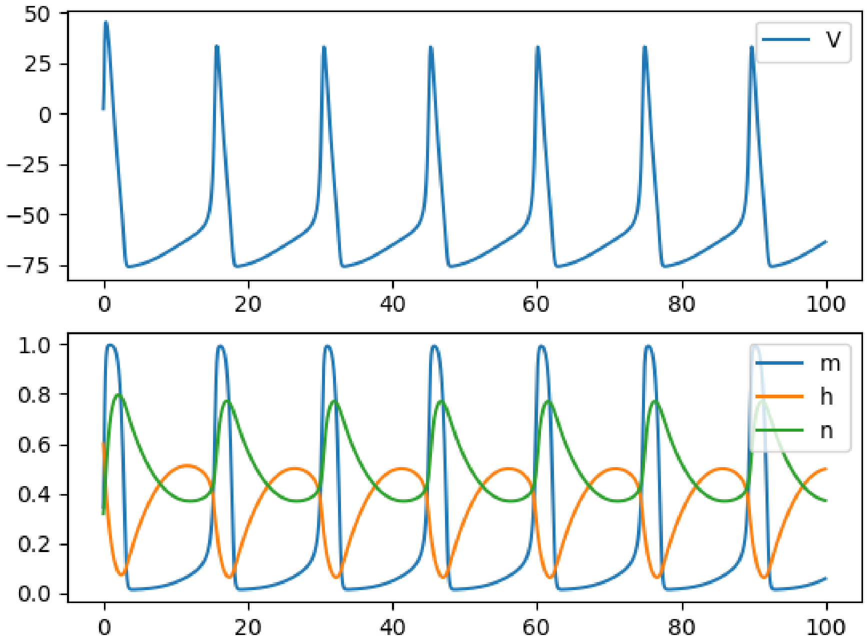
For linear ODE system: $y' = Ay + B$, the exponential euler schema is equal to

$$y_{n+1} = y_n e^{hA} - B/A(1 - e^{hA})$$

where $A = f'(y_n)$

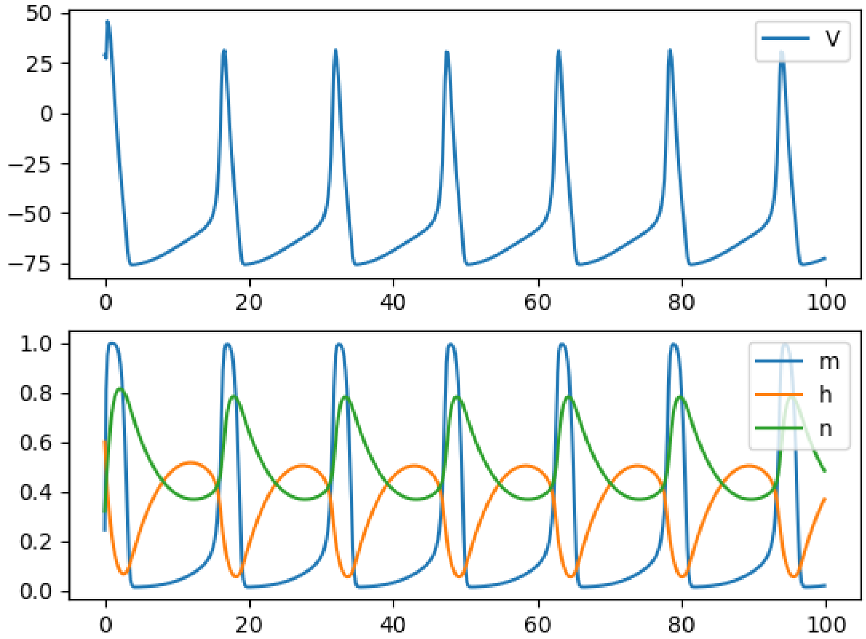
```python
hh = HH(1, monitors=['V', 'm', 'n', 'h'],
        method='exponential_euler')
hh.run(100, inputs=['input', 10], report=0.1, dt=0.1,)

show(hh.mon)
```




```python
hh = HH(1, monitors=['V', 'm', 'n', 'h'],
        method='exponential_euler')
hh.run(100, inputs=['input', 10], report=0.1, dt=0.2,)

show(hh.mon)
```



Homework

课后作业 Homework

1. 安装Anaconda/miniconda Python环境(https://docs.anaconda.com/anaconda/install/)

2. 安装 BrainPy == 1.1.0rc5 (https://pypi.org/project/brain-py/1.1.0rc5/)

3. 阅读：
   - https://brainmodels.readthedocs.io/en/latest/apis/generated/brainmodels.neurons.HH.html
   - https://brainmodels.readthedocs.io/en/latest/apis/neurons/HH_model.html

4. 【**提交作业，代码 + report (配图)**】实现上述 HH 模型

5. 将“代码+report”打包以“姓名_学号_HH报告.zip”命名, 发送到 chutianhao@stu.pku.edu.cn, 10.26截至