Instituto Tecnológico de Buenos Aires

93.54 - Métodos Numéricos

Trabajo Práctico 4: Ecuaciones Diferenciales Ordinarias

Grupo 8

Crespo, Elisabet del Pilar	59098
Martorell, Ariel	56209
Vekselman, Alan	59378

Profesores

Fierens, Pablo Ignacio

Grisales Campeón, Juan Pablo

Fecha de entrega: 13/05/2021

Índice

1. Introducción

En este trabajo se propuso implementar un algoritmo para resolver ecuaciones diferenciales ordinarias (EDOs) del siguiente estilo:

$$\frac{dx}{dt} = f\left(t, x\right), t > t_0 \tag{1}$$

Con las siguientes condiciones iniciales:

$$x(t_0) = x_0 \tag{2}$$

Luego se utilizará dicho algoritmo para resolver el modelo Hodkin-Huxley para potenciales de acción (v) en neuronas.

2. Runge-Kutta

El algoritmo a implementar para resolver EDOs está basado en Runge-Kutta de orden 4. Consiste en reemplazar las derivadas por evaluaciones de $f(t, x)$ en puntos adecuados para así poder aproximarse a la solución. Para el algoritmo de Runge-Kutta de orden 4 se utiliza la siguiente evaluación:

$$x_{k+1} = x_k + \frac{1}{6} \cdot h \cdot (k_1 + 2k_2 + 2k_3 + k_4) \tag{3}$$

Donde:

$$\begin{cases} k_1 = f\left(x_k, y_k\right) \\ k_2 = f\left(x_k + \frac{1}{2}h, y_k + \frac{1}{2}k_1 h\right) \\ k_3 = f\left(x_k + \frac{1}{2}h, y_k + \frac{1}{2}k_2 h\right) \\ k_4 = f\left(x_k + h, y_k + k_3 h\right) \end{cases} \tag{4}$$

2.1. Implementación

El algoritmo *ruku4* resuelve un sistema de ecuaciones diferenciales con parámetros iniciales aplicando el método de Runge-Kutta de orden 4.

Recibe la función a analizar *f*, el tiempo inicial *t_0*, el tiempo final *t_f*, el paso de integración *delta_t* y la condición inicial *x_0* como arreglo de *Numpy*.

Primero se define *N*, total de puntos a evaluar, como el cociente de la diferencia entre el tiempo inicial y el tiempo final sobre el paso de integración. Luego se crean la matriz *x_k* donde serán almacenados todos los valores de *x* incluido el valor inicial en su primer elemento, y el arreglo *t_k* donde se crea y almacena todo el intervalo de valores de t separados a una distancia igual al paso de integración.

Por último se procede a realizar un ciclo *for* en el que se evalúa para cada *N* anteriormente definido para la discretización en los k_1, k_2, k_3 y k_4, y se realiza la ecuación correspondiente para calcular el siguiente valor x_{k+1}.

Finalmente se devuelven los valores de t_k y las aproximaciones de x_k calculadas.

Cabe destacar que la función *f* debe recibir un valor *t* y un arreglo de *Numpy* *x* del mismo formato que el *x_0* recibido por *ruku4*. A su vez debe devolver otro arreglo de *Numpy* con el mismo formato.

3. Hodgkin-Huxley

El sistema correspondiente al modelo Hodkin-Huxley para potenciales de acción (v) en neuronas se describe de la siguiente forma:

$$\begin{cases} C\dot{v}(t) = i(t) - g_{Na}m^3h(v - v_{Na}) - g_K n^4(v - v_K) - g_L(v - v_L), t > 0 \\ \dot{n} = \alpha_n(v)(1-n) - \beta_n(v)n \\ \dot{m} = \alpha_m(v)(1-n) - \beta_m(v)m \\ \dot{h} = \alpha_h(v)(1-h) - \beta_h(v)h \end{cases} \tag{5}$$

En donde se utilizan los siguientes parámetros:

$$\begin{cases} \alpha_n(v) = 0,001\frac{v+55}{1-e^{\frac{v+55}{10}}} \quad \beta_n(v) = 0,125e^{-\frac{v+65}{80}} \\ \alpha_m(v) = 0,100\frac{v+40}{1-e^{\frac{v+40}{10}}} \quad \beta_m(v) = 4,000e^{-\frac{v+65}{18}} \\ \alpha_h(v) = 0,070^{-\frac{v+65}{30}} \quad \beta_h(v) = \frac{1}{1+e^{-\frac{v+35}{10}}} \\ i(t) = i_0, \quad t > 0 \end{cases} \tag{6}$$

y los siguientes valores:

$$\begin{cases} g_N a = 120, \quad g_K = +36 \quad g_L = 0,3, \\ v_N a = 50, \quad v_K = -77, \quad v_l = -54,4 \\ C = 1 \end{cases} \tag{7}$$

Teniendo en cuenta las siguientes condiciones iniciales:

$$v(0) = -65mV, \quad n(0) = m(0) = h(0) = 0 \tag{8}$$

Donde v tiene unidades de mV; n, m y h no tienen unidades; los parámetros g_i tienen unidades de $\frac{mS}{cm^3}$; i tiene unidad de $\frac{\mu A}{cm^3}$; y C tiene unidades de $\frac{\mu F}{cm^3}$.

3.1. Implementación

Como primera instancia se definen las contantes establecidas anteriormente y el valor inicial de la corriente de excitación *i_0* que va a definir el comportamiento del sistema. La función del algoritmo *hodgkinhuxley* recibe los handlers que representen cada uno de las ecuaciones de $\dot{v}$, $\dot{n}$, $\dot{m}$, $\dot{h}$, y t. Luego se realizan las evaluaciones correspondientes para cada parámetro y se devuelven *out_v*, *out_n*, *out_m* y *out_h* en un arreglo de *Numpy*.

4. Test

Para testear *ruku4* se procedió a utilizar las siguientes EDOs a las cuales les corresponden soluciones analíticas conocidas:

$$\frac{dy}{dt} = (y+1) \cdot sin(t) \tag{9}$$

$$\frac{dy}{dt} = 3 \cdot (y+t) \tag{10}$$

Cuyos parámetros iniciales e intervalo son aleatorios y con un paso de integración de forma tal que entren $N = 1000$ elementos en dicho intervalo. Luego se compara la solución obtenida con la analítica, teniendo en cuenta una tolerancia de $10-10$. Si se obtiene una comparación exitosa se imprime *RK Worked* y en caso contrario se imprime *RK Failed*.

Por otro lado, para evaluar el correcto funcionamiento de *hodgkinhuxley* se procedió a hallar los valores de i_0 para los cuales el modelo posee un comportamiento oscilatorio. Se utiliza *ruku4* para obtener la solución al sistema, con los valores iniciales y de i_0 establecidos previamente. Para poder analizar mejor los resultados obtenidos se decidió graficar su comportamiento, los cuales se observan a continuación.

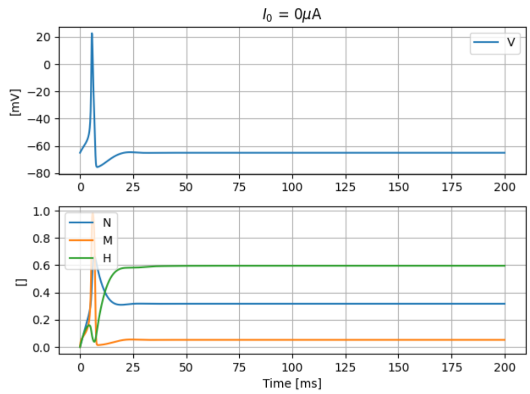

Figura 1: Algoritmo utilizando $i_0 = 0\mu A$

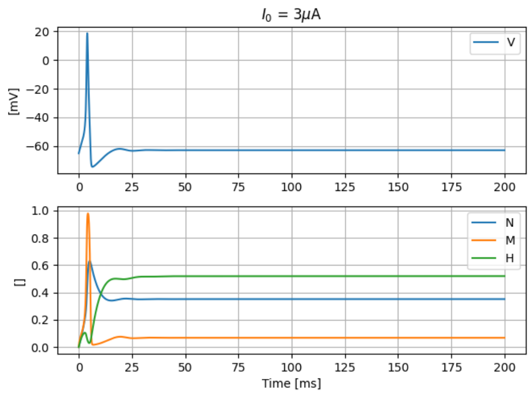

Figura 2: Algoritmo utilizando $i_0 = 3\mu A$

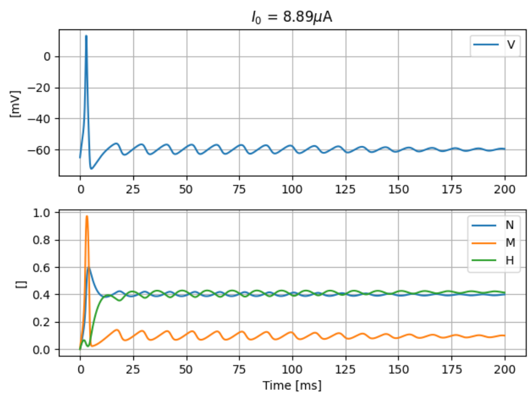

Figura 3: Algoritmo utilizando $i_0 = 8.89\mu A$

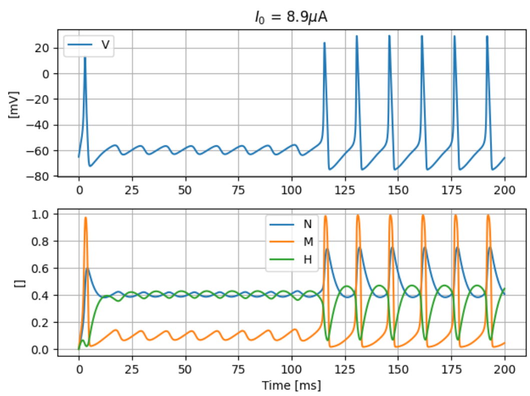

Figura 4: Algoritmo utilizando $i_0 = 8.9\mu A$

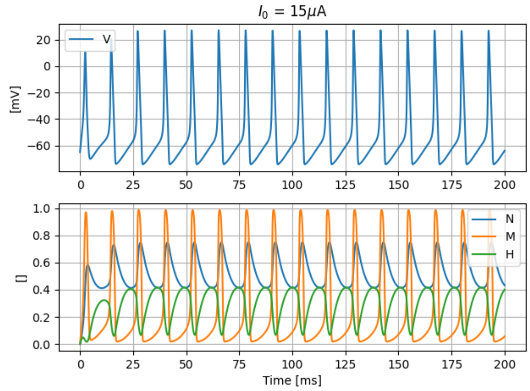

Figura 5: Algoritmo utilizando $i_0 = 15\mu A$

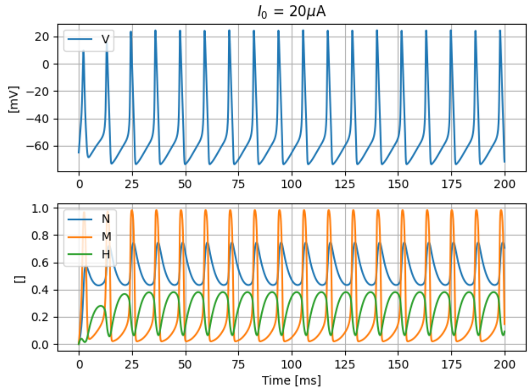

Figura 6: Algoritmo utilizando $i_0 = 20\mu A$

Como se puede observar, el régimen oscilatorio del sistema entra en efecto para un valor de i_0 de aproximadamente 8.9 $\frac{\mu A}{cm^3}$.

5. Anexo

5.1. Runge-Kutta 4

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

```
# RK4
####################################################################################
def ruku4(f, t_0, t_f, delta_t, x_0):

    N = int((t_f - t_0) / delta_t) + 1

    x_k = np.zeros((N, len(x_0)))
    x_k[0] = x_0
    t_k = t_0 + delta_t * np.arange(N)

    for i in range(N-1):
        k1 = f(t_k[i], x_k[i])
        k2 = f(t_k[i] + delta_t/2, x_k[i] + k1 * delta_t/2)
        k3 = f(t_k[i] + delta_t/2, x_k[i] + k2 * delta_t/2)
        k4 = f(t_k[i] + delta_t, x_k[i] + k3 * delta_t)

        x_k[i+1] = x_k[i] + (k1 + 2 * k2 + 2 * k3 + k4) * delta_t / 6

    return t_k, x_k
####################################################################################
```

5.2. Hodgkin-Huxley

```python
# CONSTANTS
####################################################################################
a_n = lambda v: .01 * (v+55)/(1 - np.exp(-(v+55)/10))
b_n = lambda v: .125 * np.exp(-(v+65)/80)
a_m = lambda v: .1 * (v + 40)/(1 - np.exp(-(v+40)/10))
b_m = lambda v: 4 * np.exp(-(v+65)/18)
a_h = lambda v: .07 * np.exp(-(v+65)/20)
b_h = lambda v: 1 / (1 + np.exp(-(v+35)/10))


g_Na, g_K, g_L = 120, 36, .3
v_Na, v_K, v_L = 50, -77, -54.4
C = 1
v_0 = -65
n_0 = m_0 = h_0 = 0
####################################################################################

# VARIABLES
####################################################################################
i_0 = 8.9
####################################################################################


# Hodgkin-Huxley
####################################################################################
def hodgkinhuxley(t, vars):
    v, n, m, h = vars
    out_v = (i_0 - g_Na * m**3 * h * (v - v_Na) - g_K * n**4 * (v - v_K) - g_L * (v - v_L)) / C
    out_n = a_n(v) * (1 - n) - n * b_n(v)
    out_m = a_m(v) * (1 - m) - m * b_m(v)
    out_h = a_h(v) * (1 - h) - h * b_h(v)

    return np.array([out_v, out_n, out_m, out_h])
####################################################################################
```

5.3. Test

```python
def real_f1(t_0, x_0):
    f = lambda t, x: (x + 1) * np.sin(t)
    K = (x_0 + 1) * np.exp(np.cos(t_0))
    real = lambda t: K * np.exp(-np.cos(t)) - 1

    return f, real

def real_f2(t_0, x_0):
    f = lambda x, t: 3 * (x + t)
    K = (x_0 + (3 * t_0 + 1) / 3) * np.exp(-3 * t_0)
    real = lambda t: K * np.exp(3 * t) - (3 * t + 1) / 3

    return f, real

def test():

    # Test RK4
    ##################################################

    tests = [real_f1, real_f2]

    x_0 = -10 + 20 * np.random.rand()
    t_0 = np.random.rand() * 2
    tf = t_0 + np.random.rand() * 10

    N = 1e3
    delta_t = (tf - t_0) / N

    f, real_f = tests[np.random.randint(0, len(tests))](t_0, x_0)

    t, x = ruku4(f, t_0, tf, delta_t, [x_0])
    real = real_f(t)

    if np.allclose(real, x.reshape(-1), atol = 1e-10): print('RK Worked')
    else: print('RK Failed')

    ##################################################

    # Plot de HH
    ##################################################

    i = lambda t: i_0
    x_0 = [v_0, n_0, m_0, h_0]

    t, res = ruku4(hodgkinhuxley, t_0 = 0, t_f = 200, delta_t = .1, x_0 = x_0)

    v, n, m, h = res.T

    fig, (ax1, ax2) = plt.subplots(2)
    ax1.plot(t, v, label = 'V')
    ax2.plot(t, n, label = 'N')
    ax2.plot(t, m, label = 'M')
    ax2.plot(t, h, label = 'H')
    ax1.legend(); ax2.legend()
    ax1.grid(); ax2.grid()
```

```
ax1.set_ylabel('[mV]')
ax2.set_ylabel('[]')
ax2.set_xlabel('Time [ms]')

ax1.set_title(fr'$I_0$ = {i_0}$\mu$A')

fig.tight_layout()
plt.show()
##################################################
```