from scipy.integrate import odeint

import time

import numpy as np

from mpl\_toolkits.mplot3d import Axes3D

import matplotlib.pyplot as plt

from scipy.optimize import newton

from scipy.optimize import fsolve

from scipy.integrate import ode

def func(w, t):

V, m, h, n = w

dm = lambda m, t, V: (0.1\*(-V-48)/(np.exp((-V-48)/15)-1))\*(1-m) - 0.12\*(V+8)\*m/(np.exp((V+8)/5)-1)

dh = lambda h, t, V: 0.17\*np.exp((-V-90)/20)\*(1-h) - h/(np.exp((-V-42)/10)+1)

dn = lambda n, t, V: (0.0001\*(-V-50)/(np.exp((-V-50)/10)-1))\*(1-n) - 0.002\*np.exp((-V-90)/80)\*n

dV = lambda V, t, m, h, n: -((G\_na1\*(m\*\*3)\*h + G\_na2)\*(V - 40)

+ (G\_k\*np.exp((-V-90)/50)+0.015\*np.exp((V+90)/60)+ G\_k\*n\*\*4)\*(V + 100)

+ G\_an\*(V+60))/12

return np.array([dV(V,t,m,h,n),dm(m,t,V),dh(h,t,V),dn(n,t,V)])

**# Perform Euler method**

def euler(func, w0, t):

**# Initialize arrays to store the solutions**

num\_points = len(t)

w = np.zeros((num\_points, len(w0)))

w[0] = w0

**# Perform RK4 integration**

for i in range(num\_points - 1):

h = t[i + 1] - t[i]

k1 = h \* func(w[i], t[i])

w[i + 1] = w[i] + k1

return w

**# Perform Explicit**

def explicit\_rk4(func, w0, t):

# Initialize arrays to store the solutions

num\_points = len(t)

w = np.zeros((num\_points, len(w0)))

w[0] = w0

for i in range(num\_points - 1):

h = t[i + 1] - t[i]

k1 = h \* func(w[i], t[i])

k2 = h \* func(w[i] + 0.5 \* k1, t[i] + 0.5 \* h)

k3 = h \* func(w[i] + 0.5 \* k2, t[i] + 0.5 \* h)

k4 = h \* func(w[i] + k3, t[i + 1])

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

return w

**# Perform Backward Euler**

def backward\_euler(func, w0, t):

# Initialize arrays to store the solutions

num\_points = len(t)

w = np.zeros((num\_points, len(w0)))

w[0] = w0

for i in range(num\_points - 1):

h = t[i + 1] - t[i]

equation = lambda x: x-h \* func(x, t[i + 1]) - w[i]

w[i + 1] = fsolve(equation, w[i])

return w

**# Perform Implicit**

def implicit\_rk4(func, y0, t):

n = len(t)

m = len(y0)

y = np.zeros((n, m))

y[0] = y0

for i in range(n - 1):

h = t[i + 1] - t[i]

k1 = func(y[i], t[i])

# Define a function for Newton's method to find the new state

def equation(y\_new):

k2 = func(y\_new, t[i] + h / 2)

k3 = func(y\_new, t[i] + h / 2)

k4 = func(y\_new, t[i] + h)

y\_next = y[i] + (k1 + 2 \* k2 + 2 \* k3 + k4) \* h / 6

return y[i + 1] - y\_new + y\_next

# Use Newton's method to find the new state

y\_new = newton(equation, y[i],maxiter=200)

y[i + 1] = y\_new

return y

G\_k = 1.2

G\_na1 = 400

G\_na2 = 0.14

G\_an = 0.075

initial = (-80,0.07,0.06,0.03)

t = np.linspace(0, 1000, 5000)

start\_time = time.time()

sol = euler(func,initial, t)

FE\_time = time.time() - start\_time

start\_time = time.time()

sol1 = explicit\_rk4(func, initial, t)

ERK4\_time = time.time() - start\_time

start\_time = time.time()

sol2 = odeint(func,initial, t)

LSODA\_time = time.time() - start\_time

start\_time = time.time()

sol3 = backward\_euler(func, initial, t)

BE\_time = time.time() - start\_time

start\_time = time.time()

sol4 = implicit\_rk4(func,initial, t)

IRK4\_time = time.time() - start\_time

ref = np.mean((sol,sol1,sol2,sol3,sol4),axis=0)

**#Action potential plot**

fig = plt.figure(figsize = (10,7))

# plt.plot(t, sol4[:, 0], label = 'V')

plt.plot(t, ref[:,0], label = 'V1')

plt.legend()

plt.xlabel('t (msec)', fontsize=15)

plt.ylabel('V (mV)', fontsize=15)

plt.axhline(0,color='black')

plt.axvline(0,color='black')

plt.title(f'Kinetics V (G\_k ={G\_k}, G\_na1 ={G\_na1}, G\_na2 ={G\_na2}, G\_an ={G\_an})')

plt.grid()

plt.show()

**#Conductance plot**

fig = plt.figure(figsize = (10,7))

plt.plot(t, sol1[:, 1], label = 'm')

plt.plot(t, sol1[:, 2], label = 'h')

plt.plot(t, sol1[:, 3], label = 'n')

plt.legend()

plt.xlabel('t (msec)', fontsize=15)

plt.ylabel('m/n/h', fontsize=15)

plt.axhline(0,color='black')

plt.title(f'Kinetics n, m, h (G\_k ={G\_k}, G\_na1 ={G\_na1}, G\_na2 ={G\_na2}, G\_an ={G\_an})')

plt.grid()

plt.show()

#3D view

fig = plt.figure(figsize = (7, 7))

ax = Axes3D(fig)

ax.plot(sol[:,3], sol[:,1], sol[:,2])

ax.set\_xlabel('n', fontsize=15)

ax.set\_ylabel('m', fontsize=15)

ax.set\_zlabel('h', fontsize=15)

plt.title('Dependence of channel activity parameters')

plt.show()

**# Error computation**

simulated\_data = [sol, sol1, sol2, sol3, sol4]

method\_names = ['FE', 'ERK4', 'LSODA', 'BE', 'IRK4']

rmse\_results = []

global\_rmse\_results = []

for method, data in zip(method\_names, simulated\_data):

rmse = np.sqrt(np.mean((data[:, 0] - ref[:, 0]) \*\* 2)) # For V

global\_rmse = np.sqrt(np.mean((data - ref) \*\* 2)) # For all state variables combined

rmse\_results.append(rmse)

global\_rmse\_results.append(global\_rmse)

time = [FE\_time, ERK4\_time, LSODA\_time, BE\_time, IRK4\_time]

# Create a table

print("{:<10} {:<20} {:<20} {:<20}".format("Method", "RMSE", "Global RMSE", "Computation Time"))

print("="\*80)

for method, rmse, global\_rmse, ti in zip(method\_names, rmse\_results, global\_rmse\_results, time):

print("{:<10} {:<20} {:<20} {:<20}".format(method, rmse, global\_rmse, ti))

**#sparsity matrix**

r = ode(func).set\_integrator('vode', method='bdf')

initial = np.array(initial)

r.set\_initial\_value(initial, t[0])

# Number of variables in the system

n = len(initial)

**# Initialize the Jacobian matrix with zeros**

jacobian\_matrix = np.zeros((n, n))

# Perturbation size for finite differences

epsilon = 1e-6

t = np.linspace(0,1000,5000)

**# Loop over time points and compute the Jacobian matrix**

for i in range(len(t)):

if not r.successful():

break

w = r.integrate(t[i])

for j in range(n):

w\_plus = w.copy()

w\_minus = w.copy()

w\_plus[j] += epsilon

w\_minus[j] -= epsilon

# Explicitly copy the state variable and apply perturbations

perturbed\_w\_plus = np.copy(w)

perturbed\_w\_minus = np.copy(w)

perturbed\_w\_plus[j] = w\_plus[j]

perturbed\_w\_minus[j] = w\_minus[j]

derivative = (func(perturbed\_w\_plus, t[i]) - func(perturbed\_w\_minus,t[i])) / (2 \* epsilon)

jacobian\_matrix[:, j] = derivative

**# Plot the sparsity pattern**

plt.figure(figsize=(8, 6))

plt.spy(jacobian\_matrix, markersize=2)

plt.title('Jacobian Sparsity Pattern')

plt.xlabel('Columns')

plt.ylabel('Rows')

plt.show()