Runge-Kutta-Chebyshev (RKC)Method

Alaa Essam's report

The future depends on smooth and stable jumps

(But with extra help from Chebyshev's math magic)

In **neural modeling**, sometimes your system is **stiff** — meaning the equations change very fast in some places and slowly in others.
RKC is like a smart method that can:

- Take **bigger steps** (compared to Euler)
- Be **stable for stiff systems** (like neurons with ion channels that respond *very quickly*)
- Still be **explicit** (unlike Backward Euler, which is implicit and needs solving puzzles at each step)

Tiny Math Version (Neural Model Style)

Instead of:

Step = Now + Change from Now Like in **Euler**…

Or Step = Now + Change from the Future Like in **Backward Euler**…

We say in **RKC**:

Step = Smooth jump with help from many tiny substeps,

each weighted with special Chebyshev constants

Or more precisely:

In RKC (2-stage version):

$$Y_o = Y_n$$

$$Y_1 = Y_o + \mu_1.\mathrm{h}\,\mathrm{f}(t_n, Y_o)$$

$$Y_2 = (1 - a_1)Y_o + a_1 Y_1 + \mu_2.\mathrm{h}\,\mathrm{f}(t_n + c_2.h, Y_1)$$

Where:

- Y_0 is the value at current time t_n
- Y_2 becomes the approximation for Y_{n+1}
- μ_1, μ_2, a_1, c_2 are constants based on Chebyshev polynomials (they control stability)

Intuition (No Math Mode):

You **don't guess the future** (like Backward Euler),
You **don't trust the now only** (like Euler),
You **build the next step with careful little nudges**, each nudge tested for stability.

Why It Works for Neurons?

- Ion channels = fast + slow dynamics = **stiff**
- Membrane potentials = sensitive to small input
- RKC = keeps solution **stable**, without needing tiny dt values like Euler

RKC is a great balance: faster than implicit methods, and more stable than Euler.

RKC Execution Time: 0.0030 seconds

➢ Code of Izhikevich model using Backword Eular:

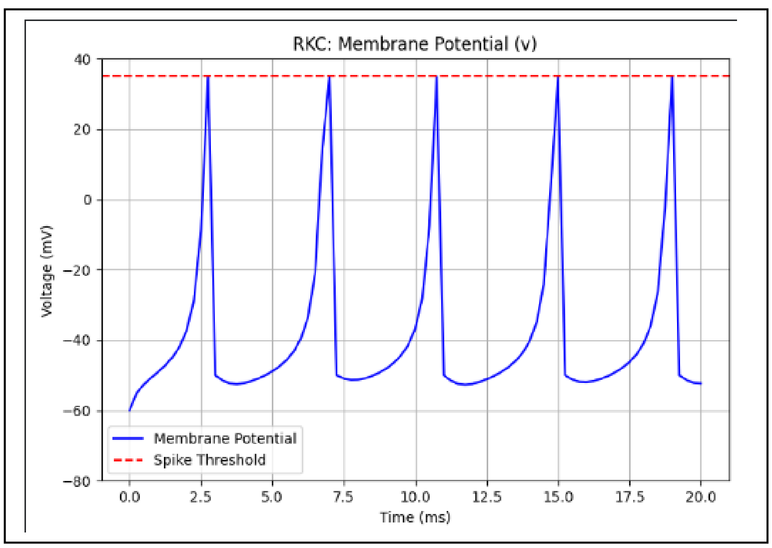
```
import numpy as np
import matplotlib.pyplot as plt
import time
# Neuron model parameters
C = 100  # Capacitance (pF)
vr = -60  # Resting potential (mV)
vt = -40  # Threshold potential (mV)
k = 0.7  # Gain parameter
a = 0.03  # Recovery time scale
b = -2  # Sensitivity of recovery to subthreshold fluctuations
c = -50  # After-spike reset value for v (mV)
d = 100  # After-spike increment for w
vpeak = 35  # Spike cutoff value (mV)
I = 100  # Input current (pA)
# Initial conditions
v0 = vr
w0 = 0
T = 20  # Total simulation time (ms)
dt = 0.25  # Time step (ms)
steps = int(T / dt)  # Number of steps

def neuron_ode(v, w, I):
    dv = (k * (v - vr) * (v - vt) - w + I) / C
    dw = a * (b * (v - vr) - w)
    return dv, dw

def rkc_solver():
    start_time = time.time()  # Start timing

    v = np.zeros(steps + 1)
    w = np.zeros(steps + 1)
    t = np.arange(0, T + dt, dt)
    v[0] = v0
    w[0] = w0
    iteration_data = []

    # RKC parameters
    s = 4  # Number of stages
    mu = 1.0  # Damping parameter
    tau = 1.0 / (mu ** 2)

    for i in range(steps):
        if v[i] >= vpeak:
            v[i] = vpeak
            v[i + 1] = c
            w[i + 1] = w[i] + d
            iteration_data.append({
                'step': i,
                'time': i * dt,
                'v_before': v[i],
                'w_before': w[i],
                'iterations': 0,
                'v_after': v[i + 1],
                'w_after': w[i + 1],
                'message': 'Spike detected - reset values'
            })
            continue

        current_v = v[i]
        current_w = w[i]
```

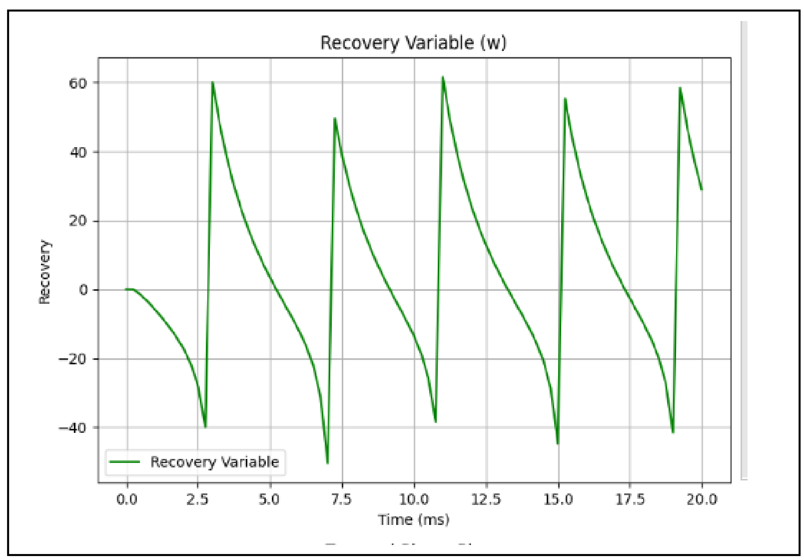
```
        # First stage
        dv, dw = neuron_ode(current_v, current_w, I)
        y0 = np.array([current_v, current_w])
        f0 = np.array([dv, dw])
        y1 = y0 + (dt / s) * f0

        # Chebyshev recurrence
        y_prev = y0
        y_curr = y1

        for j in range(2, s + 1):
            theta = np.pi / (2 * s)
            theta_j = (j - 1) * theta
            omega_0 = 1.0 + np.sin(theta) ** 2 / 3.0
            omega_j = 1.0 + np.sin(theta_j) ** 2 / 3.0

            if j == 2:
                beta = (2 * omega_j) / omega_0
            else:
                beta = (4 * omega_j) / omega_0

            alpha = beta / tau

            # Evaluate derivative at current stage
            fj = np.array(neuron_ode(y_curr[0], y_curr[1], I))

            # Update solution
            y_new = (1 - alpha) * y_prev + alpha * y_curr + beta * (dt / s) * fj

            y_prev = y_curr
            y_curr = y_new

        # Store results
        v[i + 1], w[i + 1] = y_curr

        iteration_data.append({
            'step': i,
            'time': i * dt,
            'v_before': current_v,
            'w_before': current_w,
            'iterations': s,
            'v_after': v[i + 1],
            'w_after': w[i + 1],
            'message': 'RKC update'
        })

        if v[i + 1] < -100:
            print(f"Warning: Abnormal voltage detected at t={i * dt} ms: {v[i + 1]} mV")

    end_time = time.time()  # End timing
    execution_time = end_time - start_time
    print(f"\nRKC Execution Time: {execution_time:.4f} seconds")

    return v, w, iteration_data


# Run the simulation
v_rkc, w_rkc, iteration_data = rkc_solver()

# Print iteration details
print("\nIteration Details:")
print(
```

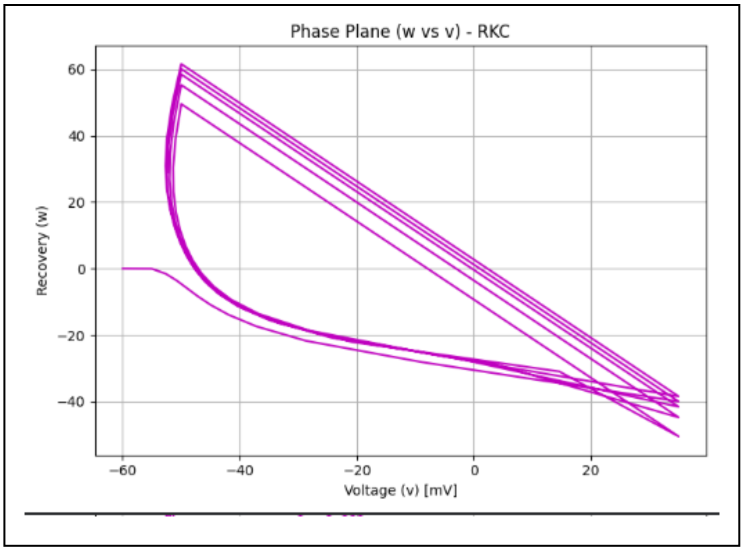
```
    f"{'Step':<6} {'Time':<8} {'Iterations':<10} {'v_before':<10} {'w_before':<10}
{'v_after':<10} {'w_after':<10} {'Message':<20}")
for data in iteration_data[:50]:  # Print first 50 steps
    print(
        f"{data['step']:<6} {data['time']:<8.2f} {data['iterations']:<10}
{data['v_before']:<10.4f} {data['w_before']:<10.4f} {data['v_after']:<10.4f}
{data['w_after']:<10.4f} {data['message']:<20}")

# Create time vector
time = np.arange(0, T + dt, dt)

# Plotting
plt.figure(figsize=(14, 10))

# Plot membrane potential
plt.subplot(2, 2, 1)
plt.plot(time, v_rkc, 'b', label='Membrane Potential')
plt.axhline(y=vpeak, color='r', linestyle='--', label='Spike Threshold')
plt.title('RKC: Membrane Potential (v)')
plt.ylabel('Voltage (mV)')
plt.xlabel('Time (ms)')
plt.legend()
plt.grid(True)
plt.ylim(-80, 40)

# Plot recovery variable
plt.subplot(2, 2, 2)
plt.plot(time, w_rkc, 'g', label='Recovery Variable')
plt.title('Recovery Variable (w)')
plt.xlabel('Time (ms)')
plt.ylabel('Recovery')
plt.legend()
plt.grid(True)

# Phase plane plot
plt.subplot(2, 2, 3)
plt.plot(v_rkc, w_rkc, 'm')
plt.title('Phase Plane (w vs v) - RKC')
plt.xlabel('Voltage (v) [mV]')
plt.ylabel('Recovery (w)')
plt.grid(True)

# Zoomed phase plane
plt.subplot(2, 2, 4)
plt.plot(v_rkc, w_rkc, 'm')
plt.title('Zoomed Phase Plane')
plt.xlabel('Voltage (v) [mV]')
plt.ylabel('Recovery (w)')
plt.xlim(-70, 40)
plt.ylim(min(w_rkc) - 10, max(w_rkc) + 10)
plt.grid(True)

plt.tight_layout()
plt.show()

# Results summary
print("\nSummary of Results:")
print(f"Maximum voltage: {max(v_rkc):.2f} mV")
print(f"Minimum voltage: {min(v_rkc):.2f} mV")
print(f"Number of spikes: {len([i for i in range(len(v_rkc)) if v_rkc[i] == vpeak])}")
```

➢ Code Output (For first 15 iteration):

```
Summary of Results:
Maximum voltage: 35.00 mV
Minimum voltage: -60.00 mV
Number of spikes: 5


Iteration Details:
Step    Time      Iterations v_before   w_before   v_after    w_after    Message
0       0.00      4          -60.0000   0.0000     -55.0262   -0.0500    RKC update
1       0.25      4          -55.0262   -0.0500    -52.6228   -1.5769    RKC update
2       0.50      4          -52.6228   -1.5769    -50.7789   -3.5963    RKC update
3       0.75      4          -50.7789   -3.5963    -49.0472   -5.8679    RKC update
4       1.00      4          -49.0472   -5.8679    -47.1816   -8.3217    RKC update
5       1.25      4          -47.1816   -8.3217    -44.9297   -10.9732   RKC update
6       1.50      4          -44.9297   -10.9732   -41.8885   -13.9137   RKC update
7       1.75      4          -41.8885   -13.9137   -37.2166   -17.3471   RKC update
8       2.00      4          -37.2166   -17.3471   -28.7266   -21.7146   RKC update
9       2.25      4          -28.7266   -21.7146   -8.9824    -28.1036   RKC update
10      2.50      4          -8.9824    -28.1036   60.6497    -39.9713   RKC update
11      2.75      0          35.0000    -39.9713   -50.0000   60.0287    Spike detected - reset values
12      3.00      4          -50.0000   60.0287    -51.5071   47.8909    RKC update
13      3.25      4          -51.5071   47.8909    -52.3171   38.0471    RKC update
14      3.50      4          -52.3171   38.0471    -52.5207   29.9381    RKC update
15      3.75      4          -52.5207   29.9381    -52.2813   23.1180    RKC update
```







➢ **Why would we use this method?**

1- Stiff Problems Need Stability

2- RKC Allows Larger Time Steps

3- Easier to Implement than Implicit Methods

➢ **Why wouldn't we use this method?**

1- Not Efficient for Non-Stiff Problems

2- Limited to Certain Types of Stiffness

3- Requires Careful Tuning

4-Only for Explicit ODEs.

5- Not Ideal for Rapid, Nonlinear Changes in Neural Models

6- Can Be Inefficient for Long Simulations

7- Not Widely Supported in Common Libraries