Backward Euler Method

Alaa Essam's report

The future depends on itself (self-referential).

Better for what's called **Stiff ODEs** — which often appear in neural simulations we can simply say it like this:

Tiny Math Version (Neural Model Context):

Instead of:
Step = Now + Change from Now (Explicit method like Euler)
You do:
Step = Now + Change from the Next (Implicit method like Backward Euler)

But to do that, you need to **solve a little puzzle** (usually a nonlinear equation)to figure out where that "next" place (the next state of the neuron) is.

This is important in neural modeling because ion channel dynamics or membrane potentials can change very fast in short time intervals. So, stiff solvers help maintain **numerical stability** without requiring very tiny time steps.

$$\mathrm{Y}_{\mathrm{n+1}}\mathbf{+1} = \mathrm{Y}_{\mathrm{n}} + hf(\mathrm{t}_{\mathrm{n+1}}\,,\mathrm{Y}_{\mathrm{n+1}})$$

➢ Code of Izhikevich model using Backword Eular:

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

# information from the refrance
C = 100
vr = -60
vt = -40
k = 0.7
a = 0.03
b = -2
c = -50
d = 20
vpeak = 35
I = 100
#  initial condition
v0 = vr
w0 = 0
T = 1000
dt = 0.25  #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 backward_euler():
    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 = []

    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]
        # Forward guess
        dv, dw = neuron_ode(current_v, current_w, I)
        next_v = current_v + dt * dv
        next_w = current_w + dt * dw
        # Newton iteration
        tolerance = 1e-6
        max_iter = 100
        iteration_count = 0
```

```python
        for _ in range(max_iter):
            iteration_count += 1

            f_v, f_w = neuron_ode(next_v, next_w, I)
            F1 = next_v - current_v - dt * f_v
            F2 = next_w - current_w - dt * f_w
            # Jacobian matrix elements (CORRECTED HERE)
            J11 = 1 - dt * (k * (2 * next_v - vr - vt) / C)
            J12 = -dt * (-1 / C)
            J21 = -dt * (a * b)
            J22 = 1 - dt * (-a)
            # Solve linear system
            det = J11 * J22 - J12 * J21
            if det == 0:
                break

            delta_v = (F2 * J12 - F1 * J22) / det
            delta_w = (F1 * J21 - F2 * J11) / det

            # Update solution
            next_v += delta_v
            next_w += delta_w

            # Check convergence
            if abs(delta_v) < tolerance and abs(delta_w) < tolerance:
                break

        v[i + 1] = next_v
        w[i + 1] = next_w

        iteration_data.append({
            'step': i,
            'time': i * dt,
            'v_before': current_v,
            'w_before': current_w,
            'iterations': iteration_count,
            'v_after': next_v,
            'w_after': next_w,
            'message': 'Normal update'
        })

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

    return v, w, iteration_data


# Run the simulation
v_be, w_be, iteration_data = backward_euler()

print("Iteration 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 to avoid too much output
    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)
```

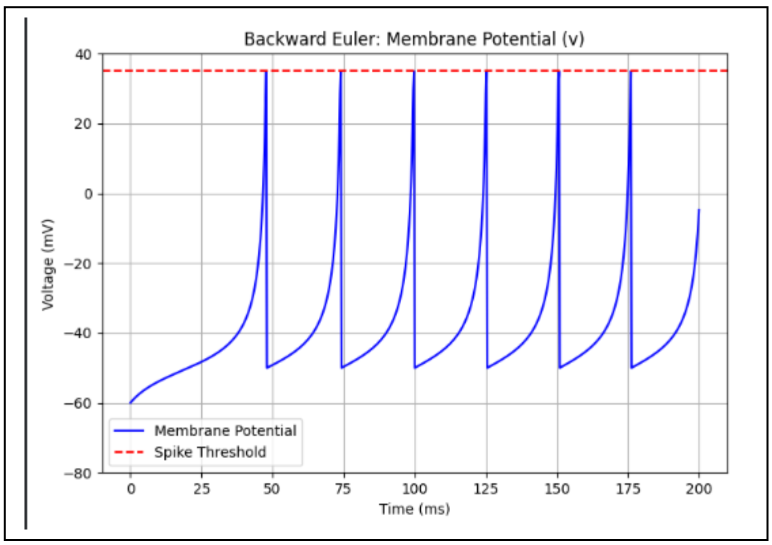
```python
plt.figure(figsize=(14, 10))

# Plot membrane potential
plt.subplot(2, 2, 1)
plt.plot(time, v_be, 'b', label='Membrane Potential')
plt.axhline(y=vpeak, color='r', linestyle='--', label='Spike Threshold')
plt.title('Backward Euler: Membrane Potential (v)')
plt.ylabel('Voltage (mV)')
plt.xlabel('Time (ms)')
plt.legend()
plt.grid(True)
plt.ylim(-80, 40)  # Adjust scale to cover -70 to +40

# Plot recovery variable
plt.subplot(2, 2, 2)
plt.plot(time, w_be, '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_be, w_be, 'm')
plt.title('Phase Plane (w vs v) - Backward Euler')
plt.xlabel('Voltage (v) [mV]')
plt.ylabel('Recovery (w)')
plt.grid(True)

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

plt.tight_layout()
plt.show()

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

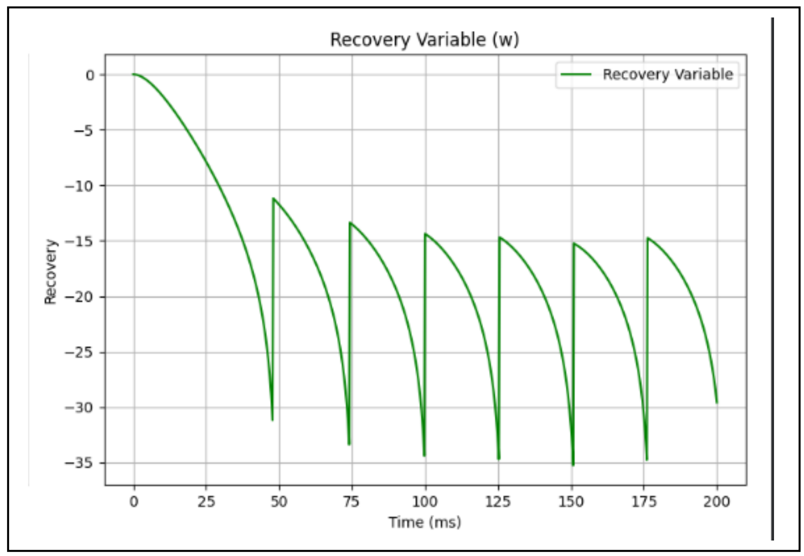
➢ Code Output (For first 10 iteration):

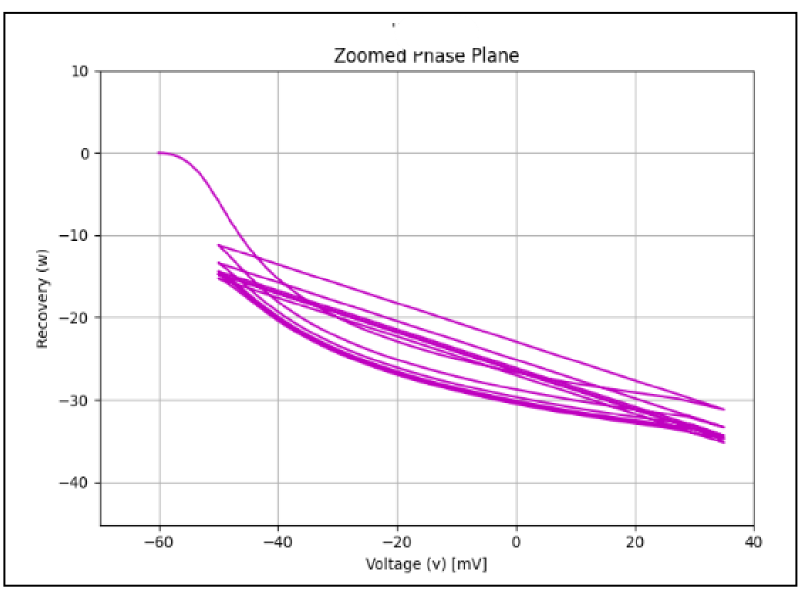
```
Summary of Results:
Maximum voltage: 35.00 mV
Minimum voltage: -60.00 mV
Number of spikes: 38
```

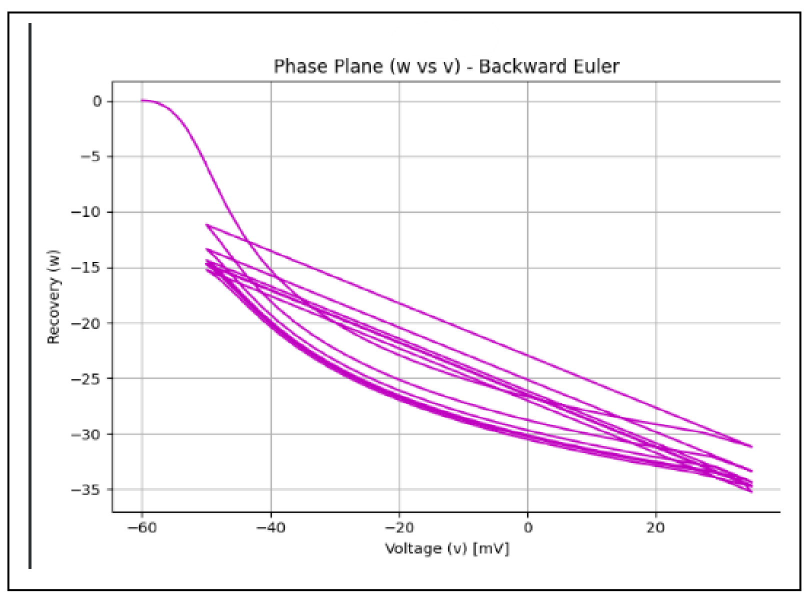
Iteration Details:

Step	Time	Iterations	v_before	w_before	v_after	w_after	Message
0	0.00	2	-60.0000	0.0000	-59.7583	-0.0036	Normal update
1	0.25	2	-59.7583	-0.0036	-59.5246	-0.0106	Normal update
2	0.50	2	-59.5246	-0.0106	-59.2982	-0.0210	Normal update
3	0.75	2	-59.2982	-0.0210	-59.0789	-0.0346	Normal update
4	1.00	2	-59.0789	-0.0346	-58.8662	-0.0512	Normal update
5	1.25	2	-58.8662	-0.0512	-58.6598	-0.0708	Normal update
6	1.50	2	-58.6598	-0.0708	-58.4593	-0.0932	Normal update
7	1.75	2	-58.4593	-0.0932	-58.2645	-0.1183	Normal update
8	2.00	2	-58.2645	-0.1183	-58.0750	-0.1461	Normal update
9	2.25	2	-58.0750	-0.1461	-57.8906	-0.1764	Normal update
10	2.50	2	-57.8906	-0.1764	-57.7110	-0.2092	Normal update









➢ Why would we use this method?

1- Stability for Stiff Equations

2-Better Long-Term Behavior

3- Works Well for Implicit Systems

4- Larger step sizes possible (more efficient for tough problems).

➢ Why wouldn't we use this method?

1- It's Implicit — You Have to Solve an Equation Each Step

2- Slower and More Computationally Expensive

3- Not Always Necessary If the system is not stiff, then Forward Euler or other explicit methods are faster and good enough.

4-Harder to Implement.