Adaptive Exponential Rosenbrock–Euler Method (ExpRESS–Euler)’s report

---

The ODE System

$dv/dt = k(v - vr)(v - vt) - w + In$

$dw/dt = a[b(v - vr) - w]$

Parameters

C : Membrane capacitance (value: $C = 100$, units: typically pF or scaled).

v : Neuron membrane potential (in millivolts, mV).

w : Recovery current (in picoamperes, pA , or scaled units).

t : Time (in milliseconds, ms).

vr : Resting membrane potential (value: $vr = -60mV$).

k : Scaling factor for the quadratic term (value: $k = 0.7$).

vt : Threshold membrane potential (value: $vt = -40mV$).

In : Input current (values: $In = 0, for, t < 101ms, In = 70, for, t \geq 101ms$, units: pA).

a : Time scale of recovery (value: $a = 0.03$, units: $ms-1$).

b : Sensitivity of recovery to membrane potential (value: $b = -2$, dimensionless or scaled).

Initial Conditions

$v(t = 0) = v0, w(t = 0) = w0$
v0 and w0 constants based on the state of the neuron.

Reset Mechanism

consider a membrane potential threshold vpeak = 35 mV, when the membrane potential reaches vpeak:

- we set $v(t) = -50$ mV
- set $w(t) = w(t) + 100$

Adaptive Exponential Rosenbrock–Euler Method

The Adaptive Exponential Rosenbrock-Euler method (ExpRESS-Euler) is a numerical technique for solving ordinary differential equations (ODEs), particularly those that are stiff, meaning they contain vastly different timescales. It combines the concepts of exponential integrators, Rosenbrock methods, and adaptive step-size control.

Method’s Formula

In standard ODE form:

$dy/dt = f(t, y)$

The **ExpRESS–Euler** updates the solution using:

$y_{n+1} = y_n + h\ \varphi_1(h * J_n) [f(t_n, y_n)]$

Where:

y_n : current solution vector (e.g. [v, w] in neuron models).

h is the Adaptive Step Size. It adjusts the time step size based on local error estimation or stiffness indicators, This keeps the method stable and efficient across varying dynamics.

$J_n \approx \partial f/\partial y$ = Jacobian (or an approximation).

$\varphi_1(z) = (e^z-1)/z$, a *matrix function* that comes from exponential integrators.

How to solve ODE using this Method

Step 1: Compute Current Values

At current time t_n, compute:

- y_n : your state vector
- $f_n=f(t_n, y_n)$

Step 2: Approximate or Compute the Jacobian

$J_n=\partial f/\partial y$ (at t_n and y_n).

- Can be exact (symbolically or numerically) or approximated via finite differences.

Step 3: Evaluate the Matrix Function φ_1

Now comes the core:

$\varphi 1(h * Jn)$

How do you do this?

- Most practical way: Use a matrix identity:
- $\varphi 1(hJ) \approx (I - hJ)^{-1}$, this is a *first-order* approximation using a resolvent (comes from the exponential series).

Step 4: Update the Solution

Use the main formula:

$yn+1 = yn + h \ \varphi 1(h * Jn) [f(tn, yn)]$

or you can use substitute $\varphi 1(h * Jn)$ by the resolvent in step 3.

Step 5: Estimate Error (For Adaptivity)

To control the step size adaptively:

1. Compute the next point using one full step:
2. $y1 = yn + h \cdot \varphi 1(h J) \cdot f$
3. Then compute it again using **two half steps**:
   - Step 1: $yn+1/2$
   - Step 2: $yn+1$

Then:

$error = ||y1 - y2||$

where y1 is the y calculated using 1 step size and y2 is the y calculated using 2 step sizes

Step 6: Adjust Step Size h

If error<tolerance, accept the step. Otherwise, reduce h and retry.

Adapt h using:

$hnew = h \cdot (tol/error)^{(0.5)}$

Clamp it between hmin and hmax to avoid insanity.

Step 7: Repeat Until Final Time.

The Python Code

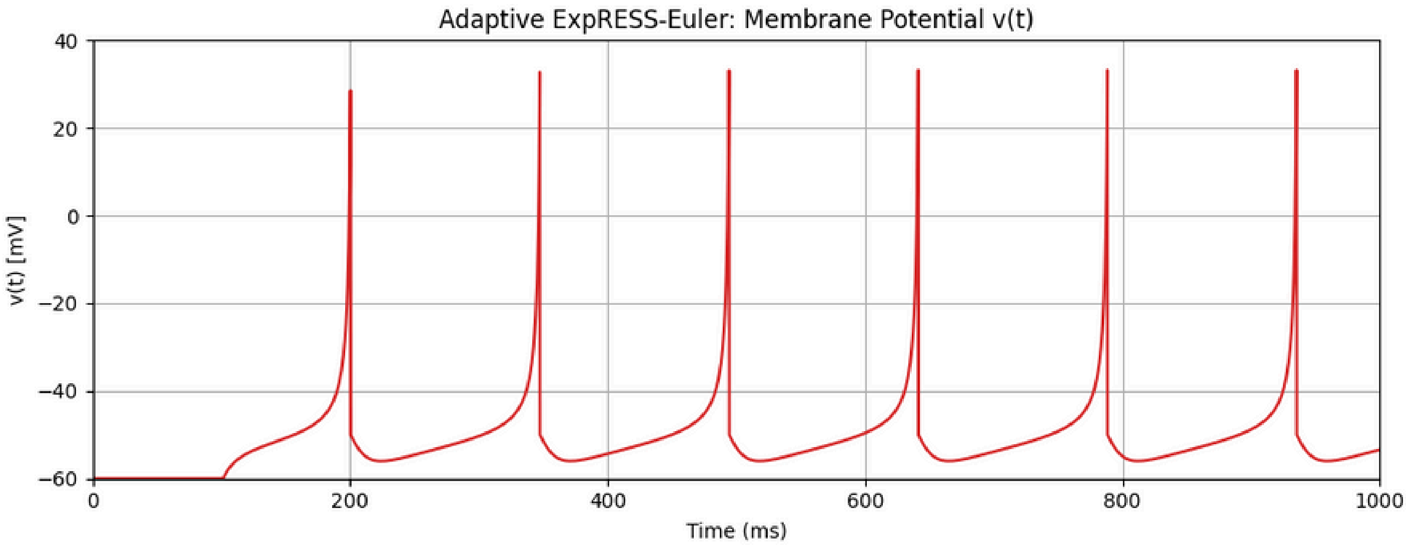
```
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import inv
import pandas as pd
import time

start_time = time.time()
# Constants
C = 100
k = 0.7
vr = -60
vt = -40
a = 0.03
b = -2
v_peak = 35
c = -50
d = 100

# Simulation setup
T = 1000  # total time (ms)
h = 0.25  # initial step size
tol = 0.5  # tolerance for adaptive step
fac_min, fac_max = 0.1, 5
h_min, h_max = 0.01, 2.0

# Initial conditions
t_vals = [0]
v_vals = [vr]
w_vals = [0]
h_vals = [h]

# Table data
recorded_data = []

t = 0
v = vr
w = 0

```

```python
while t < T:
    In = 0 if t < 101 else 70

    # Current state
    y = np.array([v, w])

    # Compute f(y)
    dvdt = (1 / C) * (k * (v - vr) * (v - vt) - w + In)
    dwdt = a * (b * (v - vr) - w)
    f = np.array([dvdt, dwdt])

    # Compute Jacobian
    dv_dv = (1 / C) * k * ((v - vr) + (v - vt))
    dv_dw = -1 / C
    dw_dv = a * b
    dw_dw = -a
    J = np.array([
        [dv_dv, dv_dw],
        [dw_dv, dw_dw]
    ])

    # First-order ExpRESS-Euler step
    I = np.eye(2)
    phi1 = inv(I - h * J)
    y1 = y + h * phi1 @ f

    # Half step + half step for error estimate (2nd-order Richardson)
    h_half = h / 2
    phi_half = inv(I - h_half * J)
    y_half = y + h_half * phi_half @ f

    v_half, w_half = y_half
    dvdt_half = (1 / C) * (k * (v_half - vr) * (v_half - vt) - w_half + In)
    dwdt_half = a * (b * (v_half - vr) - w_half)
    f_half = np.array([dvdt_half, dwdt_half])
    y2 = y_half + h_half * phi_half @ f_half

    # Error estimate
    err = np.linalg.norm(y1 - y2)
    err = max(err, 1e-10)  # Avoid divide-by-zero
```

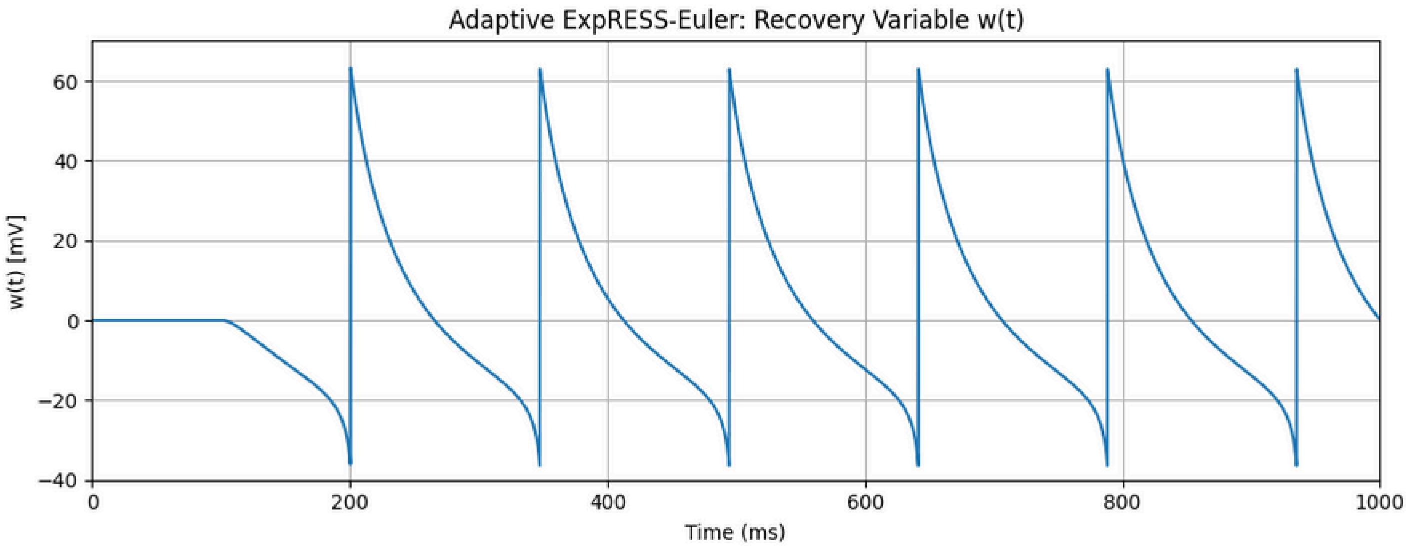
```python

    # Adaptive time step control
    if err < tol:
        t += h
        v, w = y2  # accept higher-order result
        if v >= v_peak:
            v = c
            w += d

        t_vals.append(t)
        v_vals.append(v)
        w_vals.append(w)
        h_vals.append(h)
        recorded_data.append([t, v, w, h])

        # Adjust h for next step
        h = h * min(max((tol / err) ** 0.5, fac_min), fac_max)
        h = min(max(h, h_min), h_max)
    else:
        # Reduce h and retry
        h = h * max((tol / err) ** 0.5, fac_min)
        h = max(h, h_min)

end_time = time.time()
# Print execution time
print(f"Execution time: {end_time - start_time:.8f} seconds")
```

Code Output

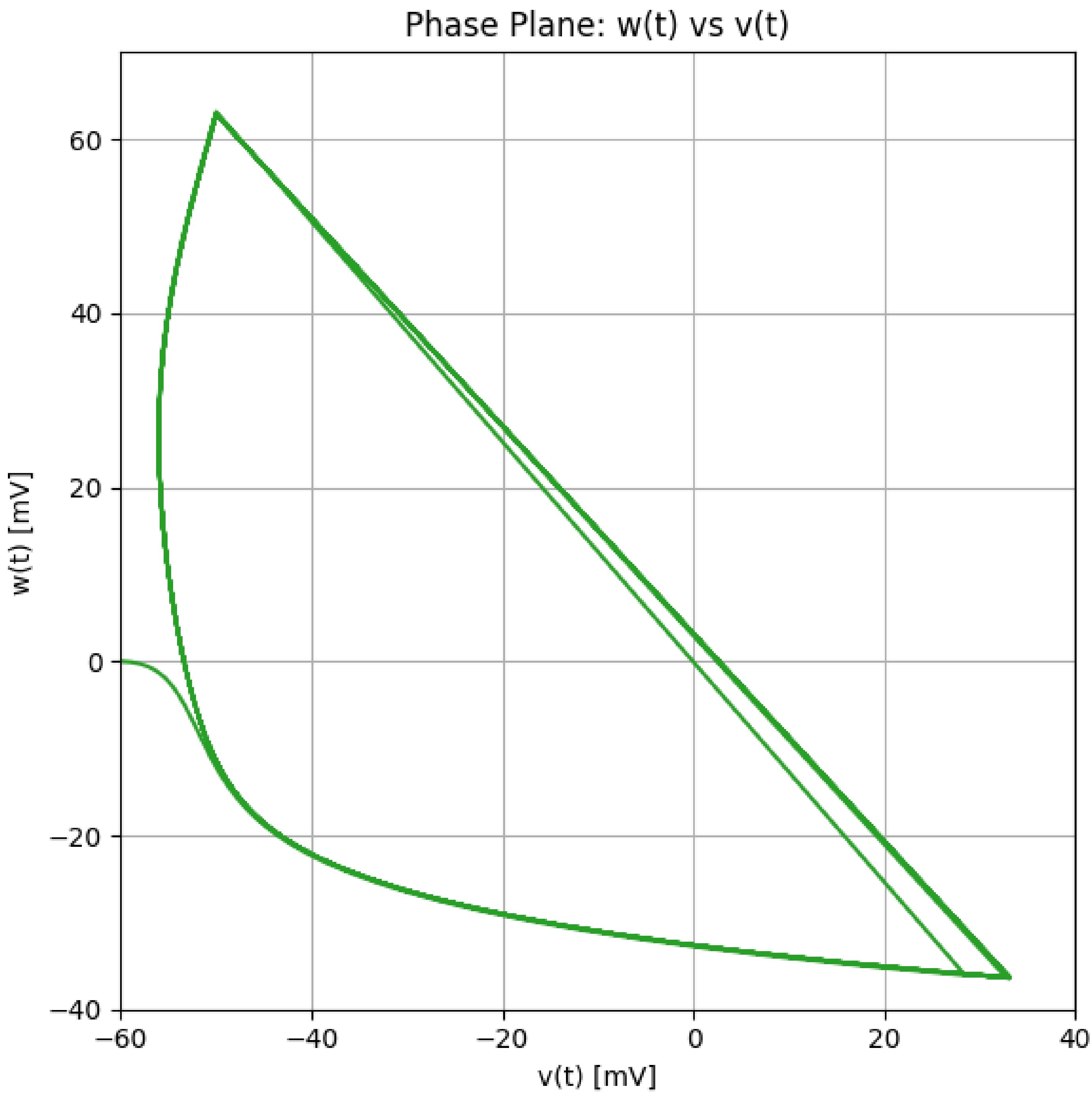
V vs. t


Adaptive ExpRESS-Euler: Membrane Potential v(t)
v(t) [mV]
Time (ms)

w vs. t


Adaptive ExpRESS-Euler: Recovery Variable w(t)
w(t) [mV]
Time (ms)

w vs. V


Phase Plane: w(t) vs v(t)
w(t) [mV]
v(t) [mV]
60
40
20
0
−20
−40
−60
−40
−20
0
20
40

```
    Time (ms)      v (mV)          w  Step size h
50       99.5 -60.000000  0.000000            2.0
51      101.5 -60.000000  0.000000            2.0
52      103.5 -58.843849 -0.102094            2.0
53      105.5 -57.917374 -0.310589            2.0
54      107.5 -57.156761 -0.598737            2.0
55      109.5 -56.519033 -0.946768            2.0
56      111.5 -55.974270 -1.339804            2.0
57      113.5 -55.501063 -1.766494            2.0
58      115.5 -55.083710 -2.218085            2.0
59      117.5 -54.710461 -2.687777            2.0
60      119.5 -54.372350 -3.170260            2.0
```

The error calculated relative to Explicit Euler Method

Time (ms)	v_sim	v_ref	v_err	**v_err %**	w_sim	w_ref	w_err	**w_err %**
0.0	-60.0000	-60.0000	0.0000	0.00%	0.000	0.0000	0.0000	—
250.0	-55.0123	-54.4819	0.5304	**0.97%**	5.950	6.2834	0.3334	**5.31%**
500.0	-51.0000	-50.6154	0.3846	**0.76%**	60.000	59.0910	0.9090	**1.54%**
750.0	-50.0000	-49.5530	0.4470	**0.90%**	-13.000	-12.4763	0.5237	**4.20%**
1000.0	-54.0000	-53.6973	0.3027	**0.56%**	2.000	1.5649	0.4351	**27.80%**

Pros and Cons of using this Method

Pros

- Handles Stiff Systems Well.
- Adaptive Time Stepping
- Stable at Larger Time Steps

Cons

- Requires Jacobian Computation, for high-dimensional systems, this becomes computationally expensive.
- Only First-Order Accurate
- Harder to Implement than Euler or RK, relatable tbh(T_T).