Simulating Oja's Learning Rule in Python

I. Introduction to Computational Neuron Models

When one thinks about learning, one of the most fundamental components tends to be the individual neuron. Therefore, when creating computational models of learning, it is important to have a mathematically representative neuron. The neuron model considered for the purposes of this paper is:

$$\sum_{i=1}^{n} w_i \vec{x_i} = \eta \tag{1}$$

Here, the neuron receives a set of scalar-valued inputs x_i. These inputs go through synaptic junctions with varying levels of coupling strength (or weights), represented by w_i. The unit then sends outputs in the form of η.

II. Creating a Differential Equation

A. Input Vectors

First, consider the term x_i. In this instance the equation is:

$$x(t) = \cos(2\pi f t + \theta) \tag{2}$$

where $f = 1\,\text{kHz}$ and phase θ was chosen. A sinusoidal $x(t)$ is most appropriate since it is controlled, which is important for learning. To control the spectrum, only f must be changed, or equivalently $\omega = 2\pi f$. This makes numerical choices clean.

B. Output

Now, consider the term $y(t)$:

$$y(t) = W^T \vec{x}(t) \tag{3}$$

Here y is the projection of the input onto w. This is important since the projection serves as the signal power along a direction. Also, since $x(t)$ is in a 2-D subspace, $y(t)$ remains a single sinusoid.

C. Weights

Finally, the differential equation to be solved for this project is:

$$\tau w'(t) = y(t)\vec{x}(t) - y^2(t)\vec{w} \tag{4}$$

This stems from Oja's Learning Rule, which adapts the initial computational neuron model into a function of learning. The term $y^2(t)\vec{w}$ represents learning, preventing unbounded growth.

III. Methods and Results

A key component of this project is choosing a solver. For this work, RK45, RK23, and Radau were tested using Python with SciPy.

A. Runge-Kutta 45

Runge-Kutta 4(5) uses 4th-order RK computation and 5th-order error estimation. The graphs show no oscillations. The $w(t)$ values quickly settle to a constant with a fast transient at the start. The $y(t)$ plot is flat, showing the solver has not captured periodic behavior. RK45 is explicit, calculating the solution at the next step using only current information. Stiff problems therefore require very small time steps.

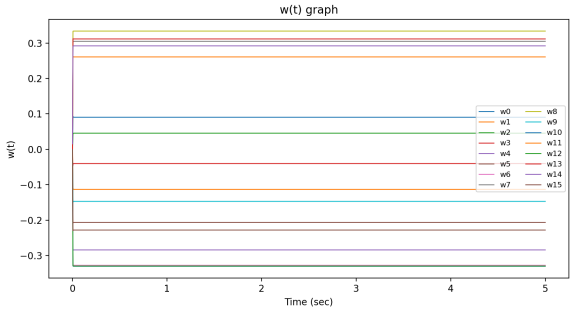

Fig. 1. RK45 solution of $w(t)$

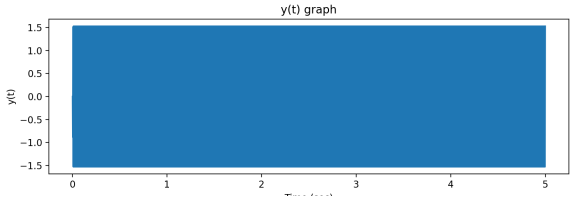

Fig. 2. RK45 solution of $y(t)$

B. Runge-Kutta 23

RK23 is a lower-order method (2nd-order computation, 3rd-order estimation). For the same tolerances, it takes smaller steps. The $w(t)$ solution is highly variable, while $y(t)$ is sinusoidal with gradually changing amplitude and phase.

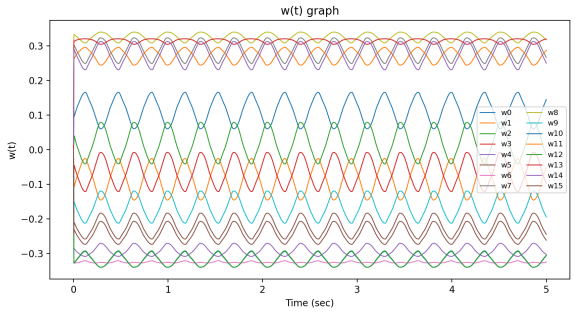

Fig. 3. RK23 solution of $w(t)$

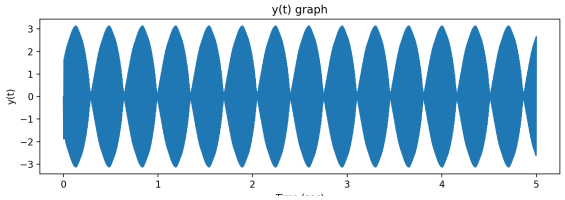

Fig. 4. RK23 solution of $y(t)$

C. Radau

Radau is an implicit Runge-Kutta method. Its figures show significant oscillation. Implicit methods create a stability domain covering the left half-plane. Belonging to Radau IIA methods, they are stiffly stable and can take large steps. The $w(t)$ solution is smoother than RK23 due to numerical damping; $y(t)$ is sinusoidal with negligible phase shift at chosen tolerances.

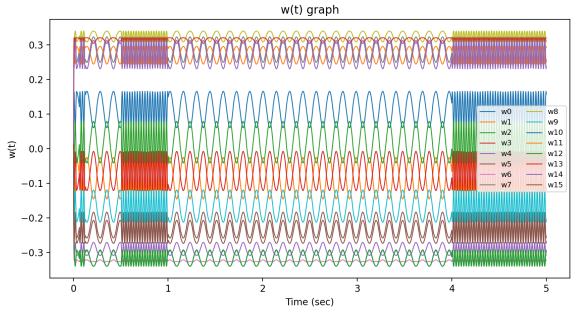

Fig. 5. Radau solution of $w(t)$

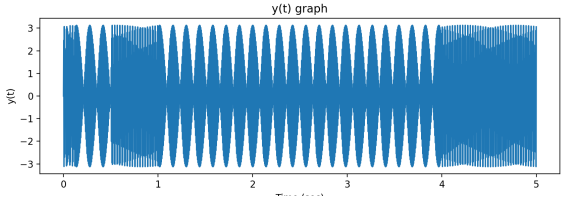

Fig. 6. Radau solution of $y(t)$

IV. Conclusion

In this work, Oja's learning rule was simulated in Python to evaluate solver performance in modeling a simple computational neuron. RK45 produced stable but overly damped solutions, RK23 provided dynamic but variable trajectories, and Radau handled stiffness robustly with minimal distortion. Solver choice strongly influences captured system dynamics. Future work could extend this analysis to higher-dimensional inputs or alternative learning rules.

References

[1] J. Sundnes, *Solving Ordinary Differential Equations in Python*, 2023.
[2] E. Oja, "A simplified neuron model as a principal component analyzer," *Journal of Mathematical Biology*, 1982.