Discovering Chaotic Dynamics: An AI-Assisted Sparse Identification of the Lorenz System using Machine Learning

Troy Chin

October 23, 2025

Abstract

The Lorenz attractor is one of the most extensively studied nonlinear dynamical systems in applied mathematics. The *Lorenz equations* were first discovered by M.I.T. mathematician and meteorologist Edward Lorenz, who was interested in fluid flow and atmospheric convection. Through data-driven methods, we analyze and model the chaotic dynamics of a particle moving along the path of the Lorenz attractor using the SINDy (Sparse Identification of Nonlinear Dynamics) algorithm.

1 Introduction

The *Lorenz equations* are a system of three coupled, nonlinear, first-order ordinary differential equations given by:

$$\dot{x} = \sigma(y - x) \tag{1}$$
$$\dot{y} = \rho(x - z) - y \tag{2}$$
$$\dot{z} = xy - \beta z \tag{3}$$

Where σ is the *Prandtl number*, ρ is the *Rayleigh number*, and β is a dimensionless constant related to the fluid layer itself. These equations were originally derived from a simplified model of fluid convection, describing how the velocity and temperature of a moving fluid evolve over time. Moreover, the Lorenz system is known to be a *dissipative system*, meaning that the energy in the system is non-conservative. In physical terms, it is **non-Hamiltonian**, which implies that the volume of the system in its *phase space* will contract under flows of energy. For canonical parameter values $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$, the system's long-term behavior forms the famous "butterfly-shaped" attractor.

2 Stability Analysis

Suppose we define a particle of mass m with a velocity magnitude of v that moves along the path of the Lorenz attractor. One can easily deduce that the particle's equations of motion are:

$$\dot{q}(t) = v(t) \tag{4}$$
$$\ddot{q}(t) = a(t) \tag{5}$$

Where $\dot{q}(t) = \begin{pmatrix} f_1(x,y,z) \\ f_2(x,y,z) \\ f_3(x,y,z) \end{pmatrix} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix}$, $\dot{q} \in \mathbb{R}^3$.

We can easily find the system's equilibrium points by setting each equation equal to zero. We construct the following equality $\dot{x} = \dot{y} = \dot{z} = 0$:

$$\begin{aligned} \sigma(y-z) &= 0 \\ \rho(x-z) - y &= 0 \\ xy - \beta z &= 0 \end{aligned}$$

Our first step is to solve for x from the first equation. After some basic algebraic manipulation, we come to the solution that $y = x$. Then, substituting into the third equation, we get:

$$x^2 = \beta z \implies z = \frac{x^2}{\beta}$$

$$\rho x - x - xz = 0 \implies x(\rho - 1 - z) = 0$$

Hence:

$$x = y = z = 0$$

Let $z = \rho - 1$. Then:

$$x^2 = \beta(\rho - 1) \implies x = y = \pm\sqrt{\beta(\rho - 1)}$$

Hence, the Lorenz system's equilibrium points are:

$$(x^*, y^*, z^*) = (\sqrt{\pm\beta(\rho-1)}, \pm\sqrt{\beta(\rho-1)}, \rho - 1)$$

These equilibrium points yield steady convection. Now, to analyze local stability, we compute the Jacobian matrix of the system:

$$J(x,y,z) = \frac{\partial f_i}{\partial q_j} = \begin{bmatrix} -\sigma & \sigma & 0 \\ \rho - z & -1 & -x \\ y & x & -\beta \end{bmatrix}. \tag{6}$$

Where:

At the origin P_0, the Jacobian becomes:

$$J_0 = \begin{bmatrix} -\sigma & \sigma & 0 \\ \rho & -1 & 0 \\ 0 & 0 & -\beta \end{bmatrix}.$$

The eigenvalues of J_0 determine stability:

- For $\rho < 1$, all eigenvalues have negative real parts, and P_0 is stable.
- For $\rho > 1$, one eigenvalue becomes positive, and P_0 becomes unstable, leading to bifurcation and the emergence of $P_\pm$.

The eigenvalues of the Jacobian then satisfy:

$$(\lambda + \beta)\big[(\lambda + \sigma)(\lambda + 1) - \sigma\rho\big] = 0.$$

Thus, one eigenvalue is $\lambda_3 = -\beta$, and the remaining two satisfy:

$$\lambda^2 + (\sigma + 1)\lambda + \sigma(1 - \rho) = 0.$$

For the origin to be stable, all real parts of eigenvalues must be negative. This occurs when $\rho < 1$. At $\rho = 1$, a **pitchfork bifurcation** occurs: the origin loses stability and the two symmetric fixed points $P_\pm$ emerge.

3 Deriving the Equations of Motion

We can now construct the equations of motion to construct the equations of motion for the particle.

$$\dot{q} = f(q) = v(t) \tag{7}$$

$$\ddot{q} = J_f(q)\dot{q} = \frac{\partial f_i}{\partial \dot{q}_j}\dot{q} = a(t) \tag{8}$$

Our first step is to find its momentum. Recall that an object's momentum is $\vec{p} = m\Delta v$. Then the particle's momentum can be expressed in terms of its mass times its velocity:

$$\vec{p} = mv(t) = m\dot{q}(t) \tag{9}$$

Recall from Newton's Second Law:

$$\vec{F_{net}} = \sum F_i = m\vec{a} = m\ddot{q(t)} \tag{10}$$

We can express the equations of motion of a particle moving along the Lorenz attractor in terms of its Jacobian multiplied by its velocity:

$$\vec{F_{net}} = m\frac{\Delta \vec{p(t)}}{\Delta t} = m\ddot{q(t)} = J_f(q(t))\dot{q}(t) \tag{11}$$

Where: $\ddot{q}(t) = a(q(t), v(t))$.

In three dimensions, we can represent the net force on the particle to be:

$$F_{net} = -\nabla V \tag{12}$$

However, for a non-Hamiltonian system like the Lorenz system, this is not the case. Our next step is to show that the Lorenz attractor is a disspative system, and that all the forces acting on the particle are non-conservative. To see this, we must ask the question: how do volumes contract under flows of energy?

If we pick an arbitrary closed surface $S(t)$ of volume $V(t)$, then S will evolve into a new surface $S(t+dt)$. We can also deduce the same conclusion for V, as the volume will evolve to $V(t+dt)$, for some dt. Now, let $\mathbf{f}$ be the instantaneous velocity acting on the surface $\hat{n}$ as the normal force acting on the surface. Then:

$$V(t+dt) = V(t) + \int_s (\mathbf{f}\cdot\hat{n})dA \implies \dot{V} = \frac{V(t+dt)-V(t)}{dt} = \int_s (\mathbf{f}\cdot\hat{n})dA \tag{13}$$

Finally, by the Stokes' Theorem, we can conclude that the change in volume of the Lorenz system is:

$$\dot{V} = \oint_V (\nabla\cdot\mathbf{F})dV \tag{14}$$

Computing the divergence of the Lorenz vector field:

$$\nabla\cdot\mathbf{F} = \frac{\partial\dot{x}}{\partial x} + \frac{\partial\dot{y}}{\partial y} + \frac{\partial\dot{z}}{\partial z} = (-\sigma) + (-1) + (-\beta) = -(\sigma+1+\beta).$$

This quantity is always negative for positive parameter values, implying:

$$V(t) = V(0)e^{-(\sigma+\beta+1)t}.$$

Thus, all trajectories contract exponentially to a strange attractor of zero volume in the long-time limit. Energy is not conserved, and the Lorenz system is intrinsically non-Hamiltonian.

4 Machine Learning Identification via SINDy

The Sparse Identification of Nonlinear Dynamics (SINDy) algorithm provides a data-driven framework to rediscover governing equations from time series data. Given a set of observed states $\mathbf{X}(t)$ and their derivatives $\dot{\mathbf{X}}(t)$, SINDy assumes:

$$\dot{\mathbf{X}}(t) = \Theta(\mathbf{X}(t))\Xi,$$

where $\Theta(\mathbf{X}) = [1, x, y, z, xy, xz, yz, x^2, y^2, z^2, \dots]$ is a library of candidate nonlinear functions, and Ξ is a sparse coefficient matrix.

By performing sparse regression (e.g., sequential thresholding or LASSO), SINDy identifies only the few active terms necessary to describe the dynamics. When applied to Lorenz trajectory data, SINDy correctly identifies:

$$\dot{x} = 10(y - x) \tag{15}$$
$$\dot{y} = 28x - xz - y \tag{16}$$
$$\dot{z} = xy - \frac{8}{3}z \tag{17}$$

matching the true governing equations to within numerical precision.

5 Numerical Implementation

Using a numerical integrator such as Runge–Kutta 4 (RK4) or `odeint`, the Lorenz system can be simulated with an initial condition $\mathbf{X}_0 = (1, 1, 1)$ over a time interval $t \in [0, 50]$. The resulting trajectory demonstrates sensitive dependence on initial conditions, a hallmark of chaos.

By collecting the simulated data and applying SINDy, the sparse regression coefficients Ξ reconstruct the Lorenz equations with minimal error. This demonstrates that data-driven discovery can extract interpretable models even from chaotic dynamics. We modeled the Lorenz equations and implemented the SINDy algorithm by writing a robust Python script which is given below:

```
    import tkinter as tk
from tkinter import ttk
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import numpy as np
from scipy.integrate import odeint
import pysindy as ps
import threading

# --- Lorenz system ---
def lorenz(state, t, s=10, r=28, b=2.667):
    x, y, z = state
    x_dot = s * (y - x)
    y_dot = r * x - y - x * z
    z_dot = x * y - b * z
    return [x_dot, y_dot, z_dot]


class LorenzApp:
```

```
def __init__(self, master):
    self.master = master
    master.title("Lorenz Attractor Simulator")
    master.geometry("1000x700")

    # --- Parameters ---
    self.s = tk.DoubleVar(value=10)
    self.r = tk.DoubleVar(value=28)
    self.b = tk.DoubleVar(value=2.667)

    # --- Layout ---
    self.frame_controls = ttk.Frame(master, padding=10)
    self.frame_controls.pack(side=tk.LEFT, fill=tk.Y)
    self.frame_plot = ttk.Frame(master)
    self.frame_plot.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

    ttk.Label(self.frame_controls, text="Lorenz Parameters", font=("Arial", 12, "bold"))
    self.add_slider("s", self.s, 0, 30, 0.5)
    self.add_slider("r", self.r, 0, 50, 0.5)
    self.add_slider("b", self.b, 0.5, 5, 0.1)

    ttk.Button(self.frame_controls, text="Run Simulation", command=self.start_thread).pa
    ttk.Button(self.frame_controls, text="Reset View", command=self.reset_view).pack(pac

    self.status = ttk.Label(self.frame_controls, text="Ready", foreground="green")
    self.status.pack(pady=5)

    # --- Figure Setup ---
    self.fig = plt.Figure(figsize=(7, 6), dpi=100)
    self.ax = self.fig.add_subplot(projection="3d")
    self.canvas = FigureCanvasTkAgg(self.fig, master=self.frame_plot)
    self.canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)
    self.ani = None

    # Initial data placeholders
    self.xyzs = None
    self.x_sim = None
    self.t = None

    # Initial plot
    self.run_initial_plot()

def add_slider(self, label, var, frm, to, step):
    frame = ttk.Frame(self.frame_controls)
    frame.pack(pady=5)
    ttk.Label(frame, text=label).pack()
    slider = ttk.Scale(frame, variable=var, from_=frm, to=to, orient=tk.HORIZONTAL)
    slider.pack(fill=tk.X, padx=10)
    ttk.Label(frame, textvariable=var).pack()

def start_thread(self):
    """Starts the simulation computation in a background thread."""
    thread = threading.Thread(target=self.compute_simulation, daemon=True)
    thread.start()

def compute_simulation(self):
    """Run the numerical computations (in background)."""
    try:
        self.status.config(text="Computing...", foreground="orange")

        # Parameters
        s, r, b = self.s.get(), self.r.get(), self.b.get()
        dt = 0.01
        t = np.arange(0, 5, dt)
        init_state = [0., 1., 1.05]

        # Heavy computations here only (no GUI calls)
        xyzs = odeint(lorenz, init_state, t, args=(s, r, b))

        model = ps.SINDy(
            optimizer=ps.STLSQ(threshold=0.05),
            feature_library=ps.PolynomialLibrary(degree=2)
        )
        model.fit(xyzs, t=dt)
        print("\n--- SINDy Model ---")
        model.print()

        x_sim = model.simulate(init_state, t)

        # Store results for main-thread plotting
        self.xyzs, self.x_sim, self.t = xyzs, x_sim, t

        # Schedule the plot update on main thread
        self.master.after(0, lambda: self.update_plot(s, r, b))

    except:
        self.status.config(text="Error", foreground="red")

def update_plot(self, s, r, b):
    """Update Matplotlib plot safely from main thread."""
    self.ax.clear()
```

```
        xyzs = self.xyzs
        x_sim = self.x_sim
        t = self.t

        self.ax.plot(xyzs[:, 0], xyzs[:, 1], xyzs[:, 2],
                     lw=0.5, color='blue', alpha=0.6, label="True Trajectory")
        self.ax.plot(x_sim[:, 0], x_sim[:, 1], x_sim[:, 2],
                     lw=0.5, color='orange', alpha=0.6, label="SINDy Reconstruction")

        (self.point,) = self.ax.plot([xyzs[0, 0]], [xyzs[0, 1]], [xyzs[0, 2]],
                                     marker='o', color='red', markersize=5)

        self.ax.set_xlabel("X Axis")
        self.ax.set_ylabel("Y Axis")
        self.ax.set_zlabel("Z Axis")
        self.ax.set_title(f"Lorenz Attractor (s={s:.2f}, r={r:.2f}, b={b:.2f})")
        self.ax.legend()

        # Adjust limits
        self.ax.set_xlim(np.min(xyzs[:, 0]), np.max(xyzs[:, 0]))
        self.ax.set_ylim(np.min(xyzs[:, 1]), np.max(xyzs[:, 1]))
        self.ax.set_zlim(np.min(xyzs[:, 2]), np.max(xyzs[:, 2]))

        # Stop old animation safely
        if self.ani:
            self.ani.event_source.stop()

        def update(i):
            self.point.set_data(np.array([xyzs[i, 0]]), np.array([xyzs[i, 1]]))
            self.point.set_3d_properties(xyzs[i, 2])
            return (self.point,)

        self.ani = FuncAnimation(self.fig, update, frames=len(t), interval=10, blit=True)
        self.canvas.draw()

        self.status.config(text="Simulation Complete", foreground="green")

    def run_initial_plot(self):
        """Initial blank plot to make GUI load fast."""
        self.ax.set_title("Lorenz Attractor (waiting for simulation)")
        self.ax.set_xlabel("X Axis")
        self.ax.set_ylabel("Y Axis")
        self.ax.set_zlabel("Z Axis")
        self.canvas.draw()

    def reset_view(self):
        self.ax.view_init(elev=30, azim=45)
        self.canvas.draw()


# --- Run the App ---
if __name__ == "__main__":
    root = tk.Tk()
    app = LorenzApp(root)
    root.mainloop()
```

6 Conclusion

The Lorenz system exemplifies how simple deterministic equations can give rise to complex, unpredictable behavior. Through analytical stability analysis, we showed how fixed points bifurcate as system parameters vary, leading to chaotic motion. The Jacobian formulation revealed that trajectories evolve under a continuously contracting phase-space volume, confirming the system's dissipative nature. Furthermore, the SINDy algorithm successfully rediscovered the Lorenz equations from data, illustrating the potential of machine learning in physics-based system identification. Such techniques promise to bridge traditional analytical modeling and modern data science, enabling automated discovery of governing laws in complex dynamical systems.