PHYS 4041 Final Paper

The Lorenz System

Caitlyn Kloeckl
December 17th, 2020

Introduction

The Lorenz System was discovered and studied by Edward Norton Lorenz along with Ellen Cole Fetter. Lorenz created a base for Chaos Theory, predictive modeling, and the understanding of complex, chaotic systems. Building off of the original Lorenz equations, the concept can be applied to meteorology, physics, environmental science, engineering, economics, biology, and more.

Theory

The Lorenz equations are a system of three ordinary differential equations. These equations were derived from a model of convection where a top plate cools and a lower plate is heated—thus simulating a simple model of the atmosphere since the atmosphere cools with an increase in altitude. The following are the Lorenz equations:

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = x(\rho - z) - y$$

$$\frac{dz}{dt} = xy - \beta z$$

The variables *x, y,* and *z* are not spatial coordinates. The variable *x* is proportional to the rate of convection, *y* to the horizontal temperature variation, and *z* to the vertical temperature variation. The constants $\sigma, \rho,$ and β are dependent on the system's characteristics. σ is the Prandtl number, ρ is the Rayleigh number over the critical Rayleigh number, and β is a geometric factor. Lorenz used the following values to show the chaotic behavior:

$$\sigma = 10, \qquad \rho = 28, \qquad \beta = 8/3$$

In order to solve the system of equations, a set of initial values must be set for *x, y,* and *z*.

Algorithm and Written Code Explained

The algorithm chosen to solve the Lorenz equations was the Runge-Kutta fourth-order method as discussed in class. The Runge-Kutta method solves initial value ordinary differential equations. The method uses the provided initial conditions and solves for the next value for the Lorenz equations. The standard formula below applies to a singular ordinary differential equation, thus in the written algorithm, three of the formulas was be applied and intertwined to solve the system of equations. The following is the formula for the Runge-Kutta fourth-order method:

$$k_1 = \Delta t\, f(t_i, y_i)$$

$$k_2 = \Delta t\, f\left(t_i + \frac{\Delta t}{2}, y_i + \frac{k_1}{2}\right)$$

$$k_3 = \Delta t\, f(t_i + \frac{\Delta t}{2}, y_i + \frac{k_2}{2})$$

$$k_4 = \Delta t\, f(t_{i+1}, y_i + k_3)$$

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(\Delta t^5)$$

The algorithm used in practice implements the following algorithm which applies three of the Runge-Kutta systems mentioned above:

$$k_1 \left\{ \begin{aligned} k_{1x} &= \Delta t\, X(x_i, y_i) \\ k_{1y} &= \Delta t\, Y(x_i, y_i, z_i) \\ k_{1z} &= \Delta t\, Z(x_i, y_i, z_i) \end{aligned} \right.$$

$$k_2 \left\{ \begin{aligned} k_{2x} &= \Delta t\, X\left(x_i + \frac{k_{x1}}{2}, y_i + \frac{k_{y1}}{2}\right) \\ k_{2y} &= \Delta t\, Y\left(x_i + \frac{k_{x1}}{2}, y_i + \frac{k_{y1}}{2}, z_i + \frac{k_{z1}}{2}\right) \\ k_{2z} &= \Delta t\, Z\left(x_i + \frac{k_{x1}}{2}, y_i + \frac{k_{y1}}{2}, z_i + \frac{k_{z1}}{2}\right) \end{aligned} \right.$$

$$k_3 \begin{cases} k_{3x} = \Delta t\, X\left(x_i + \dfrac{k_{x2}}{2}, y_i + \dfrac{k_{y2}}{2}\right) \\ k_{3y} = \Delta t\, Y\left(x_i + \dfrac{k_{x2}}{2}, y_i + \dfrac{k_{y2}}{2}, z_i + \dfrac{k_{z2}}{2}\right) \\ k_{3z} = \Delta t\, Z\left(x_i + \dfrac{k_{x2}}{2}, y_i + \dfrac{k_{y2}}{2}, z_i + \dfrac{k_{z2}}{2}\right) \end{cases}$$

$$k_4 \begin{cases} k_{4x} = \Delta t\, X(x_i + k_{x3}, y_i + k_{y3}) \\ k_{4y} = \Delta t\, Y(x_i + k_{x3}, y_i + k_{y3}, z_i + k_{z3}) \\ k_{4z} = \Delta t\, Z(x_i + k_{x3}, y_i + k_{y3}, z_i + k_{z3}) \end{cases}$$

$$i + 1\ values \begin{cases} x_{i+1} = x_i + \dfrac{1}{6}(k_{1x} + 2k_{2x} + 2k_{3x} + k_{4x}) \\ y_{i+1} = y_i + \dfrac{1}{6}(k_{1y} + 2k_{2y} + 2k_{3y} + k_{4y}) \\ z_{i+1} = z_i + \dfrac{1}{6}(k_{1z} + 2k_{2z} + 2k_{3z} + k_{4z}) \end{cases}$$

Where the functions X, Y, and Z are referring to the Lorenz Equations as follows:

$$X(x, y) = \frac{dx}{dt} = \sigma(y - x)$$

$$Y(x, y, z) = \frac{dy}{dt} = x(\rho - z) - y$$

$$Z(x, y, z) = \frac{dz}{dt} = xy - \beta z$$

The algorithm described above was then coded into the function `RK4_Lorenz()` as follows:

```
def RK4_Lorenz(starttime, endtime, N_pts, x0, y0, z0, sigma, rho, beta):
    t = np.linspace(starttime, endtime, N_pts)
    dt = t[1] - t[0]
    x = np.zeros(N_pts)
    x[0] = x0
    y = np.zeros(N_pts)
    y[0] = y0
    z = np.zeros(N_pts)
    z[0] = z0

    for i in range(N_pts - 1):

        k1 = dt * X(x[i], y[i],       sigma)
        l1 = dt * Y(x[i], y[i], z[i], rho)
        m1 = dt * Z(x[i], y[i], z[i], beta)

        k2 = dt * X(x[i] + 0.5 * k1, y[i] + 0.5 * l1,                  sigma)
        l2 = dt * Y(x[i] + 0.5 * k1, y[i] + 0.5 * l1, z[i] + 0.5 * m1, rho)
        m2 = dt * Z(x[i] + 0.5 * k1, y[i] + 0.5 * l1, z[i] + 0.5 * m1, beta)

        k3 = dt * X(x[i] + 0.5 * k2, y[i] + 0.5 * l2,                  sigma)
        l3 = dt * Y(x[i] + 0.5 * k2, y[i] + 0.5 * l2, z[i] + 0.5 * m1, rho)
        m3 = dt * Z(x[i] + 0.5 * k2, y[i] + 0.5 * l2, z[i] + 0.5 * m1, beta)

        k4 = dt * X(x[i] + k3, y[i] + l3,                  sigma)
        l4 = dt * Y(x[i] + k3, y[i] + l3, z[i] + 0.5 * m1, rho)
        m4 = dt * Z(x[i] + k3, y[i] + l3, z[i] + 0.5 * m1, beta)

        x[i + 1] = x[i] + (k1 + 2 * k2 + 2 * k3 + k4) / 6
        y[i + 1] = y[i] + (l1 + 2 * l2 + 2 * l3 + l4) / 6
        z[i + 1] = z[i] + (m1 + 2 * m2 + 2 * m3 + m4) / 6

    return x, y, z, t
```

The function returns 4 arrays—*x, y, z, and t*—after implementing the Runge-Kutta fourth-order method. It takes in values for the start time, end time, the total number of points, initial values for x, y, and z, and constant values for $\sigma, \rho,$ and β. The for loop in the `RK4_Lorenz()` function calls Lorenz equations coded as follows:

```
def X(x, y, sigma):
    return sigma * (y - x)

def Y(x, y, z, rho):
    return x*(rho - z) - y

def Z(x, y, z, beta):
    return x * y - beta * z
```

Once the arrays—*x, y, z, and t*—were generated, the arrays were then used by the functions `Lorenz_Bifurcation_Plot()`, `Lorenz_2D_Plot()`, and `Lorenz_3D_Plot()` (shown in Appendix A) and produced all of the following plots to describe the physical phenomena.

Implementation

Implementing the algorithm above displays the uniqueness of the Lorenz system. Figure 1 depicts a typical output with constants $\sigma = 10,\ \rho = 28$, and $\beta = 8/3$.

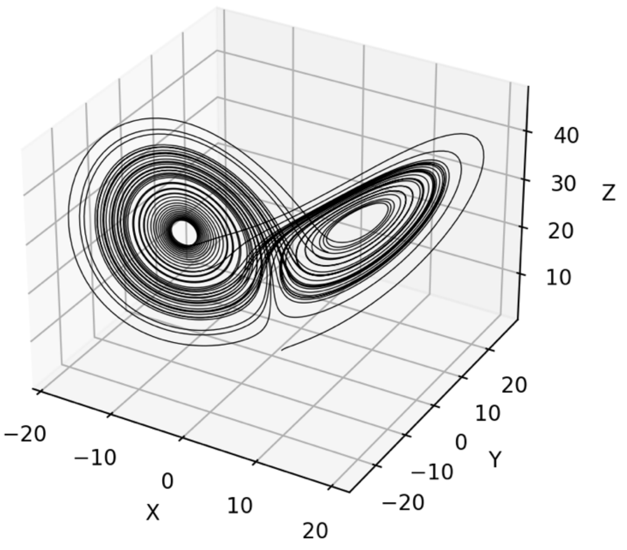

Figure 1: The Lorenz system plotted with $\sigma = 10,\ \rho = 28$, *and* $\beta = 8/3$. *Initial values were set to* $x_0 = y_0 = z_0 = 1$.

The method of calculating each point (X, Y, Z) is very accurate with an error of $O(\Delta t^5)$ as derived from the Runge-Kutta algorithm. The main stability of this system of equations depends on the initial conditions. One small change in the initial conditions has a larger effect on the x, y, and z values as time increases.

However, the effect of one small change in the initial conditions is a key effect of chaos, specifically deterministic chaos. One small change in the initial conditions—in this case, a difference of 10^{-7}—leads to significant change in the outcome, even though the equations are derived from physical laws. This effect is shown to in Figure 2. If both plots were identical, there would only be one color. When taking a closer look at the plot, the lines diverge. This is more obvious when the X, Y, and Z values are isolated as is shown more next.

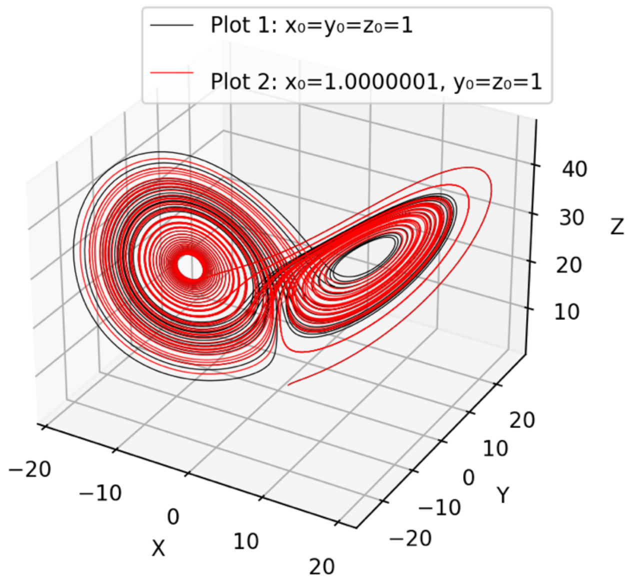

Figure 2: Two Lorenz systems plotted on top of each other with a 10^{-6} *difference in* x_0. *This was generated by* `Lorenz_2D_Plot()` *in Appendix A.*

This image below singles out the X-values from Figure 2 above. Towards the beginning of the system, the X values are seemingly identical. However, the system reaches $t = 30$, the paths diverge.

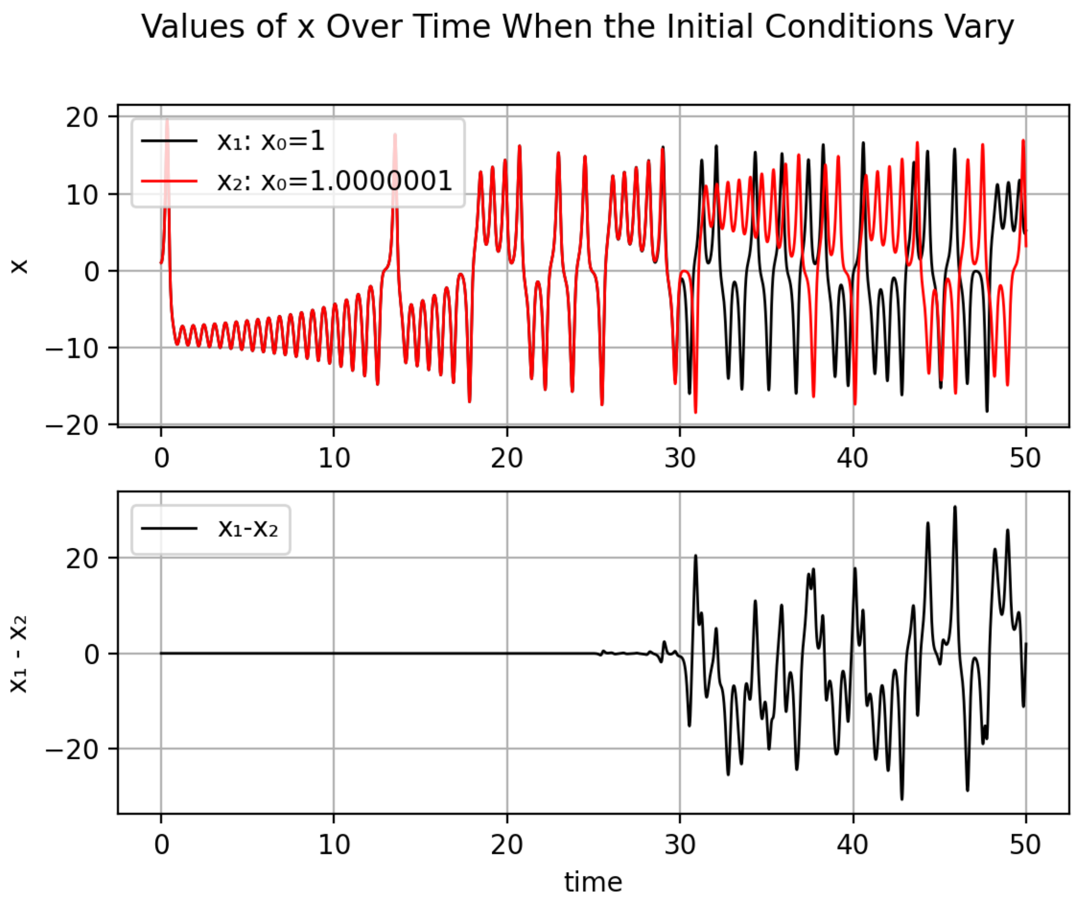

Figure 3: X values of the Lorenz system over time. For this system y_0 and z_0 were set to equal 1. The constants for both x_1 and x_2 were $\sigma = 10,\ \rho = 28$, and $\beta = 8/3$. The top plot is x_1 and x_2 plotted over time and the bottom plot is $x_1 - x_2$ over time. This was generated by `Lorenz_2D_Plot()` *in Appendix A.*

This principle is why it is still difficult to predict weather further out in time. A fundamental part of these chaotic models is that the initial conditions must be known with *infinite* accuracy, otherwise, the predictive model fails after an extended period of time.

It is important to note that y_0 and z_0 were set to equal 1 for both x_1 and x_2. Only the x_0 values varied. This variation in x also had a similar effect on the Y and Z values as is shown in Figure 4 to the right. The small change in x_0 also had a very similar effect on X and Y as they both diverge at $t = 30$.

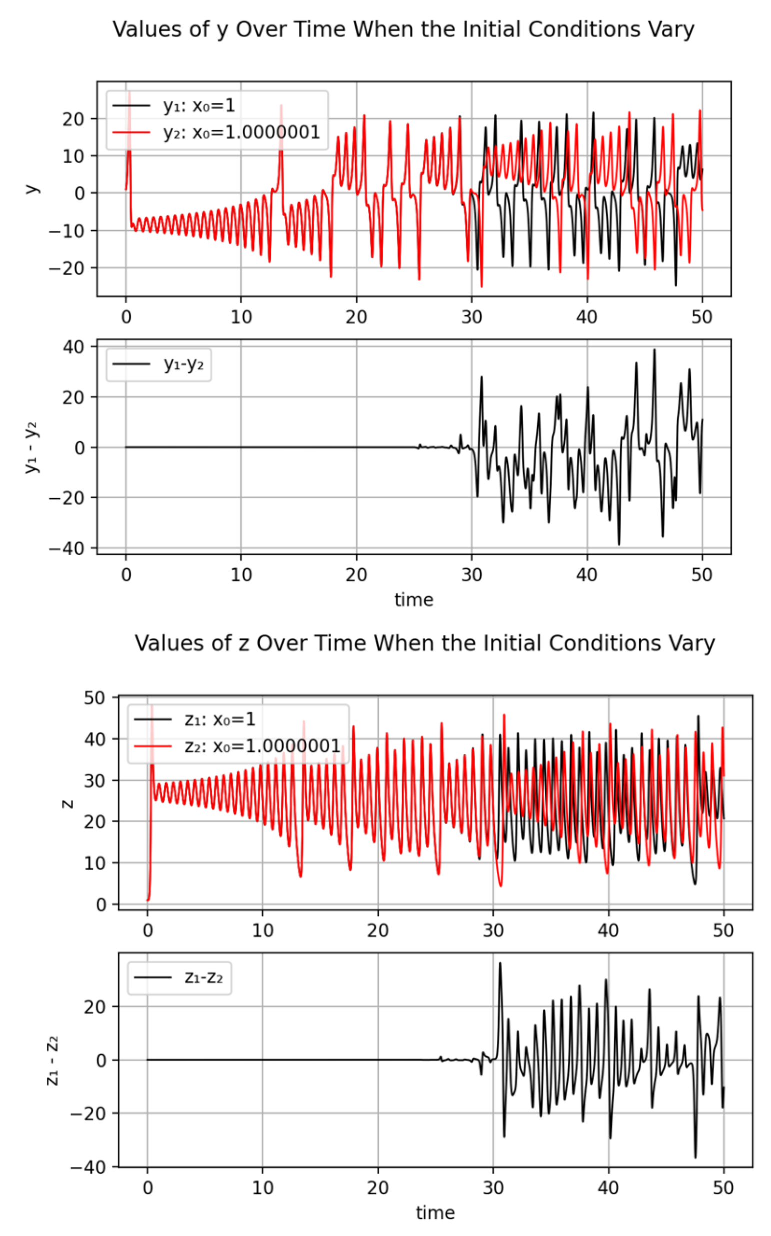

Figure 4: X and Y values of the Lorenz system over time. Compared to Figure 3, X and Y exhibit very similar behavior when x_0 is varied. This was generated by `Lorenz_2D_Plot()` *in Appendix A.*

Bifurcation

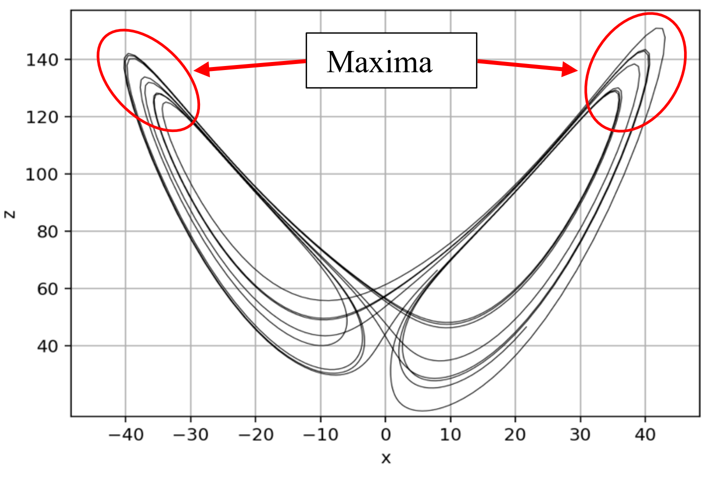

Figure 5: A plot defining the maxima of Z when the Lorenz equation is plotted in X and Z. This was generated by a modified version of `Lorenz_2D_Plot()` *in Appendix A.*

The Lorenz system discussed above had specific values chosen for σ, ρ, and β. These were chosen because they exhibit the classical chaotic behavior. By definition, chaos is disordered and unpredictable. However, there have been strategies discovered to somewhat order the chaos. This is done is by studying the structural changes of systems while varying one of the parameters, also known as bifurcation theory.

In order to study the bifurcation, or branching, in the Lorenz equations, one of the constants (σ, ρ, or β) must be slowly changed over many iterations of the Lorenz system. For the following analysis, the parameter chosen to vary was ρ, and the structural change chosen to analyze is the local maxima of Z. Shown below is plot of how the local maxima of Z evolve when the value of ρ is varied (the other constants are held at σ=10 and β=6). The maxima of Z for the analysis are defined In Figure 5.

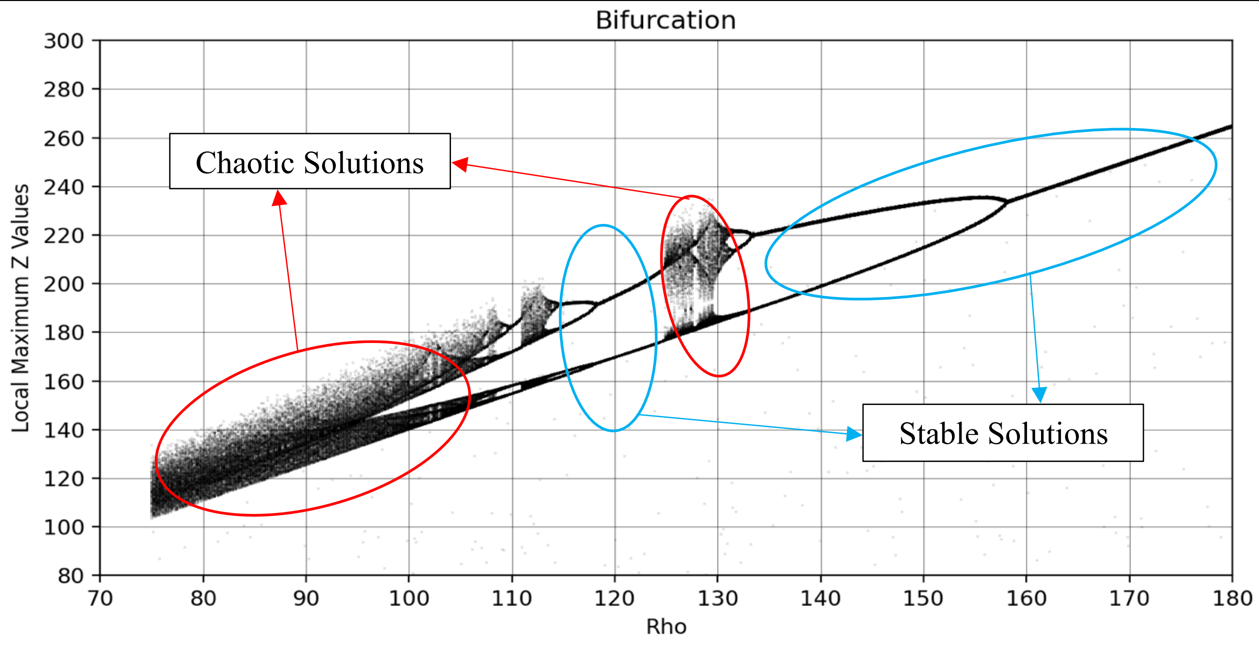

Figure 6: A plot of how the local maxima of Z in the Lorenz system evolve when ρ is varied. The chaotic solutions are labeled in red. The stable solutions are labeld in blue. This was generated by `Lorenz_Bifurcation_Plot()` *in Appendix A.*

As shown in Figure 6, an interesting pattern results from the analysis. Key factors are labeled in the plot. The chaotic solutions labeled in red is what was described in implementation section—non-repeating, unpredictable solutions. The bifurcation plot is broken down in the following diagram.

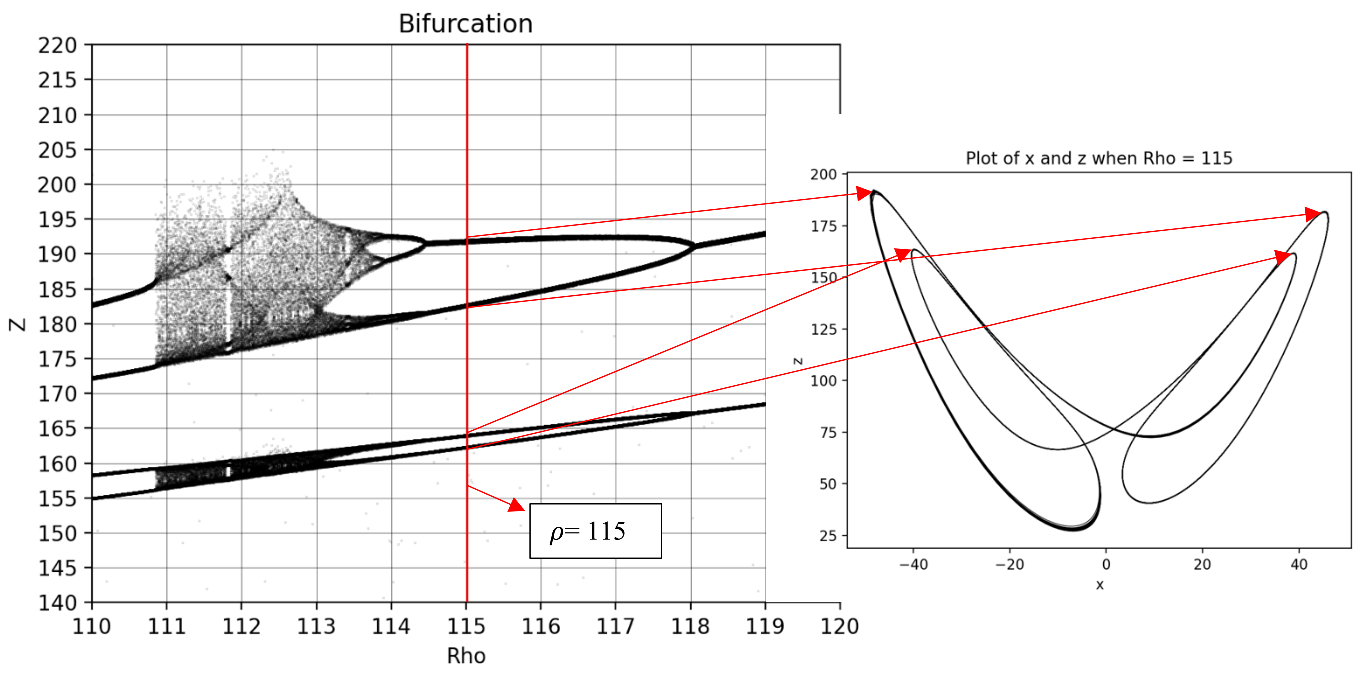

Figure 7: To the left, a section of the plot from Figure 6 is shown, centered around ρ=115. To the right, a plot of the Lorenz equations in x and z are shown for a significant amount of time with ρ =115. This was generated by `Lorenz_Bifurcation_Plot()` *along with in a modified version of* `Lorenz_2D_Plot()` *in Appendix A.*

In Figure 7, to the left, a smaller section of the bifurcation plot was generated. To the right, the arrows point to what the bifurcation plot represents in regard to the Lorenz equations when ρ=115. The figure to the right was generated over an extended period of time presenting a stable solution to the Lorenz equations. Figure 7 reveals a repeating and predictable pattern—Z and X values are cyclical.

This is useful for many reasons. From chaos, this method derived a predictable pattern for a specific value of ρ=115 (when σ and β were held constant) which is stable and periodic. Figure 6 shows a map of the system and its behavior for values of ρ. If the same analysis were to b be done for σ, and β, similar plots could be generated, thus gaining further insight to the system as a whole.

Conclusion

In conclusion, the Lorenz system of ordinary differential equations are easily and accurately modeled using the Runge-Kutta fourth-order method. Using the Runge-Kutta method allows an accurate insight into how the Lorenz system acts and how the system is structured. The analysis of the Lorenz system gives insight into fundamental features of chaotic phenomena. This includes that chaotic systems are highly dependent and sensitive to initial conditions (deterministic). It also gives insight to the concept that although many systems may seem random, many can be exactly calculated and predicted if the variables involved are known with infinite accuracy.

The analysis of the Lorenz system also gives insight to the bifurcation, or branching, of the system. By slightly varying a constant in the Lorenz system, the larger structure of the system was revealed. All of these factors help further understand seemingly unpredictable systems and allows patterns to be utilize.

Works Cited

Yang, Xin-She. "Lorenz Equation." *Lorenz Equation - an Overview | ScienceDirect Topics*, 2017, www.sciencedirect.com/topics/mathematics/lorenz-equation.

Weisstein, Eric W. "Lorenz Attractor." From *MathWorld*--A Wolfram Web Resource. https://mathworld.wolfram.com/LorenzAttractor.html.

"Lorenz System." *Wikipedia*, Wikimedia Foundation, 29 Oct. 2020, https://en.wikipedia.org/wiki/Lorenz_system

"Bifurcation Theory." *Wikipedia*, Wikimedia Foundation, 4 Dec. 2020, https://en.wikipedia.org/wiki/Bifurcation_theory.

Guckenheimer, John. "Bifurcation." *Scholarpedia*, http://www.scholarpedia.org/article/Bifurcation.

Shen, Bing Lu, et al. "Stable and Unstable Regions of the Lorenz System." *Nature News*, Nature Publishing Group, 8 Oct. 2018, https://www.nature.com/articles/s41598-018-33010-z.

Sun, Kehui, and J. C. Sprott. *Bifurcations of Fractional-Order Diffusionless Lorenz System* . https://arxiv.org/pdf/0907.2077.pdf.

"Hopf Bifurcation." *Wikipedia*, Wikimedia Foundation, 12 Nov. 2020, https://en.wikipedia.org/wiki/Hopf_bifurcation.

Appendix

Appendix A: Python Code to Produce Plots

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

# the three differential equations:
def X(x, y, sigma):
    return sigma * (y - x)


def Y(x, y, z, rho):
    return x*(rho - z) - y


def Z(x, y, z, beta):
    return x * y - beta * z


# using the runge-kutta method to solve these 3 differential equations:
def RK4_Lorenz(starttime, endtime, N_pts, x0, y0, z0, sigma, rho, beta):
    t = np.linspace(starttime, endtime, N_pts)
    dt = t[1] - t[0]

    x = np.zeros(N_pts)
    x[0] = x0

    y = np.zeros(N_pts)
    y[0] = y0

    z = np.zeros(N_pts)
    z[0] = z0

    for i in range(N_pts - 1):

        k1 = dt * X(x[i], y[i],       sigma)
        l1 = dt * Y(x[i], y[i], z[i], rho)
        m1 = dt * Z(x[i], y[i], z[i], beta)

        k2 = dt * X(x[i] + 0.5 * k1, y[i] + 0.5 * l1,                  sigma)
        l2 = dt * Y(x[i] + 0.5 * k1, y[i] + 0.5 * l1, z[i] + 0.5 * m1, rho)
        m2 = dt * Z(x[i] + 0.5 * k1, y[i] + 0.5 * l1, z[i] + 0.5 * m1, beta)

        k3 = dt * X(x[i] + 0.5 * k2, y[i] + 0.5 * l2,                  sigma)
        l3 = dt * Y(x[i] + 0.5 * k2, y[i] + 0.5 * l2, z[i] + 0.5 * m1, rho)
        m3 = dt * Z(x[i] + 0.5 * k2, y[i] + 0.5 * l2, z[i] + 0.5 * m1, beta)

        k4 = dt * X(x[i] + k3, y[i] + l3,                  sigma)
        l4 = dt * Y(x[i] + k3, y[i] + l3, z[i] + 0.5 * m1, rho)
        m4 = dt * Z(x[i] + k3, y[i] + l3, z[i] + 0.5 * m1, beta)

        x[i + 1] = x[i] + (k1 + 2 * k2 + 2 * k3 + k4) / 6
        y[i + 1] = y[i] + (l1 + 2 * l2 + 2 * l3 + l4) / 6
        z[i + 1] = z[i] + (m1 + 2 * m2 + 2 * m3 + m4) / 6

    return x, y, z, t
```

```python
def Lorenz_3D_Plot():
    # first scenario:
    #(starttime, endtime, N_pts, x0, y0, z0,sigma,rho,beta)
    points1 = RK4_Lorenz(0, 50, 10000, 1, 1, 1, 10, 28, 8/3)
    t1 = points1[3]
    z1 = points1[2]
    y1 = points1[1]
    x1 = points1[0]

    # second scenario:
    points2 = RK4_Lorenz(0, 50, 10000, 1.000001, 1, 1, 10, 28, 8/3)
    t2 = points2[3]
    z2 = points2[2]
    y2 = points2[1]
    x2 = points2[0]

    # plot 3d graph
    fig = plt.figure()
    ax = fig.gca(projection="3d")
    ax.plot(x1, y1, z1, 'k', linewidth=0.5)
    ax.plot(x2, y2, z2, 'r', linewidth=0.5)
    plt.legend(['Plot 1: x₀=y₀=z₀=1', '\nPlot 2: x₀=1.0000001, y₀=z₀=1'])
    plt.draw()
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    plt.show()



def Lorenz_2D_Plot():
    # first scenario:
    # (starttime, endtime, N_pts, x0, y0, z0,sigma,rho,beta)
    points1 = RK4_Lorenz(0, 50, 10000, 1, 1, 1, 10, 28, 8 / 3)
    t1 = points1[3]
    z1 = points1[2]
    y1 = points1[1]
    x1 = points1[0]

    # second scenario:
    points2 = RK4_Lorenz(0, 50, 10000, 1.0000001, 1, 1, 10, 28, 8 / 3)
    t2 = points2[3]
    z2 = points2[2]
    y2 = points2[1]
    x2 = points2[0]


    fig, axs = plt.subplots(2)
    fig.suptitle('Values of x Over Time When the Initial Conditions Vary')


    # plot 2d plot of both x values
    axs[0].set(ylabel="x")
    axs[0].plot(t1, x1, 'k', t2, x2, 'r', linewidth=1)
    axs[0].legend(['x₁: x₀=1', 'x₂: x₀=1.0000001'])
    axs[0].grid(True)

    # plot the difference of x1 and x2 over time
    plt.xlabel("time")
    plt.ylabel("x₁ − x₂")
    axs[1].plot(t1, x1-x2, 'k', linewidth=1)
    axs[1].legend(['x₁-x₂'])
    axs[1].grid(True)
    plt.show()


    # how this effects the Y values:
    fig, axs = plt.subplots(2)
    fig.suptitle('Values of y Over Time When the Initial Conditions Vary')
    # plot 2d plot of both y values
    axs[0].set(ylabel="y")
    axs[0].plot(t1, y1, 'k', t2, y2, 'r', linewidth=1)
    axs[0].legend(['y₁: x₀=1', 'y₂: x₀=1.0000001'])
    axs[0].grid(True)

    # plot the difference of y1 and y2 over time
    plt.xlabel("time")
    plt.ylabel("y₁ − y₂")
    axs[1].plot(t1, y1 - y2, 'k', linewidth=1)
    axs[1].legend(['y₁-y₂'])
    axs[1].grid(True)
    plt.show()

    # how this effects the Z values:
    fig, axs = plt.subplots(2)
    fig.suptitle('Values of z Over Time When the Initial Conditions Vary')
    # plot 2d plot of both z values
    axs[0].set(ylabel="z")
    axs[0].plot(t1, z1, 'k', t2, z2, 'r', linewidth=1)
    axs[0].legend(['z₁: x₀=1', 'z₂: x₀=1.0000001'])
    axs[0].grid(True)

    # plot the difference of z1 and z2 over time
    plt.xlabel("time")
    plt.ylabel("z₁ − z₂")
    axs[1].plot(t1, z1 - z2, 'k', linewidth=1)
    axs[1].legend(['z₁-z₂'])
    axs[1].grid(True)
    plt.show()


def Lorenz_Bifurcation_Plot():
    # setting constants for Bifurcation
    R_resolution = 1000
    R_values = np.linspace(75, 180, R_resolution)
    loading = np.linspace(1, 100, R_resolution)
    x0 = y0 = z0 = 1

    # making empty lists, these will be the values plotted
    Zmax_values_toPlot = []
    R_values_toPlot = []

    # bifurcation for various values of R (rho)
    for index in range(R_resolution):
        R = R_values[index]
        N_pts = 10000

        # (starttime, endtime, N_pts, x0, y0, z0,sigma,rho,beta)
        points1 = RK4_Lorenz(0, 50, N_pts, x0, y0, z0, 10, R, 6)
        t1 = points1[3]
        z1 = points1[2]
        y1 = points1[1]
        x1 = points1[0]

        # just to make sure the program is making progress:
        if round(loading[index - 1]) < round(loading[index]):
            print('Bifurcation Loading...', round(loading[index]), '%')

        # find local maxima to plot
        for zindex in range(len(z1)-1):
            current_z = z1[zindex]
            previous_z = z1[zindex - 1]
            next_z = z1[zindex + 1]

            # identify the local maxes by definition of a max:
            if previous_z < current_z and current_z > next_z:
                R_values_toPlot.append(R)
                Zmax_values_toPlot.append(current_z)


        # set next lorenz run to start where we left off:
        x0 = x1[-1]
        y0 = y1[-1]
        z0 = z1[-1]

    # Bifurcation Plot
    xlabel("Rho")
    ylabel("Z")
    title("Bifurcation")
    plt.plot(R_values_toPlot, Zmax_values_toPlot, linestyle='none', marker=".",
markersize=0.1, color='k')
    grid(color='k', linestyle='-', linewidth=0.20)
    plt.xlim(75, 180)
    plt.ylim(80, 300)
    plt.xticks(np.arange(70, 181, step=10))
    plt.yticks(np.arange(80, 301, step=20))
    plt.show()



# Runs the File
Lorenz_3D_Plot()
Lorenz_2D_Plot()
Lorenz_Bifurcation_Plot()
```