Computational Physics Project

Lorenz System

Name: Yogeshwaran R

Roll No: B200161EP

Introduction:

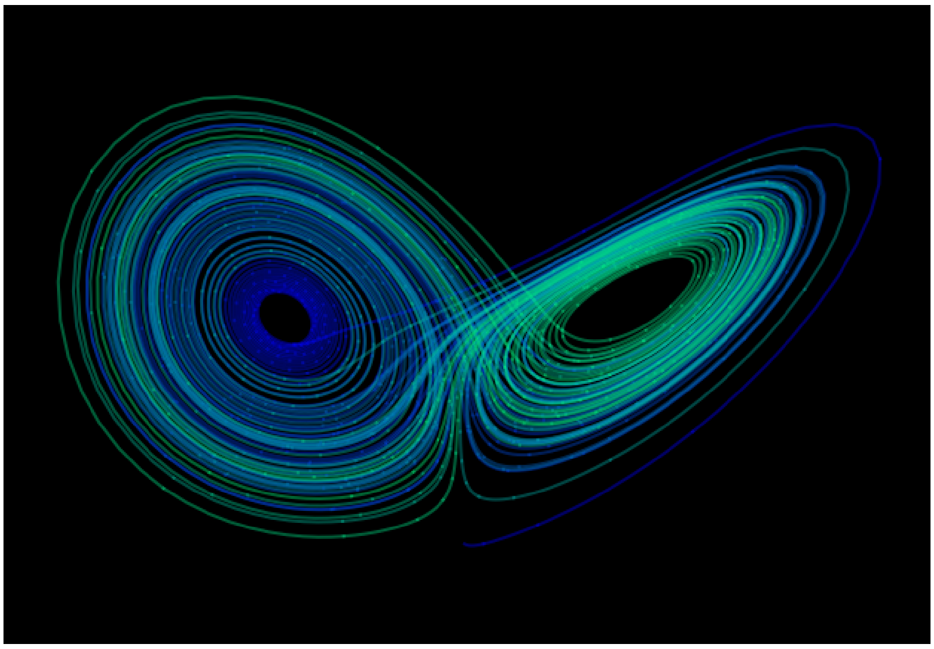
The Lorenz system is a set of nonlinear differential equations that describe a chaotic system first studied by mathematician Edward Lorenz. It has become a well-known example of a dynamical system that exhibits chaotic behaviour. The system has been studied extensively in mathematics, physics, and other fields, and has numerous applications in science and engineering.

The Lorenz System Formula:

The Lorenz system consists of three nonlinear differential equations:

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

where x, y, and z are the variables, t is the time, and σ, ρ, and β are the parameters. The values of these parameters determine the behaviour of the system.



The Lorenz system exhibits a number of interesting phenomena, such as chaotic behaviour and sensitivity to initial conditions. Chaotic behaviour refers to the seemingly random and unpredictable motion of the system over time, even though the equations governing the system are deterministic. This property has made the Lorenz system an important tool for studying chaos theory and nonlinear dynamics.

Application of the Lorenz System:

- The Lorenz system has been used as a model for a wide variety of natural phenomena, including fluid flow, weather patterns, and the dynamics of the heart.

- The Lorenz system has been used to study turbulence, which is a chaotic phenomenon that is difficult to predict and control. By understanding the dynamics of the Lorenz system, researchers have gained insights into how turbulence arises and how it can be controlled.

- The Lorenz system has also been used to study the onset of chaos in physical systems. It is a relatively simple system that exhibits chaotic behaviour, making it an ideal model for studying the transition from order to chaos.

- The Lorenz system has applications in cryptography and secure communications. Chaotic systems like the Lorenz system are difficult to predict, which makes them useful for generating random sequences that can be used for encryption.

- The Lorenz system has been used as a test bed for numerical methods for solving differential equations. The system is relatively simple, but it exhibits complex behaviour, making it an ideal model for testing the accuracy and efficiency of numerical methods.

- Applications of the Lorenz system include weather forecasting, fluid dynamics, and control theory. In weather forecasting, the system has been used to model atmospheric convection and the behaviour of storms. In fluid dynamics, the Lorenz system has been used to study turbulence and the behaviour of fluids in complex systems. In control theory, the Lorenz system has been used as a test case for evaluating different control strategies and algorithms.

Project Description:

The project presented is a simulation of the Lorenz system using the Python programming language. The program uses the fourth-order Runge-Kutta method to solve the differential equations that define the system. The program then animates the solution in a 3D plot, showing the trajectory of the system as it evolves over time. The program also highlights a point on the trajectory at each time step to help visualise the movement of the system.

The simulation produced a chaotic trajectory in three dimensions, with the x, y, and z variables oscillating and intersecting each other. The animation showed the trajectory of the system as it evolved over time, with a blue dot representing the current position of the system at each time step. The text box displayed the current time of the simulation.

Runge-Kutta 4th Order

Runge-Kutta 4th Order method (RK4) is a numerical method used to solve ordinary differential equations (ODEs). The method is a modification of the Euler method that improves its accuracy by using intermediate slopes to estimate the next point in the solution.

The basic idea behind RK4 is to calculate four slopes (k1, k2, k3, k4) at each step and use them to estimate the next point in the solution. The formula for the RK4 method is as follows:

$$\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right), \\
k_3 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right), \\
k_4 &= f(t_n + h, y_n + hk_3).
\end{aligned}$$

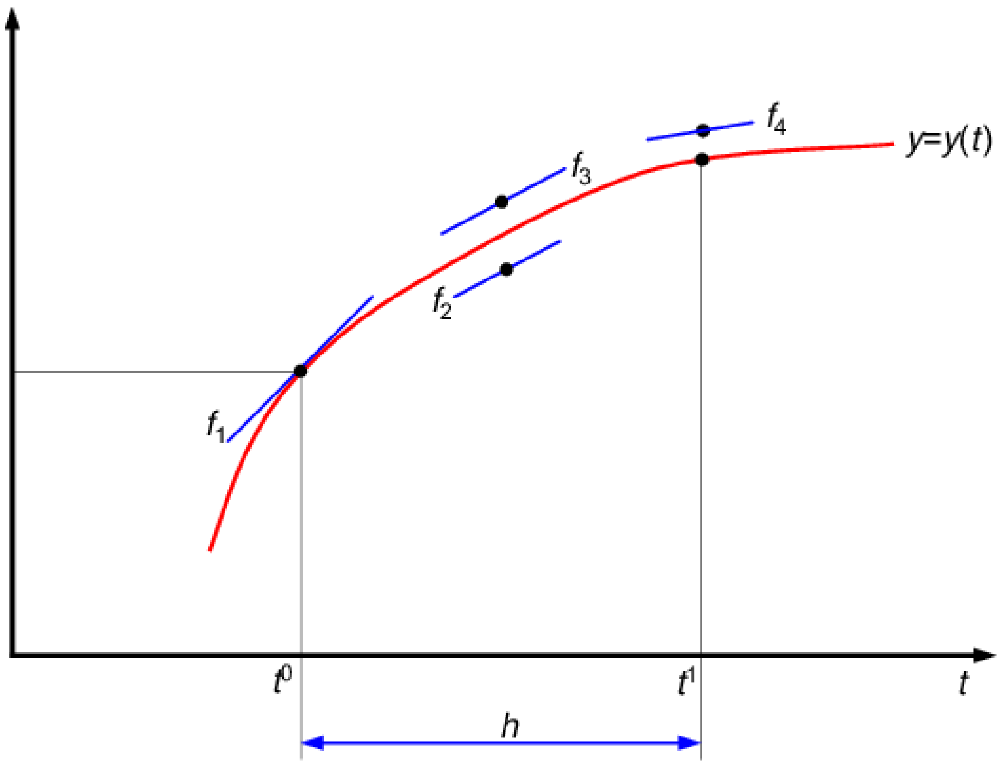
$$
\begin{aligned}
y_{n+1} &= y_n + \tfrac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right), \\
t_{n+1} &= t_n + h
\end{aligned}
$$

for $n = 0, 1, 2, 3, \ldots$, using[2]

$$
\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right), \\
k_3 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right), \\
k_4 &= f\left(t_n + h, y_n + hk_3\right).
\end{aligned}
$$

where h is the step size, t_n is the current time, y_n is the current value of the solution, and f is the function that defines the ODE. The next value of the solution is given by y_{n+1}
RK4 is a fourth-order method, which means that the error in the method is proportional to h^4. This makes it more accurate than lower-order methods like Euler's method, which have an error proportional to h^2.

Geometric interpretation of RK4



Overall, RK4 is a widely used numerical method for solving ODEs because of its accuracy and simplicity. It is a reliable tool for approximating solutions to ODEs in cases where analytical solutions are not possible or practical to obtain.

Conclusion:

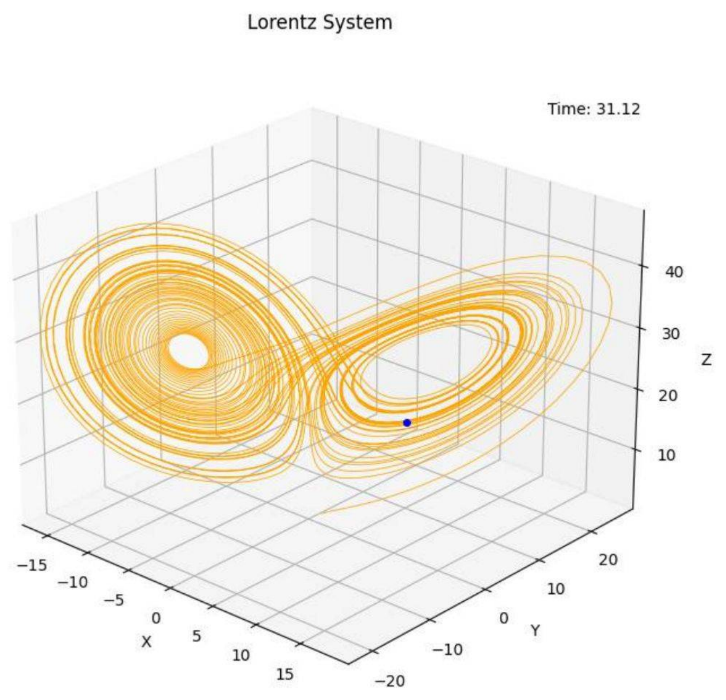
The Lorenz system is a fascinating example of a chaotic system. It has been used extensively in the study of nonlinear dynamics and chaos, and has numerous applications in science and engineering. The project presented above provides a simple but effective demonstration of the behaviour of the Lorenz system, and serves as a useful introduction to the topic.

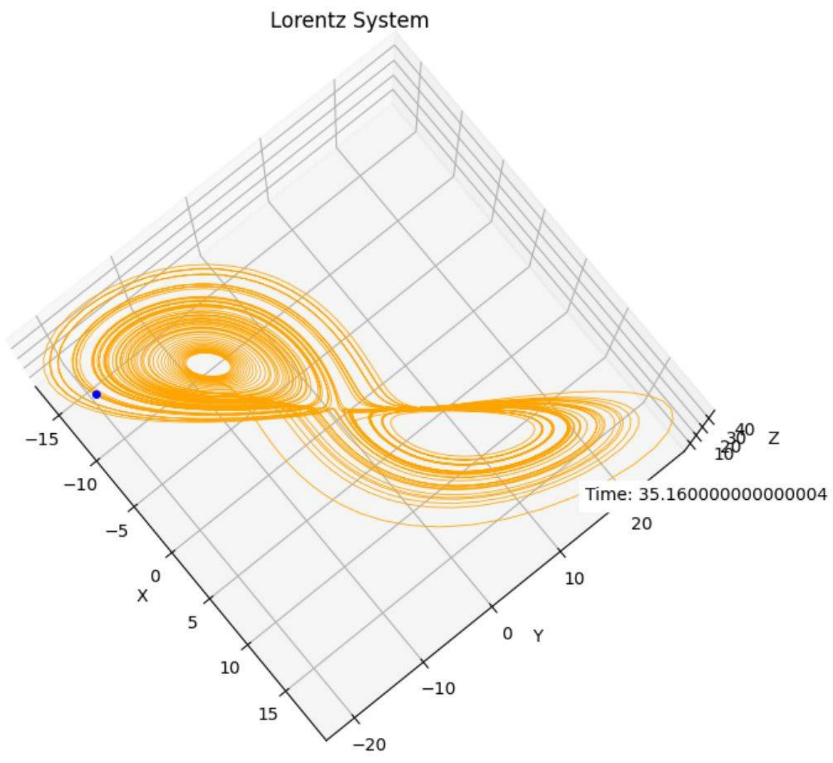
Code :

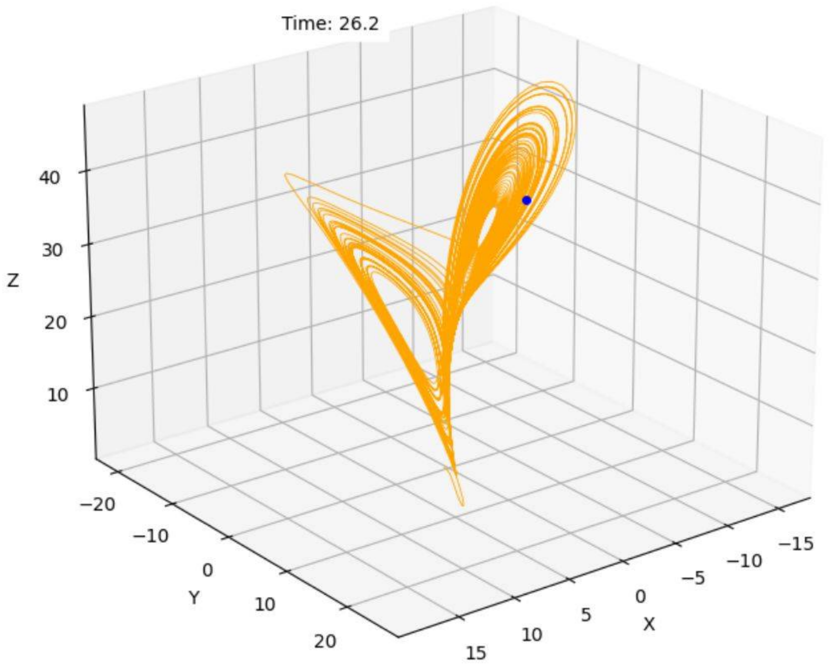
```
                 # Lorenz system  ~ yogeshwaran

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Define the constants and initial conditions
sigma = 10
rho = 28                         # change the values for different
result
beta = 8/3
x0, y0, z0 = 1, 1, 1             # initial condt
t0, tf, h = 0, 60, 0.01          # h-(stepsize)

# Define the Lorenz system
def lorenz(t, xyz, sigma, rho, beta):
   x, y, z = xyz
   dxdt = sigma * (y - x)
   dydt = x * (rho - z) - y          #lorenz eqns
   dzdt = x * y - beta * z
   return [dxdt, dydt, dzdt]

# Define the function to solve the Lorenz system using the fourth-order
Runge-Kutta method
def solve_lorenz(x0, y0, z0, sigma, rho, beta, t0, tf, h):
   t = np.arange(t0, tf, h)
   n = len(t)
   x = np.zeros(n)
   y = np.zeros(n)
   z = np.zeros(n)
   x[0], y[0], z[0] = x0, y0, z0
   for i in range(n-1):                        # fourth-order Runge-Kutta
method
       k1 = h * np.array(lorenz(t[i], [x[i], y[i], z[i]], sigma, rho,
beta))
       k2 = h * np.array(lorenz(t[i] + 0.5*h, [x[i] + 0.5*k1[0], y[i] +
0.5*k1[1], z[i] + 0.5*k1[2]], sigma, rho, beta))
       k3 = h * np.array(lorenz(t[i] + 0.5*h, [x[i] + 0.5*k2[0], y[i] +
0.5*k2[1], z[i] + 0.5*k2[2]], sigma, rho, beta))
```

```
        k4 = h * np.array(lorenz(t[i] + h, [x[i] + k3[0], y[i] + k3[1],
z[i] + k3[2]], sigma, rho, beta))
        x[i+1] = x[i] + (1/6) * (k1[0] + 2*k2[0] + 2*k3[0] + k4[0])
        y[i+1] = y[i] + (1/6) * (k1[1] + 2*k2[1] + 2*k3[1] + k4[1])
        z[i+1] = z[i] + (1/6) * (k1[2] + 2*k2[2] + 2*k3[2] + k4[2])
    return t, x, y, z

# Solving the Lorenz system using the fourth-order Runge-Kutta method
t, x, y, z = solve_lorenz(x0, y0, z0, sigma, rho, beta, t0, tf, h)


# animation function
def animate(i):
    if 2*i>=len(t):             #avoids bound error of x,y,z array
        exit()
    ax.clear()
    ax.set_xlim((np.min(x), np.max(x)))
    ax.set_ylim((np.min(y), np.max(y)))
    ax.set_zlim((np.min(z), np.max(z)))
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_title('Lorentz System')
    ax.plot(x, y, z, color='orange',linewidth=0.6)
#plot of trajectory
    ax.plot(x[2*i], y[2*i], z[2*i], 'o',color='blue',markersize=4)
#plot of dot
    # skipping some points to make animation faster
    ax.text(20, 10, 70, "Time: {}".format(t[2*i]), color='black',
backgroundcolor='white')

# Create the figure and axes
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Creating the animation
animation = FuncAnimation(fig, animate, frames=len(t), interval=1)

# Showing the animation
plt.show()
```

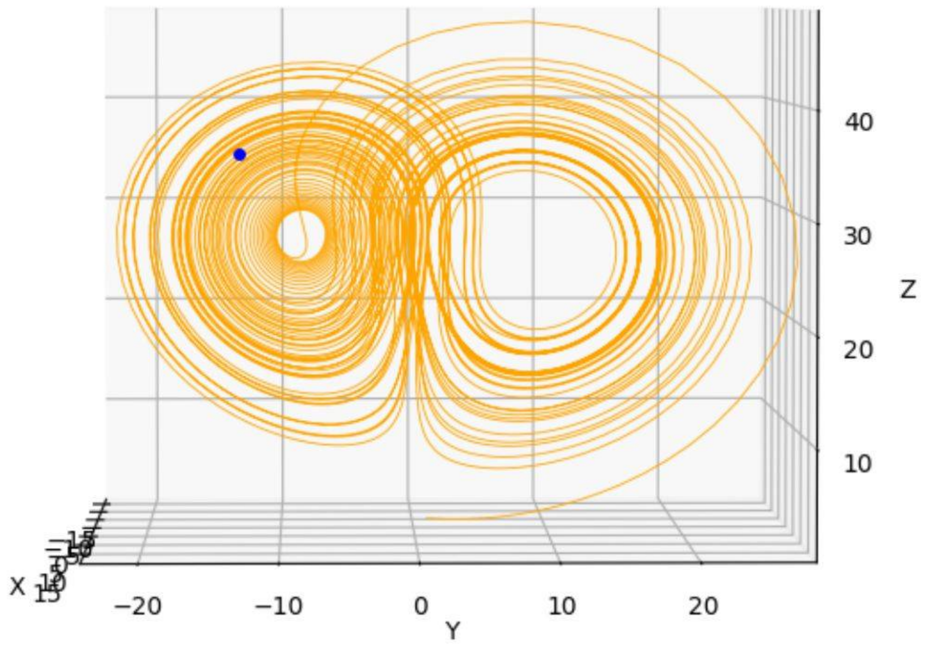
Results: Different projection of Plots







Time: 26.2
Z
Y
X



Z
Y
X