Ordinary Differential Equations

Applications Using Numerical Methods

Cosmic Charade

In physics, many systems are governed by ordinary differential equations (ODEs). These equations describe how a physical quantity changes over time or space. Analytical solutions to ODEs exist only for simple cases, but in real-world physics problems, exact solutions are often not possible. This is where **numerical methods** like **Euler's Method** and the **Runge-Kutta Method** become essential. These methods allow us to approximate solutions to ODEs, enabling us to model complex systems that appear in classical mechanics, electromagnetism, quantum mechanics, and beyond.

In this material, we will explore some fundamental physics problems, solve their ODEs using numerical methods, and visualize the results. This guide is targeted at undergraduate physics students to help them understand how numerical methods can be applied to real-world physics problems.

Modeling Simple Harmonic Motion (SHM)

Problem Definition

A mass m attached to a spring undergoes **simple harmonic motion (SHM)** when displaced from its equilibrium position. The restoring force is proportional to the displacement and is described by Hooke's Law:

$$F = -kx$$

where:

- k is the spring constant,
- x is the displacement from equilibrium.

Using Newton's second law $F = ma = m\frac{d^2x}{dt^2}$, the equation of motion becomes:

$$m\frac{d^2x}{dt^2} = -kx$$

Dividing through by m and letting $\omega^2 = \frac{k}{m}$, we get the ODE:

$$\frac{d^2x}{dt^2} + \omega^2 x = 0$$

This second-order ODE can be broken down into two first-order ODEs:

$$\frac{dx}{dt} = v, \quad \frac{dv}{dt} = -\omega^2 x$$

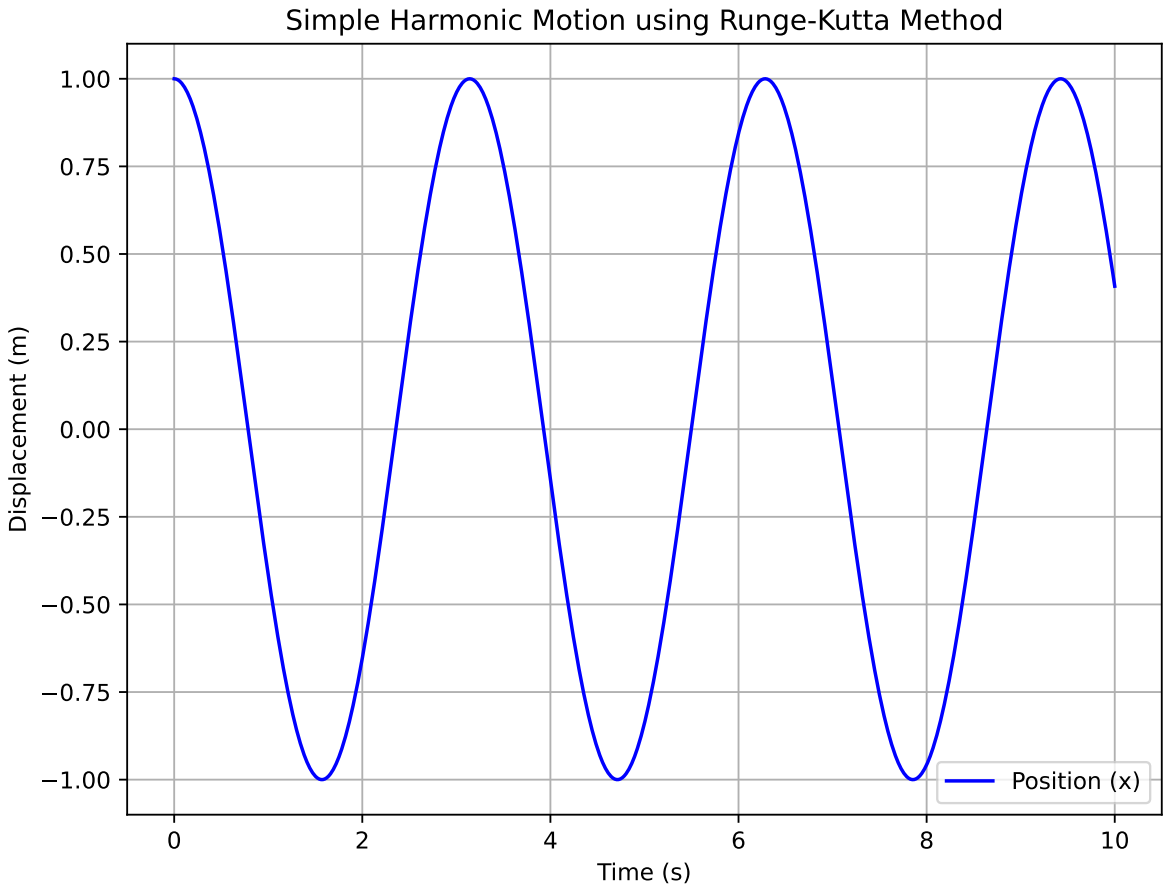
where v is the velocity of the mass.

Numerical Solution using the Runge-Kutta Method:

We will solve the ODEs using the **4th-order Runge-Kutta (RK4) method**.

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

# Define the ODE system for SHM
def shm(t, state, omega):
   # state = [x, v] where x is position and v is velocity
    x, v = state
    dxdt = v  # First ODE: dx/dt = v
    dvdt = -omega**2 * x  # Second ODE: dv/dt = -omega^2 * x
    return np.array([dxdt, dvdt])

# Runge-Kutta 4th order method for solving ODEs
def runge_kutta_4(f, state0, t, omega):
```

```
  # Array to store [x, v] values at each time step
    state = np.zeros((len(t), len(state0)))
    state[0] = state0  # Initial conditions [x0, v0]

    h = t[1] - t[0]  # Time step size
    for i in range(1, len(t)):
        k1 = h * f(t[i-1], state[i-1], omega)
        k2 = h * f(t[i-1] + h/2, state[i-1] + k1/2, omega)
        k3 = h * f(t[i-1] + h/2, state[i-1] + k2/2, omega)
        k4 = h * f(t[i], state[i-1] + k3, omega)

        state[i] = state[i-1] + (k1 + 2*k2 + 2*k3 + k4) / 6

    return state

# Parameters for SHM
omega = 2.0  # Angular frequency
x0 = 1.0     # Initial position
v0 = 0.0     # Initial velocity
state0 = [x0, v0]  # Initial state [position, velocity]
t = np.linspace(0, 10, 1000)  # Time array from 0 to 10 seconds

# Solve the ODE using Runge-Kutta
solution = runge_kutta_4(shm, state0, t, omega)
x_values = solution[:, 0]  # Extract position values

# Plot the results
plt.figure(figsize=(8, 6))
plt.plot(t, x_values, label='Position (x)', color='blue')
plt.title("Simple Harmonic Motion using Runge-Kutta Method")
plt.xlabel("Time (s)")
plt.ylabel("Displacement (m)")
plt.grid(True)
plt.legend()
plt.show()
```



Code Breakdown:

1. **Define the ODE system for SHM**:
   - The function `shm(t, state, omega)` defines the system of first-order ODEs for SHM. The state vector contains both the position x and velocity v, and the function returns their time derivatives:

$$\frac{dx}{dt} = v, \quad \frac{dv}{dt} = -\omega^2 x$$

2. **Runge-Kutta 4th Order Method**:
   - The `runge_kutta_4` function is an implementation of the RK4 method for solving ODEs. It computes the solution at each time step using the four stages k_1, k_2, k_3, k_4, where each stage is an intermediate estimate of the slope.

- The function takes in the ODE function `f`, the initial state `state0`, the time array `t`, and the angular frequency `omega`.
- The solution at each time step is stored in the `state` array.

3. **Parameters for SHM**:
   - The initial conditions for position ($x_0 = 1$) and velocity $v_0 = 0$ are set.
   - The angular frequency $\omega = 2.0$ is used.
   - A time array `t` is generated using `np.linspace` from 0 to 10 seconds.
4. **Solve the ODE**:
   - The `runge_kutta_4` function is called to compute the solution. The position values are extracted from the `solution` array for plotting.
5. **Plotting**:
   - The results are visualized with a plot of displacement x versus time t, showing the oscillatory nature of SHM.

Damped Harmonic Oscillator

Problem Definition:

The **damped harmonic oscillator** is a classic example in physics, particularly in mechanics. It describes a mass-spring system where the oscillation is subject to a damping force (e.g., friction or air resistance), which reduces the amplitude over time.

The equation of motion for the **damped harmonic oscillator** is given by the second-order differential equation:

$$\frac{d^2x}{dt^2} + 2\gamma\frac{dx}{dt} + \omega_0^2 x = 0$$

Where:

- $x(t)$ is the displacement of the mass at time t,
- γ is the **damping coefficient**, representing the amount of damping,
- ω_0 is the **natural angular frequency** of the undamped oscillator.

This equation can be rewritten as a system of **two first-order differential equations** by introducing the velocity $v = \frac{dx}{dt}$:

$$\frac{dx}{dt} = v$$

$$\frac{dv}{dt} = -2\gamma v - \omega_0^2 x$$

Mathematical Formulation:

1. Let $x(t)$ be the position, and $v(t)$ be the velocity of the system.
2. The system of equations becomes:

$$\frac{dx}{dt} = v$$

$$\frac{dv}{dt} = -2\gamma v - \omega_0^2 x$$

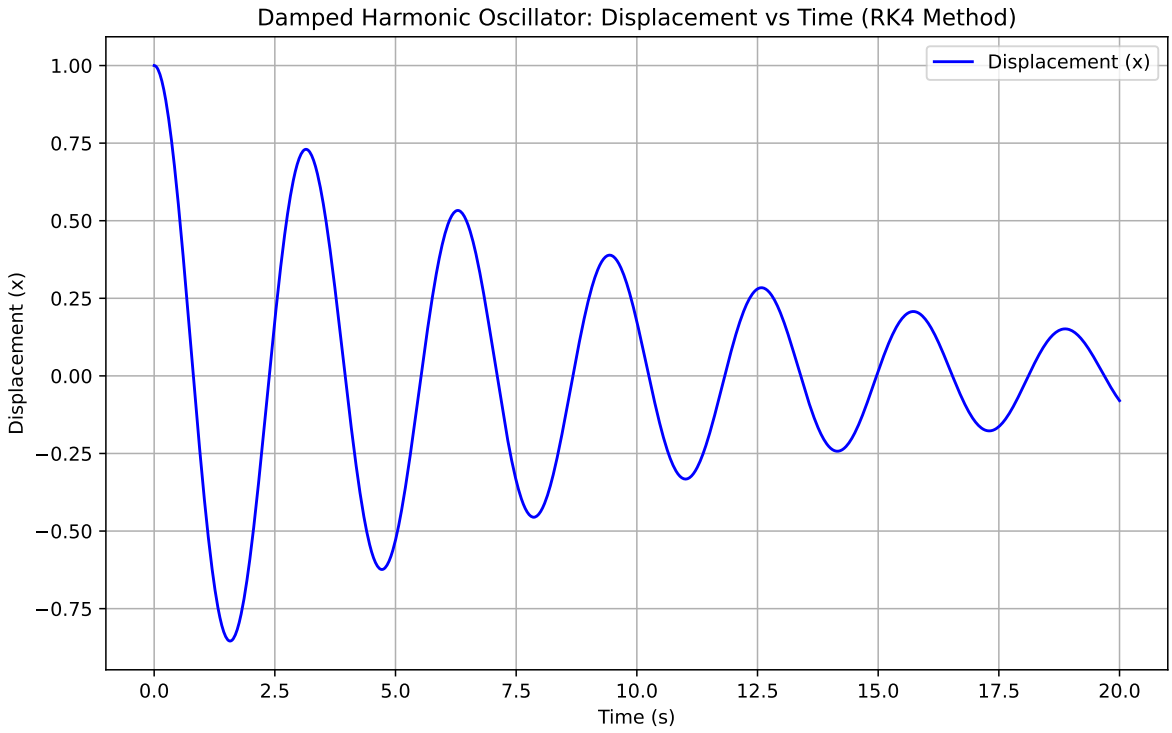
3. Initial conditions will be specified as:
   - $x(0) = x_0$: Initial displacement.
   - $v(0) = v_0$: Initial velocity.

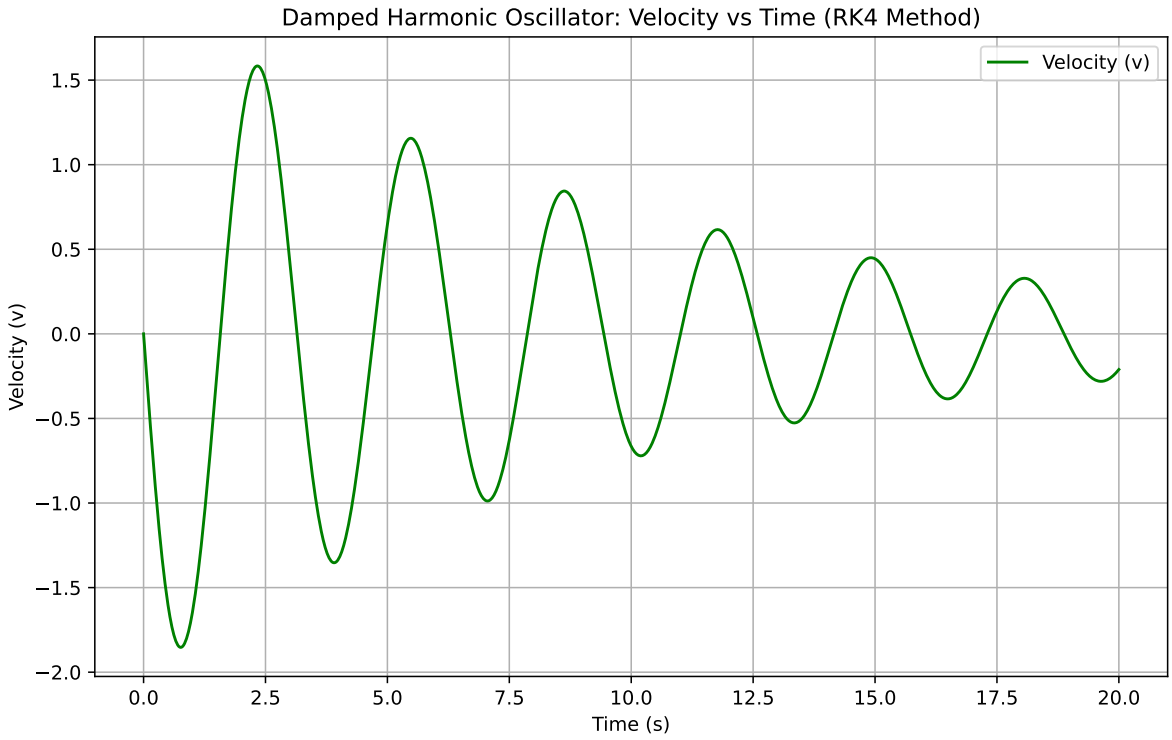
We will solve this system using the **Runge-Kutta 4th Order Method (RK4)**.

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

# Define the system of first-order ODEs for the damped harmonic oscillator
def damped_oscillator(state, t, gamma, omega0):
  # state = [x, v] where x is position and v is velocity
    x, v = state
    dxdt = v  # dx/dt = v
    dvdt = -2 * gamma * v - omega0**2 * x  # dv/dt = -2 v - ²x
    return np.array([dxdt, dvdt])

# Runge-Kutta 4th Order Method (RK4)
def runge_kutta_4(f, state0, t, gamma, omega0):
  # Array to store [x, v] values at each time step
```

```
    state = np.zeros((len(t), len(state0)))
    state[0] = state0  # Set initial conditions [x0, v0]
    h = t[1] - t[0]  # Step size (time step)

    # Iteratively apply RK4 for each time step
    for i in range(1, len(t)):
        k1 = h * f(state[i-1], t[i-1], gamma, omega0)
        k2 = h * f(state[i-1] + k1 / 2, t[i-1] + h / 2, gamma, omega0)
        k3 = h * f(state[i-1] + k2 / 2, t[i-1] + h / 2, gamma, omega0)
        k4 = h * f(state[i-1] + k3, t[i-1] + h, gamma, omega0)

        state[i] = state[i-1] + (k1 + 2 * k2 + 2 * k3 + k4) / 6

    return state

# Parameters for the damped harmonic oscillator
gamma = 0.1  # Damping coefficient
omega0 = 2.0  # Natural angular frequency
x0 = 1.0  # Initial displacement
v0 = 0.0  # Initial velocity
state0 = [x0, v0]  # Initial state [x0, v0]

# Time array: from 0 to 20 seconds, with 1000 points
t = np.linspace(0, 20, 1000)

# Solve the ODE using RK4
solution = runge_kutta_4(damped_oscillator, state0, t, gamma, omega0)
x_values = solution[:, 0]  # Extract position (x) values from the solution
v_values = solution[:, 1]  # Extract velocity (v) values from the solution

# Plot the displacement over time
plt.figure(figsize=(10, 6))
plt.plot(t, x_values, label="Displacement (x)", color="blue")
plt.title("Damped Harmonic Oscillator: Displacement vs Time (RK4 Method)")
plt.xlabel("Time (s)")
plt.ylabel("Displacement (x)")
plt.grid(True)
plt.legend()
plt.show()

# Plot the velocity over time
plt.figure(figsize=(10, 6))
```

```
plt.plot(t, v_values, label="Velocity (v)", color="green")
plt.title("Damped Harmonic Oscillator: Velocity vs Time (RK4 Method)")
plt.xlabel("Time (s)")
plt.ylabel("Velocity (v)")
plt.grid(True)
plt.legend()
plt.show()
```





Code Breakdown:

1. **ODE for Damped SHM**:
   - The function `damped_shm(t, state, omega, gamma)` defines the system of ODEs for damped harmonic motion. It includes the damping term $2\gamma v$, in addition to the standard harmonic motion equation.
2. **Solve with RK4**:
   - The same `runge_kutta_4` function is reused, with an additional parameter `gamma` for the damping coefficient.
3. **Plotting**:
   - The results show the damped oscillations, with the amplitude decaying over time.

The **Runge-Kutta Method** (RK4) to solve physical problems modeled by ordinary differential equations (ODEs). Specifically, we tackled two classical physics problems:

- The **Simple Harmonic Oscillator** modeled by a second-order ODE, which we reduced to two first-order ODEs and solved using RK4.

- The **Damped Harmonic Oscillator**, which included a frictional damping term, illustrating how real-world systems lose energy over time.

The use of numerical methods allows us to approximate solutions where exact analytical methods are not feasible, and the visualizations provide intuitive understanding of the system's behavior.

Radioactive Decay

Problem Definition

The rate at which a radioactive substance decays is proportional to the amount of the substance present. This can be modeled by the **radioactive decay equation**:

$$\frac{dN}{dt} = -\lambda N$$

where:

- $N(t)$ is the amount of the radioactive substance at time t,
- λ is the **decay constant** (characteristic of the substance).

The solution to this ODE is an exponential decay:

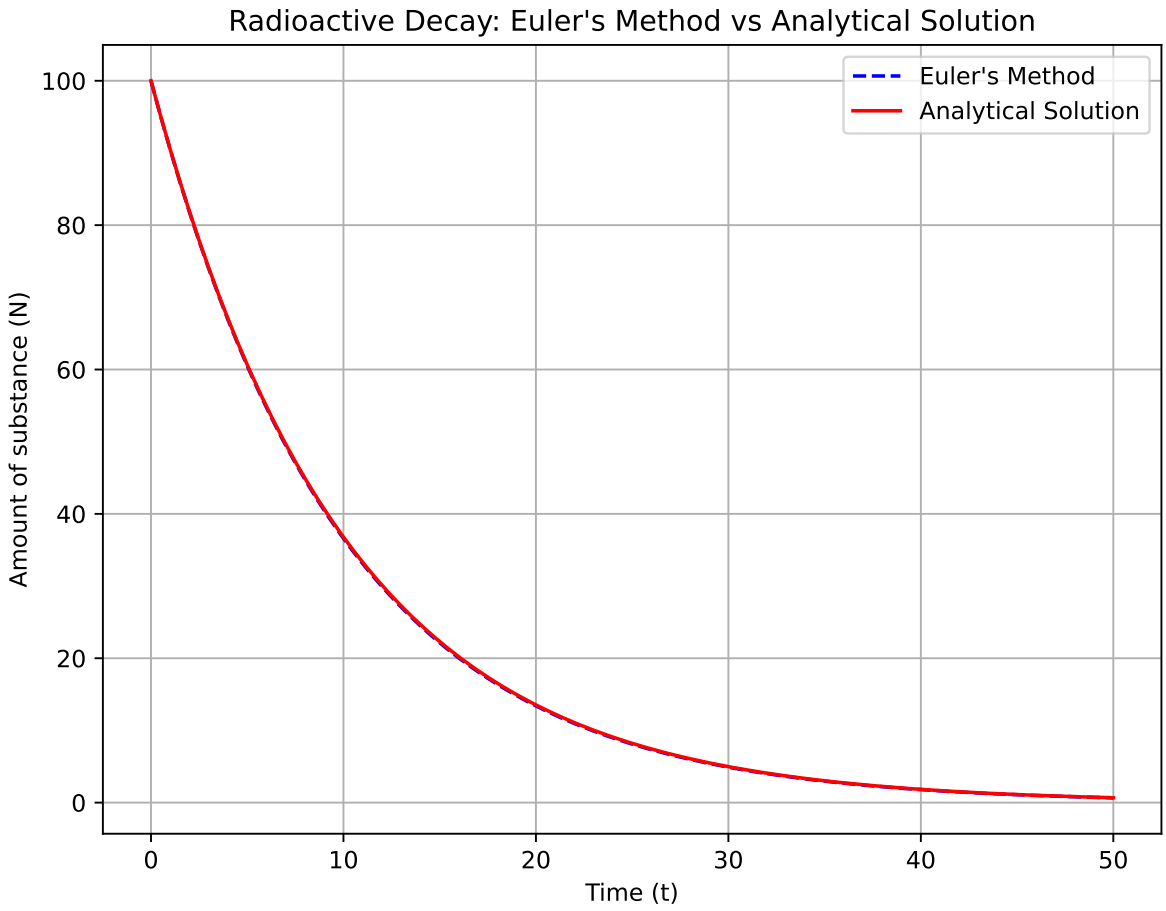
$$N(t) = N_0 e^{-\lambda t}$$

where N_0 is the initial amount of the substance at t. While we know the analytical solution, we will solve this problem using **Euler's method** to approximate $N(t)$ for a given decay constant and initial amount.

Objective

Use **Euler's method** to numerically solve the radioactive decay equation and compare the numerical solution to the analytical solution.

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the differential equation for radioactive decay
def radioactive_decay(N, t, decay_constant):
    return -decay_constant * N  # dN/dt = - N

# Euler's Method implementation
def euler_method(f, N0, t, decay_constant):
    N_values = np.zeros(len(t))  # Initialize array to store N values
    N_values[0] = N0  # Initial condition N(0) = N0

    h = t[1] - t[0]  # Time step size

    for i in range(1, len(t)):
        N_values[i] = N_values[i-1] + h * f(N_values[i-1],
                      t[i-1], decay_constant)

    return N_values

# Parameters
N0 = 100  # Initial amount of substance
decay_constant = 0.1  # Decay constant
t = np.linspace(0, 50, 500)  # Time from 0 to 50, with 500 points

# Solve the ODE using Euler's method
N_euler = euler_method(radioactive_decay, N0, t, decay_constant)

# Analytical solution for comparison
N_analytical = N0 * np.exp(-decay_constant * t)

# Plotting the results
plt.figure(figsize=(8, 6))
plt.plot(t, N_euler, label="Euler's Method", color='blue', linestyle='--')
plt.plot(t, N_analytical, label="Analytical Solution", color='red')
plt.title("Radioactive Decay: Euler's Method vs Analytical Solution")
plt.xlabel("Time (t)")
plt.ylabel("Amount of substance (N)")
plt.legend()
plt.grid(True)
plt.show()
```



Code Breakdown:

1. **Radioactive Decay ODE**:
   - The function `radioactive_decay(N, t, decay_constant)` defines the ODE $\frac{dN}{dt} = -\lambda N$. This models the rate of change of the radioactive substance over time, where is the decay constant.
2. **Euler's Method Implementation**:
   - The `euler_method(f, N0, t, decay_constant)` function implements Euler's method:
     - It initializes an array `N_values` to store the amount of substance at each time step.
     - The time step size `h` is determined from the time array `t`.

– In each iteration, the new value of NNN is computed using Euler's update formula:

$$N_{i+1} = N_i + h \cdot f(N_i, t_i, \lambda)$$

where $f(N, t, \lambda) = -\lambda N$ is the decay rate.

3. **Parameters**:
   - $N_0 = 100$: The initial amount of the radioactive substance.
   - $\lambda = 0.1$: The decay constant.
   - The time array `t` spans from 0 to 50 units of time, divided into 500 intervals.
4. **Comparison with Analytical Solution**:
   - We also compute the analytical solution $N(t) = N_0 e^{-\lambda t}$ for comparison.
5. **Plotting**:
   - The plot shows both the numerical solution from **Euler's method** and the exact **analytical solution**. The blue dashed line represents the Euler's method approximation, and the red line represents the analytical solution.

Discussion:

- **Motive**: To understand how numerical methods like Euler's method can approximate solutions to ODEs, even when analytical solutions are available for verification.
- **Key Learning**: Euler's method is simple and effective for solving first-order ODEs like radioactive decay, but it may introduce errors, particularly for large time steps. These errors can be visualized by comparing the Euler solution to the analytical solution.
- **Accuracy Consideration**: Euler's method is a first-order method, meaning the error decreases linearly with the time step hhh. Students can experiment by varying the time step to see how it affects the accuracy of the solution.

Motion Under Constant Force (Euler's Method)

Problem Definition:

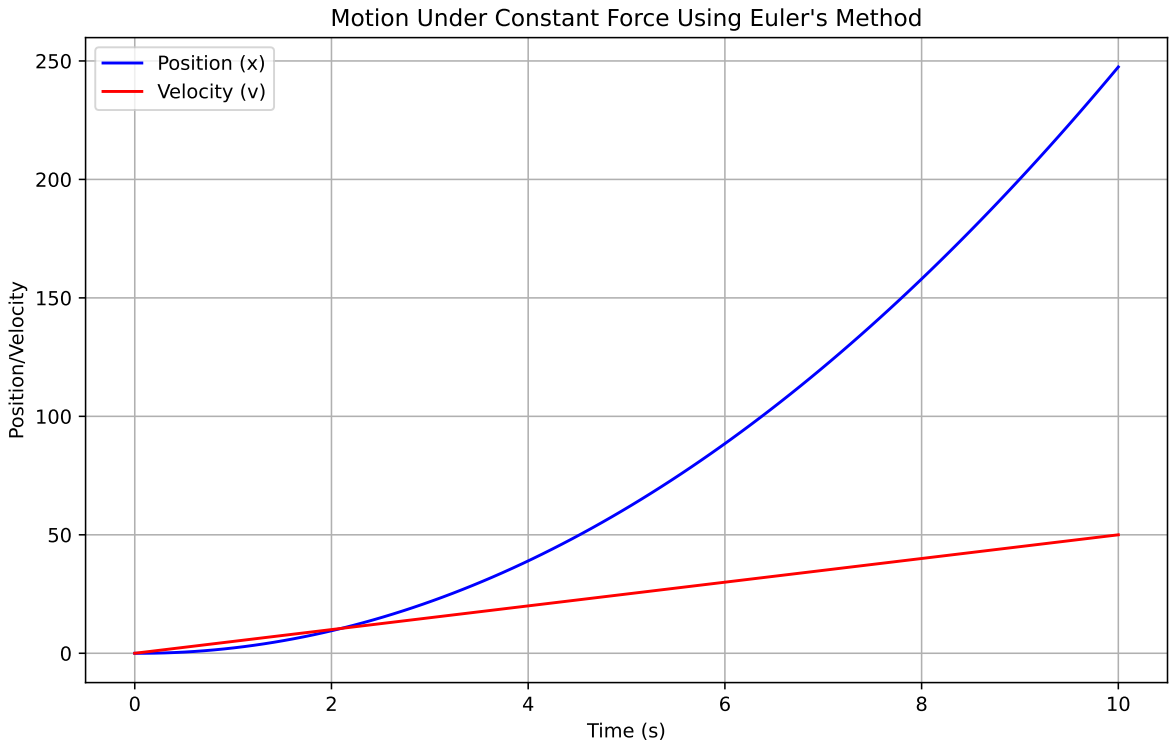
A particle of mass m is subjected to a constant force F. The equations are:

$$\frac{dx}{dt} = v, \quad \frac{dv}{dt} = \frac{F}{m}$$

We want to find the position and velocity of the particle over time using **Euler's method**.

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

# Define the ODE system for motion under constant force
def constant_force(state, t, F, m):
    x, v = state  # state = [x, v]
    dxdt = v
    dvdt = F / m
    return np.array([dxdt, dvdt])

# Euler's Method implementation
def euler_method(f, state0, t, F, m):
    state = np.zeros((len(t), len(state0)))
    state[0] = state0
    h = t[1] - t[0]

    for i in range(1, len(t)):
        state[i] = state[i-1] + h * f(state[i-1], t[i-1], F, m)

    return state

# Parameters
F = 10  # Force (N)
m = 2  # Mass (kg)
x0 = 0  # Initial position (m)
v0 = 0  # Initial velocity (m/s)
state0 = [x0, v0]  # Initial state [position, velocity]
t = np.linspace(0, 10, 100)  # Time from 0 to 10 seconds

# Solve the ODE using Euler's method
solution = euler_method(constant_force, state0, t, F, m)
x_values = solution[:, 0]  # Extract position values
```

```
v_values = solution[:, 1]  # Extract velocity values

# Plotting the results
plt.figure(figsize=(10, 6))
plt.plot(t, x_values, label="Position (x)", color='blue')
plt.plot(t, v_values, label="Velocity (v)", color='red')
plt.title("Motion Under Constant Force Using Euler's Method")
plt.xlabel("Time (s)")
plt.ylabel("Position/Velocity")
plt.grid(True)
plt.legend()
plt.show()
```



Explanation:

- We defined the ODE system where the position x is updated by velocity v, and velocity is updated by the force per unit mass.
- **Euler's method** is used to iteratively compute the new values of position and velocity over time.

- The plot shows both the position and velocity as a function of time.

Pendulum Without Damping (Runge-Kutta Method)

Problem Definition:

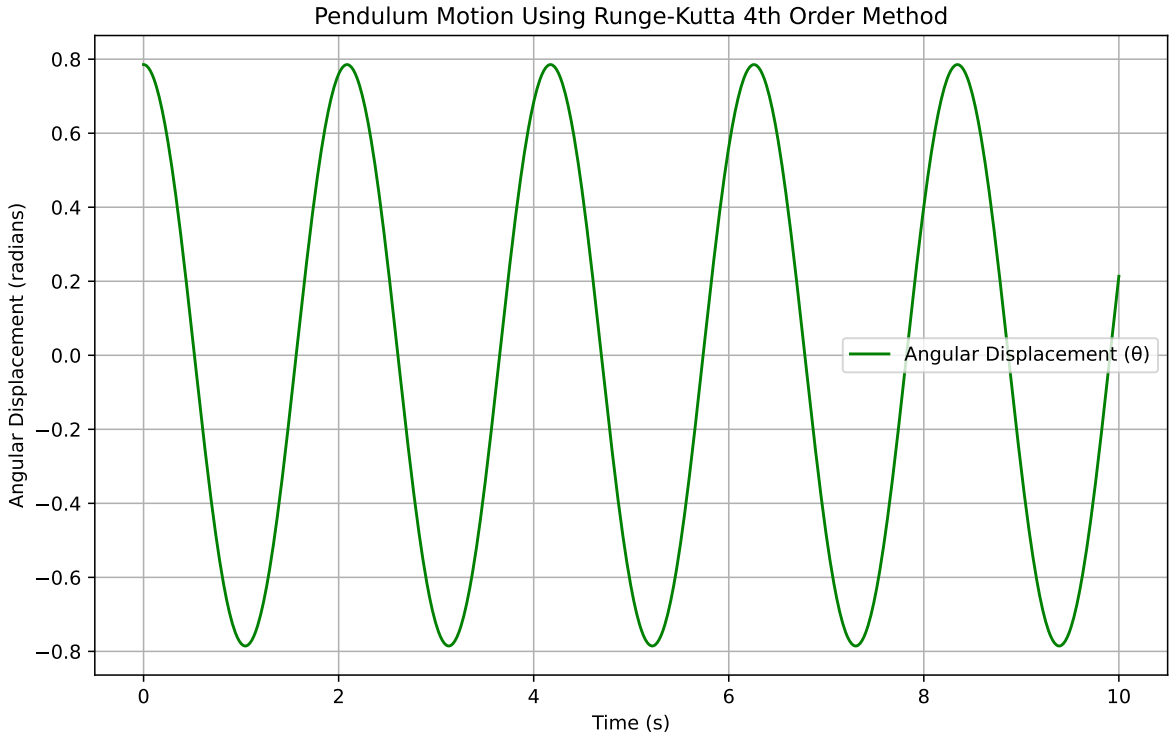
A simple pendulum's equation of motion is:

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}\sin(\theta) = 0$$

We'll solve this system using **Runge-Kutta 4th order (RK4)**.

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

# Define the ODE system for the pendulum
def pendulum(state, t, g, L):
    theta, omega = state  # state = [theta, omega]
    dtheta_dt = omega
    domega_dt = -(g / L) * np.sin(theta)
    return np.array([dtheta_dt, domega_dt])

# RK4 Method implementation
def runge_kutta_4(f, state0, t, g, L):
    state = np.zeros((len(t), len(state0)))
    state[0] = state0
    h = t[1] - t[0]

    for i in range(1, len(t)):
        k1 = h * f(state[i-1], t[i-1], g, L)
        k2 = h * f(state[i-1] + 0.5 * k1, t[i-1] + 0.5 * h, g, L)
        k3 = h * f(state[i-1] + 0.5 * k2, t[i-1] + 0.5 * h, g, L)
        k4 = h * f(state[i-1] + k3, t[i-1] + h, g, L)
        state[i] = state[i-1] + (k1 + 2*k2 + 2*k3 + k4) / 6

    return state

# Parameters
g = 9.81  # Acceleration due to gravity (m/s^2)
L = 1.0  # Length of pendulum (m)
theta0 = np.pi / 4  # Initial angle (radians)
```

```
omega0 = 0.0  # Initial angular velocity (rad/s)
state0 = [theta0, omega0]  # Initial state [angle, angular velocity]
t = np.linspace(0, 10, 1000)  # Time from 0 to 10 seconds

# Solve the ODE using RK4
solution = runge_kutta_4(pendulum, state0, t, g, L)
theta_values = solution[:, 0]  # Extract angular displacement values

# Plotting the results
plt.figure(figsize=(10, 6))
plt.plot(t, theta_values, label="Angular Displacement ( )", color='green')
plt.title("Pendulum Motion Using Runge-Kutta 4th Order Method")
plt.xlabel("Time (s)")
plt.ylabel("Angular Displacement (radians)")
plt.grid(True)
plt.legend()
plt.show()
```



Explanation:

- The pendulum's angular displacement θ and angular velocity ω are updated using the RK4 method.
- The RK4 method provides better accuracy than Euler's method, especially for systems with oscillations like the pendulum.
- The plot shows how the angle θ evolves over time.

Cooling of an Object Using Newton's Law of Cooling (Euler's Method)

Problem Definition

Newton's Law of Cooling states that the rate of change of the temperature of an object is proportional to the difference between its temperature and the ambient temperature:

$$\frac{dT}{dt} = -k(T - T_{ambient})$$

where:

- $T(t)$ is the temperature of the object at time t,
- $T_{ambient}$ is the ambient temperature,
- k is a constant that depends on the material and the surroundings.

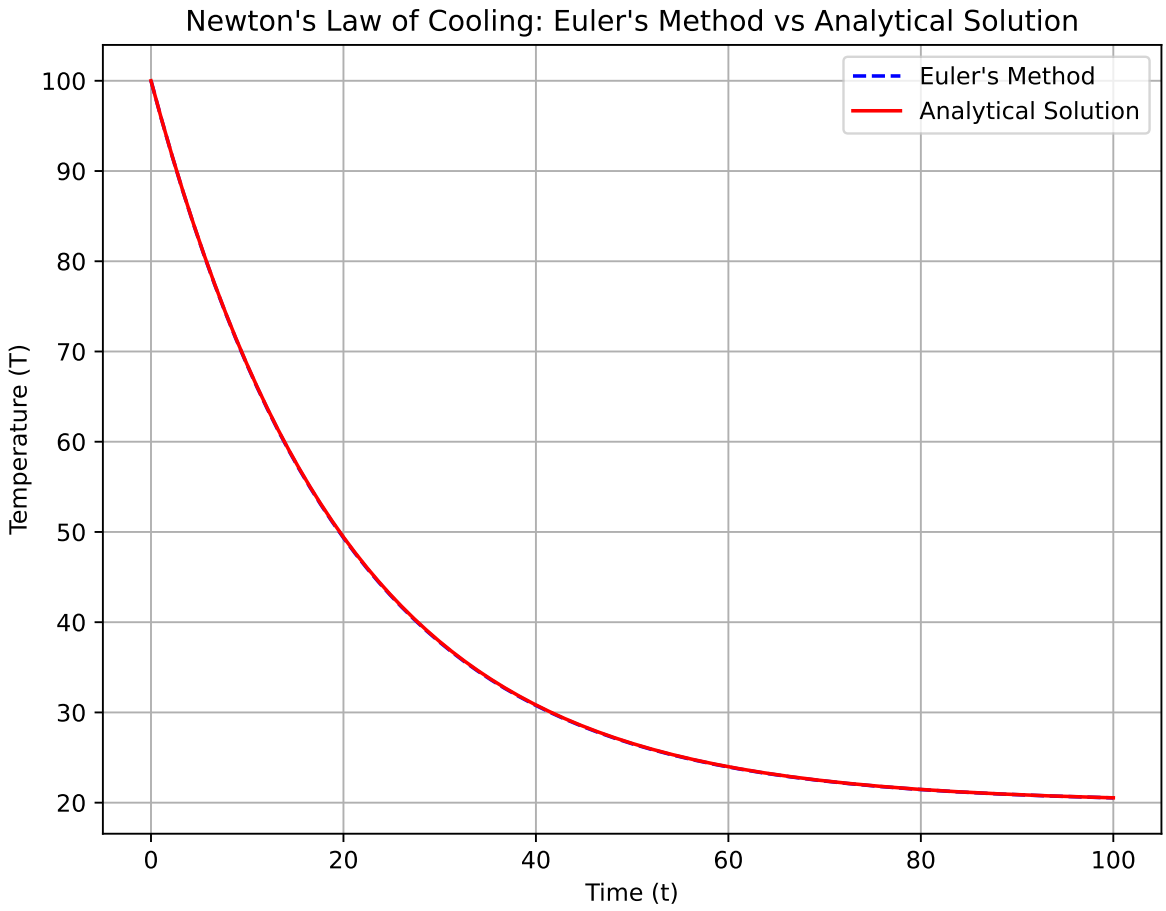
We will solve this ODE using **Euler's method**.

Objective

Use **Euler's method** to numerically solve Newton's Law of Cooling and study how the temperature of an object approaches the ambient temperature over time.

```
# Define the differential equation for Newton's law of cooling
def newtons_law_of_cooling(T, t, k, T_ambient):
    return -k * (T - T_ambient)  # dT/dt = -k(T - T_ambient)

# Euler's Method for cooling
def euler_method(f, T0, t, k, T_ambient):
    T_values = np.zeros(len(t))  # Initialize array to store T values
```

```
    T_values[0] = T0  # Initial temperature T(0) = T0

    h = t[1] - t[0]  # Time step size

    for i in range(1, len(t)):
        T_values[i] = T_values[i-1] + h * f(T_values[i-1], t[i-1], k, T_ambient)

    return T_values

# Parameters
T0 = 100  # Initial temperature of the object
T_ambient = 20  # Ambient temperature
k = 0.05  # Cooling constant
t = np.linspace(0, 100, 1000)  # Time from 0 to 100, with 1000 points

# Solve the ODE using Euler's method
T_euler = euler_method(newtons_law_of_cooling, T0, t, k, T_ambient)

# Analytical solution for comparison
T_analytical = T_ambient + (T0 - T_ambient) * np.exp(-k * t)

# Plotting the results
plt.figure(figsize=(8, 6))
plt.plot(t, T_euler, label="Euler's Method", color='blue', linestyle='--')
plt.plot(t, T_analytical, label="Analytical Solution", color='red')
plt.title("Newton's Law of Cooling: Euler's Method vs Analytical Solution")
plt.xlabel("Time (t)")
plt.ylabel("Temperature (T)")
plt.legend()
plt.grid(True)
plt.show()
```



Code Breakdown:

1. **Newton's Law of Cooling ODE**:
   - The function `newtons_law_of_cooling(T, t, k, T_ambient)` defines the ODE $\frac{dT}{dt} = -k(T - T_{ambient})$, where T is the temperature of the object and kkk is the cooling constant.
2. **Euler's Method for Cooling**:
   - The `euler_method` function solves the cooling problem using Euler's method. The function calculates the temperature T at each time step using the Euler update formula:

$$T_{i+1} = T_i + h \cdot f(T_i, t_i, k, T_{ambient})$$

3. **Parameters**:
    - $T_0 = 100$: The initial temperature of the object.
    - $T_{ambient} = 20$: The ambient temperature.
    - $k = 0.05$: The cooling constant.
4. **Comparison with Analytical Solution**:
    - The analytical solution for Newton's law of cooling is

$$T(t) = T_{ambient} + (T_0 - T_{ambient})e^{-kt}$$

      We compute this for comparison.
5. **Plotting**:
    - The plot shows the temperature as a function of time for both the Euler method and the analytical solution, illustrating how the object cools to the ambient temperature over time.

Understanding Nonlinear Dynamics and Chaos Using ODEs

Motivation

Nonlinear dynamics and chaos theory are fundamental areas in physics and applied mathematics that deal with complex systems whose behavior is highly sensitive to initial conditions. This sensitivity leads to unpredictability, even though the system is deterministic. Unlike linear systems, where small changes in initial conditions lead to small changes in the outcome, nonlinear systems exhibit disproportionate and sometimes chaotic behavior. Chaos is particularly fascinating because it reveals that deterministic systems can be unpredictable.

Fixed-point attractors, **limit cycles**, and **strange attractors** are central concepts in chaos theory. These are states to which a system tends to evolve, either stabilizing at a point, cycling through periodic patterns, or exhibiting complex, non-repeating chaotic behavior.

In this study material, we will explore nonlinear dynamics, chaos, and fixed-point attractors using ordinary differential equations (ODEs). We will focus on several key problems:

- **Logistic Map** (Discrete Chaos)
- **Lorenz Attractor** (Chaotic System)

Each problem will be solved using Python with detailed explanations of the code, and visualizations will help highlight the chaotic nature of the systems.

Logistic Map: Discrete Chaos

Problem Definition:

The **logistic map** is a simple, yet profound, model for population growth. It is represented by the recursion:

$$x_{n+1} = rx_n(1 - x_n)$$

where:

- x_n is the population at step nnn,
- r is a growth parameter.

For certain values of r, the system exhibits chaotic behavior, particularly when $r > 3.57$.

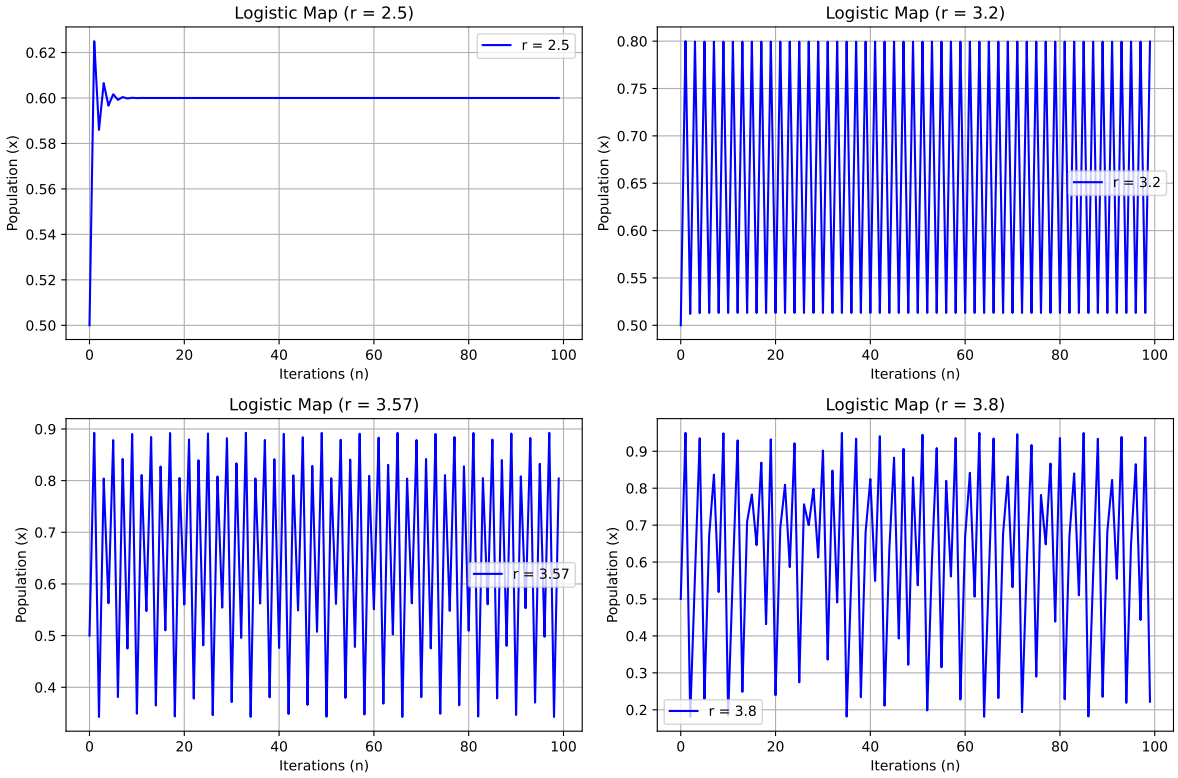
Objective:

Investigate the behavior of the logistic map for different values of r and observe the transition from stable fixed points to chaos.

Python Code for Logistic Map

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

# Logistic map function
def logistic_map(r, x, n):
    x_values = np.zeros(n)
    x_values[0] = x
    for i in range(1, n):
        x_values[i] = r * x_values[i-1] * (1 - x_values[i-1])
    return x_values

# Parameters
r_values = [2.5, 3.2, 3.57, 3.8]  # Different r values for comparison
x0 = 0.5  # Initial population
n = 100  # Number of iterations

# Create a 2x2 grid of subplots
```

```
fig, axs = plt.subplots(2, 2, figsize=(12, 8))

# Flatten the 2D axes array to iterate over it more easily
axs = axs.ravel()

# Plot the logistic map for different values of r
for i, r in enumerate(r_values):
    x_values = logistic_map(r, x0, n)
    axs[i].plot(range(n), x_values, label=f'r = {r}', color='blue')
    axs[i].set_title(f'Logistic Map (r = {r})')
    axs[i].set_xlabel("Iterations (n)")
    axs[i].set_ylabel("Population (x)")
    axs[i].grid(True)
    axs[i].legend()

# Adjust layout to prevent overlap
plt.tight_layout()

# Show the plot
plt.show()
```



Explanation:

1. **Logistic Map Function**:
   - This function iterates the logistic map equation $x_{n+1} = rx_n(1 - x_n)$ over `n` steps.
   - The initial population is set to $x_0 = 0.5$.
2. **Plot**:
   - We investigate the behavior of the map for several values of r and visualize the transition from fixed points to chaotic behavior.
   - For $r < 3$, the system stabilizes at a fixed point, while for $r > 3.57$, the system enters chaotic regimes with no repeating patterns.
   - To partition the plot into a matrix of panels, we can use `matplotlib`'s `plt.subplot` or `plt.subplots`. We've writes code to create a 2x2 grid of plots, each corresponding to a different value of r.
     - We used `plt.subplots(2, 2)` to create a 2x2 grid of subplots.

- The `axs` object is an array of the subplot axes, which is then flattened with `axs.ravel()` for easy iteration.
- Each plot is created in a separate panel, corresponding to a different value of r.
- `plt.tight_layout()` ensures that the subplots are spaced nicely without overlapping.

Key Learning:

The logistic map demonstrates how a simple nonlinear recursion can lead to chaotic behavior. Small changes in the parameter r lead to vastly different outcomes, showcasing the sensitivity of nonlinear systems.

Lorenz Attractor: A Classic Chaotic System

Problem Definition:

The **Lorenz system** is a set of three nonlinear differential equations that model atmospheric convection:

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

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

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

with the initial conditions : $x(0) = y(0) = z(0) = 1$

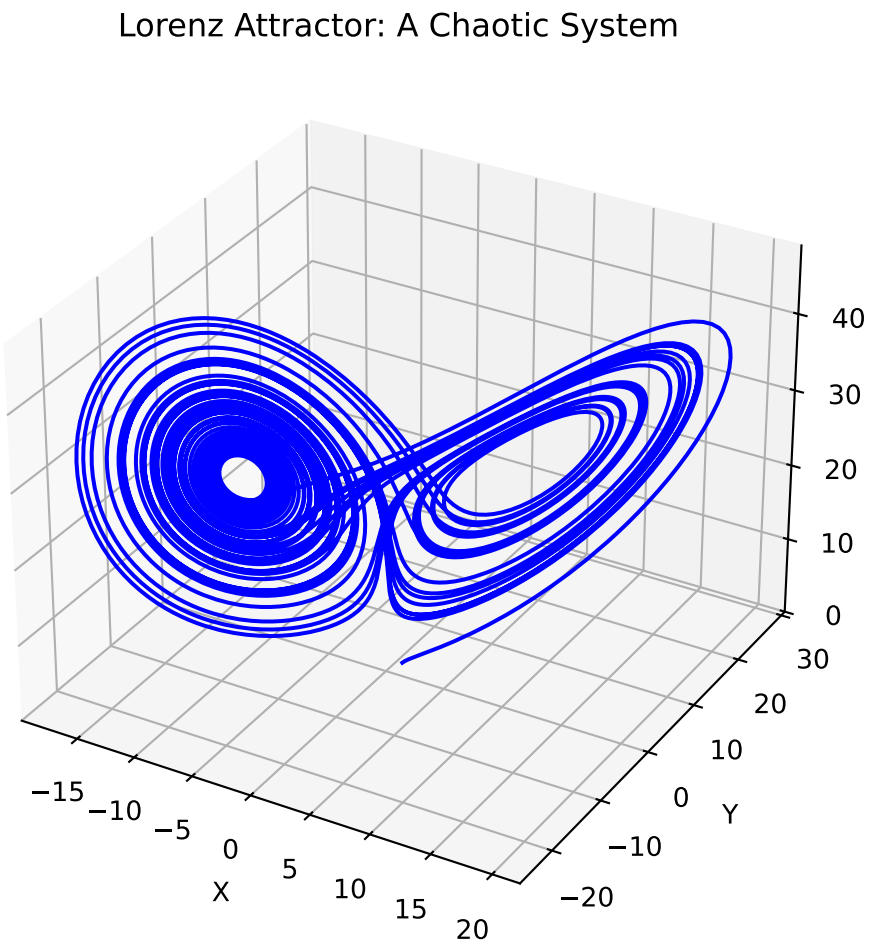
Objective

Solve the Lorenz system using the **Runge-Kutta 4th order method (RK4)** and observe the chaotic behavior for certain parameter values.

Python Code for Lorenz Attractor

```
from mpl_toolkits.mplot3d import Axes3D

# Lorenz system ODEs
def lorenz_system(state, t, sigma, rho, beta):
    x, y, z = state
    dxdt = sigma * (y - x)
    dydt = x * (rho - z) - y
    dzdt = x * y - beta * z
    return np.array([dxdt, dydt, dzdt])

# RK4 method for Lorenz system
def runge_kutta_4(f, state0, t, sigma, rho, beta):
    state = np.zeros((len(t), len(state0)))
    state[0] = state0
    h = t[1] - t[0]

    for i in range(1, len(t)):
        k1 = h * f(state[i-1], t[i-1], sigma, rho, beta)
        k2 = h * f(state[i-1] + 0.5 * k1, t[i-1] + 0.5 * h, sigma, rho, beta)
        k3 = h * f(state[i-1] + 0.5 * k2, t[i-1] + 0.5 * h, sigma, rho, beta)
        k4 = h * f(state[i-1] + k3, t[i-1] + h, sigma, rho, beta)
        state[i] = state[i-1] + (k1 + 2*k2 + 2*k3 + k4) / 6

    return state

# Parameters for Lorenz system
sigma = 10.0
rho = 28.0
beta = 8.0 / 3.0
state0 = [1.0, 1.0, 1.0]  # Initial condition [x0, y0, z0]
t = np.linspace(0, 50, 10000)  # Time array

# Solve Lorenz system using RK4
solution = runge_kutta_4(lorenz_system, state0, t, sigma, rho, beta)
x_values = solution[:, 0]
y_values = solution[:, 1]
z_values = solution[:, 2]

# Plotting Lorenz attractor in 3D
fig = plt.figure(figsize=(10, 6))
```

```
ax = fig.add_subplot(111, projection='3d')
ax.plot(x_values, y_values, z_values, color='blue')
ax.set_title("Lorenz Attractor: A Chaotic System")
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
plt.show()
```



Explanation:

1. **Lorenz System ODEs**:
    - The three equations represent the Lorenz system, a simplified model for atmospheric convection.

- These equations are highly sensitive to initial conditions, resulting in chaotic behavior for certain parameter values.

2. **Runge-Kutta 4th Order Method (RK4)**:
   - The RK4 method is used to solve the system of equations numerically.
   - The state vector consists of three variables: x, y, and z, representing the system's evolution over time.

3. **3D Plot**:
   - The resulting Lorenz attractor is a strange attractor, exhibiting chaotic behavior with complex, non-repeating trajectories.
   - Even though the system is deterministic, small differences in initial conditions lead to vastly different outcomes.

Practice Problem Set

1. Damped Driven Pendulum

Problem Definition:

The **damped driven pendulum** is an example of a **nonlinear dynamical system**. The equation of motion for a damped driven pendulum is given by:

$$\frac{d^2\theta}{dt^2} + 2\gamma\frac{d\theta}{dt} + \omega_0^2 \sin(\theta) = A\cos(\omega t)$$

where:

- θ is the angle of displacement,
- γ is the damping coefficient,
- ω_0 is the natural frequency,
- A is the amplitude of the driving force,
- ω is the driving frequency.

Solve the damped driven pendulum using the **RK4 method**.

2. Heat Transfer in a Rod (Euler's Method)

Problem Definition:

The temperature distribution along a thin rod can be described by the heat equation in one dimension:

$$\frac{dT}{dt} = \alpha \frac{d^2T}{dx^2}$$

where $T(x,t)$ is the temperature, and α is the thermal diffusivity.

Objective:

Solve for the temperature distribution over time using **Euler's method**.

3. Motion of a Falling Object with Air Resistance (Euler's Method)

Problem Definition:

A falling object experiences air resistance proportional to its velocity. The equation of motion is:

$$\frac{dv}{dt} = g - kv$$

where $v(t)$ is the velocity, g is the acceleration due to gravity, and k is the air resistance constant.

Objective:

Use **Euler's method** to solve for the velocity of the object over time.

4. Population Growth with Carrying Capacity (Euler's Method)

Problem Definition:

The **logistic growth equation** describes population growth with a limiting carrying capacity:

$$\frac{dP}{dt} = rP\left(1 - \frac{P}{K}\right)$$

where $P(t)$ is the population, r is the growth rate, and K is the carrying capacity.

Objective:

Use **Euler's method** to approximate the population over time.

5. RLC Circuit (Runge-Kutta Method)

Problem Definition:

An RLC circuit consists of a resistor, inductor, and capacitor in series. The equation governing the charge on the capacitor is:

$$\frac{d^2Q}{dt^2} + R\frac{dQ}{dt} + \frac{1}{C}Q = V(t)$$

where L is the inductance, R is the resistance, C is the capacitance, and $V(t)$ is the input voltage.

Objective:

Solve for the charge on the capacitor $Q(t)$ using the **Runge-Kutta method (RK4)**.

Conclusion

In this guide, we have expanded the exploration of **ODEs in physics.** These examples illustrate the power of **numerical methods** such as the **Euler's Method & Runge-Kutta method** in solving complex ODEs that arise in real-world physics. Each problem demonstrates the interplay between physical laws and mathematical models, emphasizing the importance of computational tools in modern physics. Also, using numerical methods like **Runge-Kutta Method (RK4)**, we were able to solve nonlinear ODEs and visualize the chaotic behavior that emerges from deterministic systems. This study material highlights the importance of understanding chaos and nonlinear dynamics in various physical systems.