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

Problem 1, (1) Gaussian quadrature

```
# Define the function
def f(x):
    return 4*x**3 - (x + 3)**(-2) - 2*x + 5

# Gaussian Quadrature method
result_gauss, error_gauss = integrate.quadrature(f, -2, 1)
print(f"Gaussian Quadrature result: {result_gauss}")
```

```
Gaussian Quadrature result: 2.2500000007464926

C:\Users\jhyang\AppData\Local\Temp\ipykernel_25052\2804665587.py:6:
DeprecationWarning: `scipy.integrate.quadrature` is deprecated as of
SciPy 1.12.0and will be removed in SciPy 1.15.0. Please
use`scipy.integrate.quad` instead.
  result_gauss, error_gauss = integrate.quadrature(f, -2, 1)
```

```
# Define the function
def f(x):
    return 4*x**3 - (x + 3)**(-2) - 2*x + 5

# Compute the true value using quad (very accurate)
true_value, error = integrate.quad(f, -2, 1)
print(f"True value: {true_value}")
```

```
True value: 2.249999999999999
```

```
# Calculate absolute error
error = abs(true_value - result_gauss)
print(f"Absolute Error: {error}")
print("It is close!!!")
```

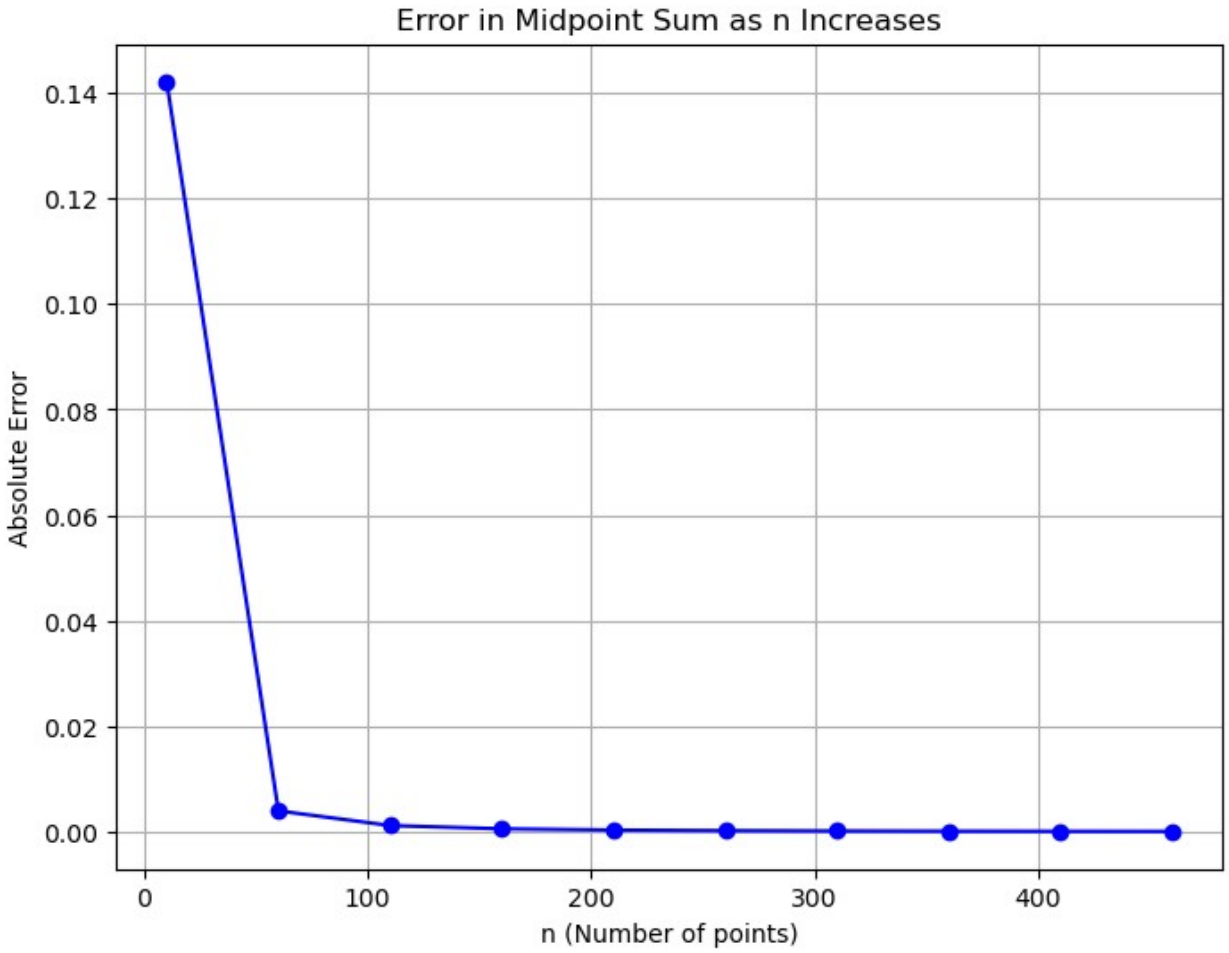
```
Absolute Error: 7.464935336543022e-10
It is close!!!
```

Problem 1, (2) Midpoint sum

```
# Define the function
def f(x):
    return 4*x**3 - (x + 3)**(-2) - 2*x + 5

# Midpoint sum method
def midpoint_sum(f, a, b, n):
    h = (b - a) / n
    midpoints = np.linspace(a + h/2, b - h/2, n)
    return h * np.sum(f(midpoints))
```

```python
# Compute the true value using scipy.integrate.quad
true_value, _ = integrate.quad(f, -2, 1)

# Define a range of n values
n_values = np.arange(10, 510, 50)

# Lists to store the results and errors
midpoint_results = []
midpoint_errors = []

# Compute midpoint sum for different n values and calculate errors
for n in n_values:
    result_midpoint = midpoint_sum(f, -2, 1, n)
    midpoint_results.append(result_midpoint)
    error = np.abs(result_midpoint - true_value)
    midpoint_errors.append(error)
    print(f"n = {n}, Midpoint Sum Result = {result_midpoint}, Absolute
Error = {error}")

# Plot the error vs n
plt.figure(figsize=(8, 6))
plt.plot(n_values, midpoint_errors, marker='o', linestyle='-',
color='b')
plt.title('Error in Midpoint Sum as n Increases')
plt.xlabel('n (Number of points)')
plt.ylabel('Absolute Error')
plt.grid(True)
plt.show()
```

```
n = 10, Midpoint Sum Result = 2.3921617861870783, Absolute Error =
0.1421617861870792
n = 60, Midpoint Sum Result = 2.253954896370443, Absolute Error =
0.003954896370443883
n = 110, Midpoint Sum Result = 2.251176701347723, Absolute Error =
0.001176701347723963
n = 160, Midpoint Sum Result = 2.250556179260962, Absolute Error =
0.0005561792609629634
n = 210, Midpoint Sum Result = 2.2503228623070184, Absolute Error =
0.0003228623070192782
n = 260, Midpoint Sum Result = 2.2502106249458516, Absolute Error =
0.0002106249458524445
n = 310, Midpoint Sum Result = 2.2501481608396694, Absolute Error =
0.0001481608396704604
n = 360, Midpoint Sum Result = 2.250109863140737, Absolute Error =
0.0001098631407376871
n = 410, Midpoint Sum Result = 2.2500847011731717, Absolute Error =
8.470117317260772e-05
n = 460, Midpoint Sum Result = 2.250067288611044, Absolute Error =
6.728861104488004e-05
```

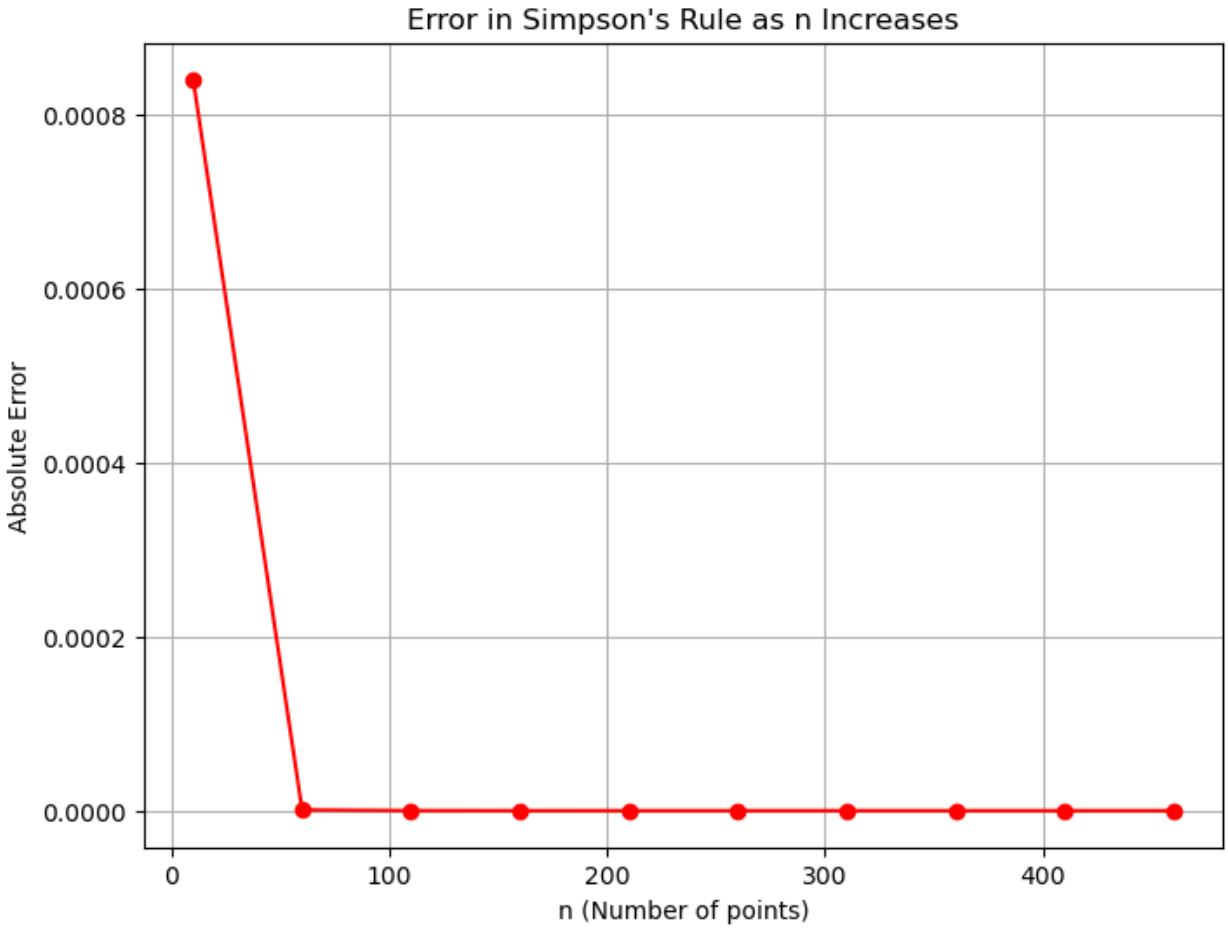


Problem 1, (3) Simpson sum

```
# Define the function
def f(x):
    return 4*x**3 - (x + 3)**(-2) - 2*x + 5

# Simpson's rule method
def simpson_sum(f, a, b, n):
    if n % 2 == 1:  # n must be even
        n += 1
    h = (b - a) / n
    x = np.linspace(a, b, n + 1)
    y = f(x)
    return h / 3 * (y[0] + y[-1] + 4 * np.sum(y[1:-1:2]) + 2 *
np.sum(y[2:-2:2]))

# Compute the true value using quad
true_value, _ = integrate.quad(f, -2, 1)

# Range of n values
n_values = np.arange(10, 510, 50)
```

```
# Lists to store results and errors for Simpson's Rule
simpson_results = []
simpson_errors = []

# Compute Simpson's rule for different n values and calculate errors
for n in n_values:
    result_simpson = simpson_sum(f, -2, 1, n)
    simpson_results.append(result_simpson)
    error = np.abs(result_simpson - true_value)
    simpson_errors.append(error)
    print(f"n = {n}, Simpson's Rule Result = {result_simpson},
Absolute Error = {error}")

# Plotting error vs n for Simpson's rule
plt.figure(figsize=(8, 6))
plt.plot(n_values, simpson_errors, marker='o', linestyle='-',
color='r')
plt.title('Error in Simpson\'s Rule as n Increases')
plt.xlabel('n (Number of points)')
plt.ylabel('Absolute Error')
plt.grid(True)
plt.show()
```

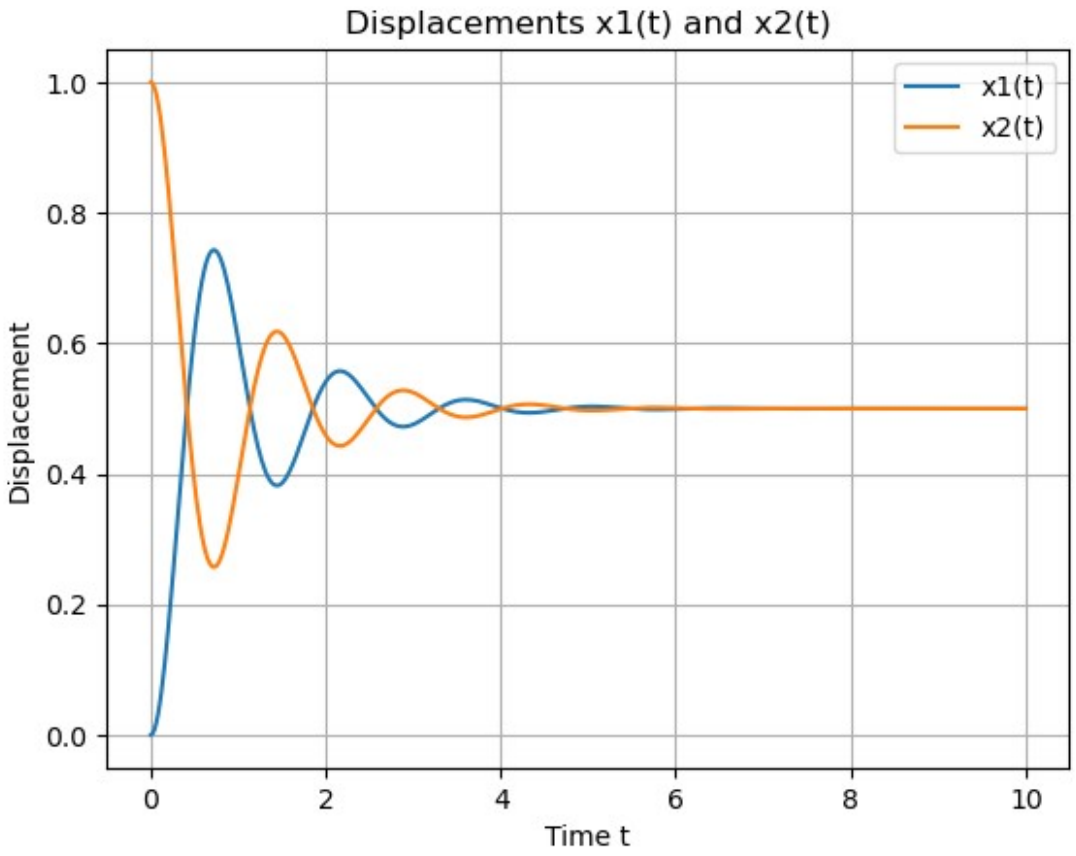
```
n = 10, Simpson's Rule Result = 2.249160012524373, Absolute Error =
0.0008399874756261916
n = 60, Simpson's Rule Result = 2.249999174813552, Absolute Error =
8.251864471731096e-07
n = 110, Simpson's Rule Result = 2.2499999265016717, Absolute Error =
7.349832742420404e-08
n = 160, Simpson's Rule Result = 2.249999983557248, Absolute Error =
1.644275116063909e-08
n = 210, Simpson's Rule Result = 2.249999994456223, Absolute Error =
5.5437761048438006e-09
n = 260, Simpson's Rule Result = 2.249999997640068, Absolute Error =
2.359930917350539e-09
n = 310, Simpson's Rule Result = 2.249999998832096, Absolute Error =
1.167903107557322e-09
n = 360, Simpson's Rule Result = 2.2499999993577826, Absolute Error =
6.422165022001991e-10
n = 410, Simpson's Rule Result = 2.249999999618247, Absolute Error =
3.817519633741995e-10
n = 460, Simpson's Rule Result = 2.249999999759062, Absolute Error =
2.4093704809047267e-10
```


Error in Simpson's Rule as n Increases
Absolute Error
0.0008
0.0006
0.0004
0.0002
0.0000
0
100
200
300
400
n (Number of points)

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
```

Problem 2, (1): Solve for x1(t) and x2(t)

```
# Parameters
m = 1
k = 10
c = 1

# Define the system of equations
def equations(t, y):
    x1, v1, x2, v2 = y
    dx1dt = v1
    dx2dt = v2
    dv1dt = (c * (v2 - v1) + k * (x2 - x1)) / m
    dv2dt = (c * (v1 - v2) + k * (x1 - x2)) / m
    return [dx1dt, dv1dt, dx2dt, dv2dt]

# Initial conditions: x1(0) = 0, v1(0) = 0, x2(0) = 1, v2(0) = 0
initial_conditions = [0, 0, 1, 0]

# Time span for the solution
t_span = (0, 10)
t_eval = np.linspace(0, 10, 500)

# Solve the system
solution = solve_ivp(equations, t_span, initial_conditions,
t_eval=t_eval)

# Plot the results
plt.plot(solution.t, solution.y[0], label='x1(t)')
plt.plot(solution.t, solution.y[2], label='x2(t)')
plt.xlabel('Time t')
plt.ylabel('Displacement')
plt.title('Displacements x1(t) and x2(t)')
plt.legend()
plt.grid(True)
plt.show()
```


Displacements x1(t) and x2(t)
x1(t)
x2(t)
Displacement
Time t
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10

Problem 2, (2): Transforming the System into First-Order Form

Given the system of two masses, springs, and dampers:

$$m\,\ddot{x}_1(t)+c\,\dot{x}_1(t)+k\,x_1(t)=c\,\dot{x}_2(t)+k\,x_2(t)$$

$$m\,\ddot{x}_2(t)+c\,\dot{x}_2(t)+k\,x_2(t)=c\,\dot{x}_1(t)+k\,x_1(t)$$

With the initial conditions:

$$x_1(0)=0,\dot{x}_1(0)=0,x_2(0)=1,\dot{x}_2(0)=0$$

And the parameters: $m=1,k=10,c=1$

Step 1: Simplifying the Equations

Given the parameters $m=1,k=10,c=1$, the system simplifies to:

$$\ddot{x}_1(t)+\dot{x}_1(t)+10\,x_1(t)=\dot{x}_2(t)+10\,x_2(t)$$

$$\ddot{x}_2(t)+\dot{x}_2(t)+10\,x_2(t)=\dot{x}_1(t)+10\,x_1(t)$$

Step 2: Introducing New Variables

Let:

$$y_1(t)=\dot{x}_1(t)$$

$$y_2(t)=\dot{x}_2(t)$$

Then the equations become:

$$\dot{y}_1(t)=\ddot{x}_1(t)$$

$$\dot{y}_2(t)=\ddot{x}_2(t)$$

Step 3: Writing the System as a Matrix Equation

From the first equation:

$$\dot{y}_1+y_1+10\,x_1=y_2+10\,x_2$$

Rearranging:

$$\dot{y}_1=y_2+10\,x_2-y_1-10\,x_1$$

From the second equation:

$$\dot{y}_2+y_2+10\,x_2=y_1+10\,x_1$$

Rearranging:

$\dot{y}_2 = y_1 + 10\,x_1 - y_2 - 10\,x_2$

Thus, we have the following first-order system:

$\dot{x}_1 = y_1$

$\dot{y}_1 = y_2 + 10\,x_2 - y_1 - 10\,x_1$

$\dot{x}_2 = y_2$

$\dot{y}_2 = y_1 + 10\,x_1 - y_2 - 10\,x_2$

Step 4: Matrix Representation

We can express the system as a matrix equation:

$$\begin{bmatrix} \dot{x}_1(t) \\ \dot{y}_1(t) \\ \dot{x}_2(t) \\ \dot{y}_2(t) \end{bmatrix} = \mathbf{B} \begin{bmatrix} x_1(t) \\ y_1(t) \\ x_2(t) \\ y_2(t) \end{bmatrix}$$

Where B is a matrix that describes the system dynamics:

$$\mathbf{B} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -10 & -1 & 10 & 1 \\ 0 & 0 & 0 & 1 \\ 10 & 1 & -10 & -1 \end{bmatrix}$$

This is the first-order form of the system.

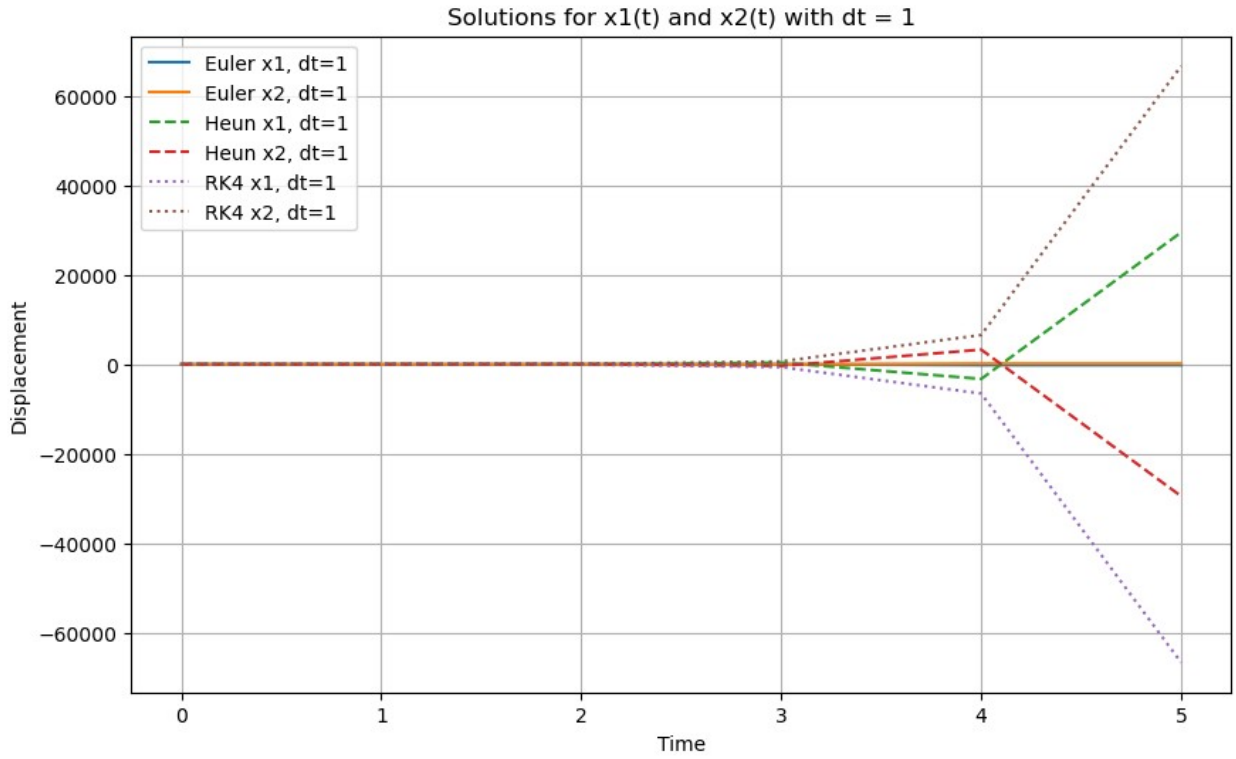
```
import numpy as np
import matplotlib.pyplot as plt
```

Problem 2, (3): uses different numerical methods (Euler, Heun's method, and 4th-order Runge-Kutta) to solve the system of equations

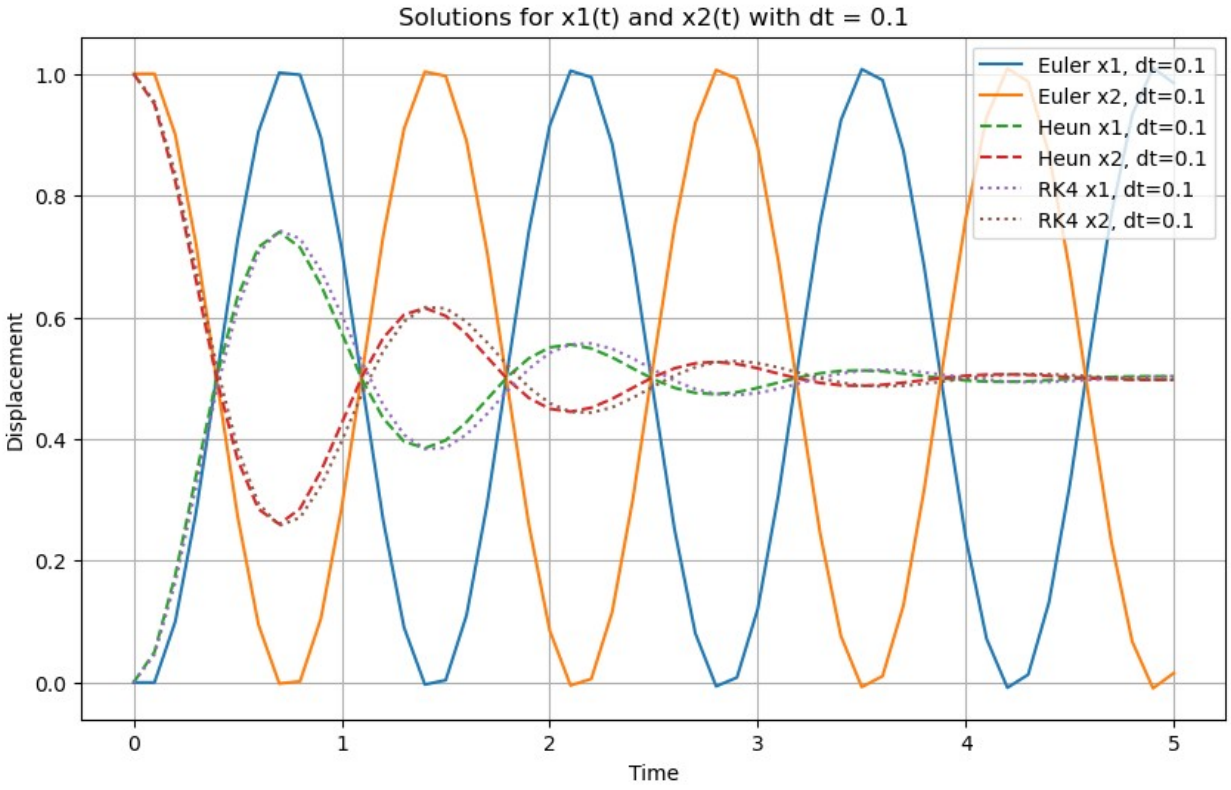
```
# Define the matrix B
B = np.array([
    [0, 1, 0, 0],
    [-10, -1, 10, 1],
    [0, 0, 0, 1],
    [10, 1, -10, -1]
])

# Initial conditions
X0 = np.array([0, 0, 1, 0])

# Time settings
T = 5
time_steps = [10**0, 10**-1, 10**-2, 10**-3, 10**-4]

# Assume the true solution is unknown

# Function to compute the derivative dX/dt = B * X
def derivative(X):
    return B @ X

# Euler method
def euler_method(X0, dt, T):
    num_steps = int(T / dt)
    X = X0.copy()
    solution = [X]
    for _ in range(num_steps):
        X = X + dt * derivative(X)
        solution.append(X)
    return np.array(solution)

# Heun's method
def heun_method(X0, dt, T):
    num_steps = int(T / dt)
    X = X0.copy()
    solution = [X]
    for _ in range(num_steps):
        k1 = derivative(X)
        k2 = derivative(X + dt * k1)
        X = X + dt * 0.5 * (k1 + k2)
        solution.append(X)
    return np.array(solution)

# 4th-order Runge-Kutta method
```

```
def rk4_method(X0, dt, T):
    num_steps = int(T / dt)
    X = X0.copy()
    solution = [X]
    for _ in range(num_steps):
        k1 = derivative(X)
        k2 = derivative(X + 0.5 * dt * k1)
        k3 = derivative(X + 0.5 * dt * k2)
        k4 = derivative(X + dt * k3)
        X = X + (dt / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
        solution.append(X)
    return np.array(solution)

# Function to calculate global error for a given method
def calculate_global_error(true_solution, numerical_solution):
    return np.linalg.norm(true_solution - numerical_solution, ord=2)

# Simulate and compute errors for each method and time step
for dt in time_steps:
    euler_solution = euler_method(X0, dt, T)
    heun_solution = heun_method(X0, dt, T)
    rk4_solution = rk4_method(X0, dt, T)

    # Time vector for plotting
    time = np.linspace(0, T, len(euler_solution))

    # Plot x1 and x2 for each method
    plt.figure(figsize=(10, 6))
    plt.plot(time, euler_solution[:, 0], label=f'Euler x1, dt={dt}')
    plt.plot(time, euler_solution[:, 2], label=f'Euler x2, dt={dt}')
    plt.plot(time, heun_solution[:, 0], '--', label=f'Heun x1,
dt={dt}')
    plt.plot(time, heun_solution[:, 2], '--', label=f'Heun x2,
dt={dt}')
    plt.plot(time, rk4_solution[:, 0], ':', label=f'RK4 x1, dt={dt}')
    plt.plot(time, rk4_solution[:, 2], ':', label=f'RK4 x2, dt={dt}')
    plt.xlabel('Time')
    plt.ylabel('Displacement')
    plt.title(f'Solutions for x1(t) and x2(t) with dt = {dt}')
    plt.legend()
    plt.grid()
    plt.show()

    # Assuming the RK4 solution with smallest dt is the most accurate,
using it as "true" solution
    true_solution = rk4_method(X0, 10**-4, T)
    euler_error = calculate_global_error(true_solution[-1],
euler_solution[-1])
    heun_error = calculate_global_error(true_solution[-1],
```

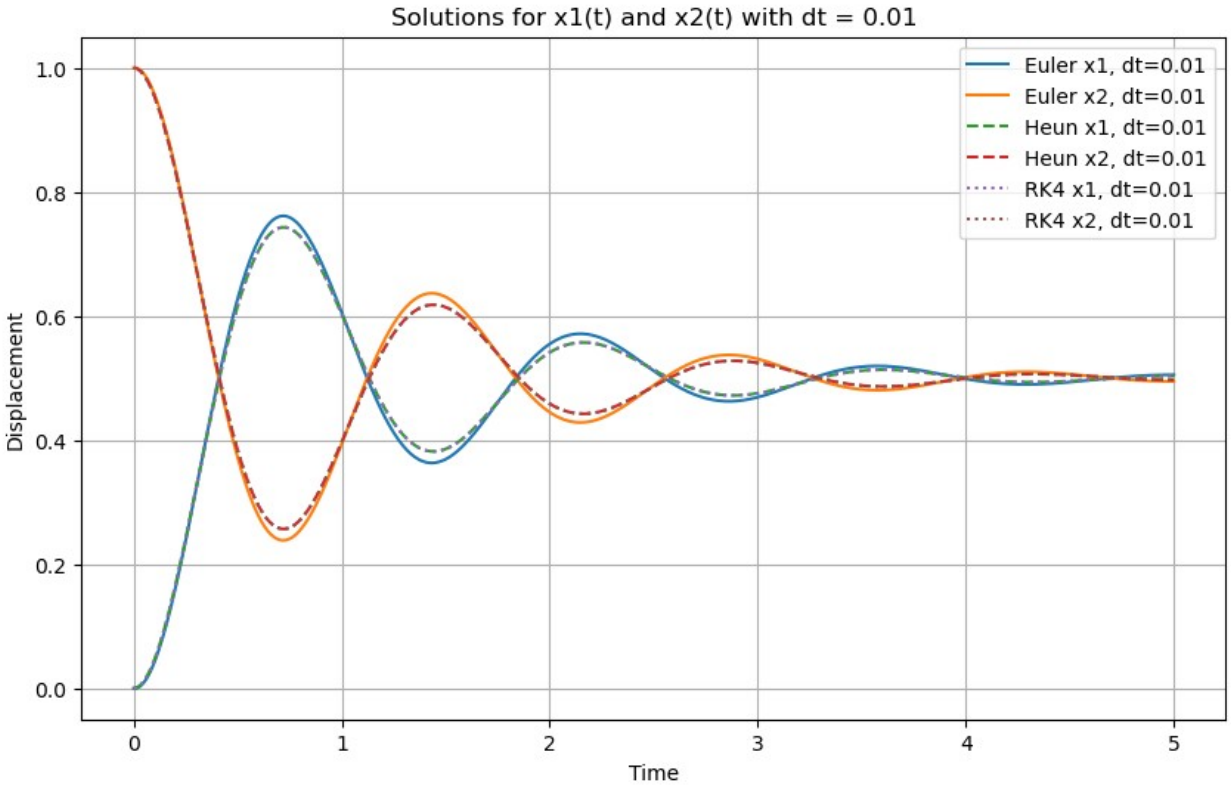
```
heun_solution[-1])
    rk4_error = calculate_global_error(true_solution[-1],
rk4_solution[-1])

    print(f"dt = {dt}:")
    print(f"  Euler method error: {euler_error}")
    print(f"  Heun's method error: {heun_error}")
    print(f"  4th-order Runge-Kutta error: {rk4_error}")
```



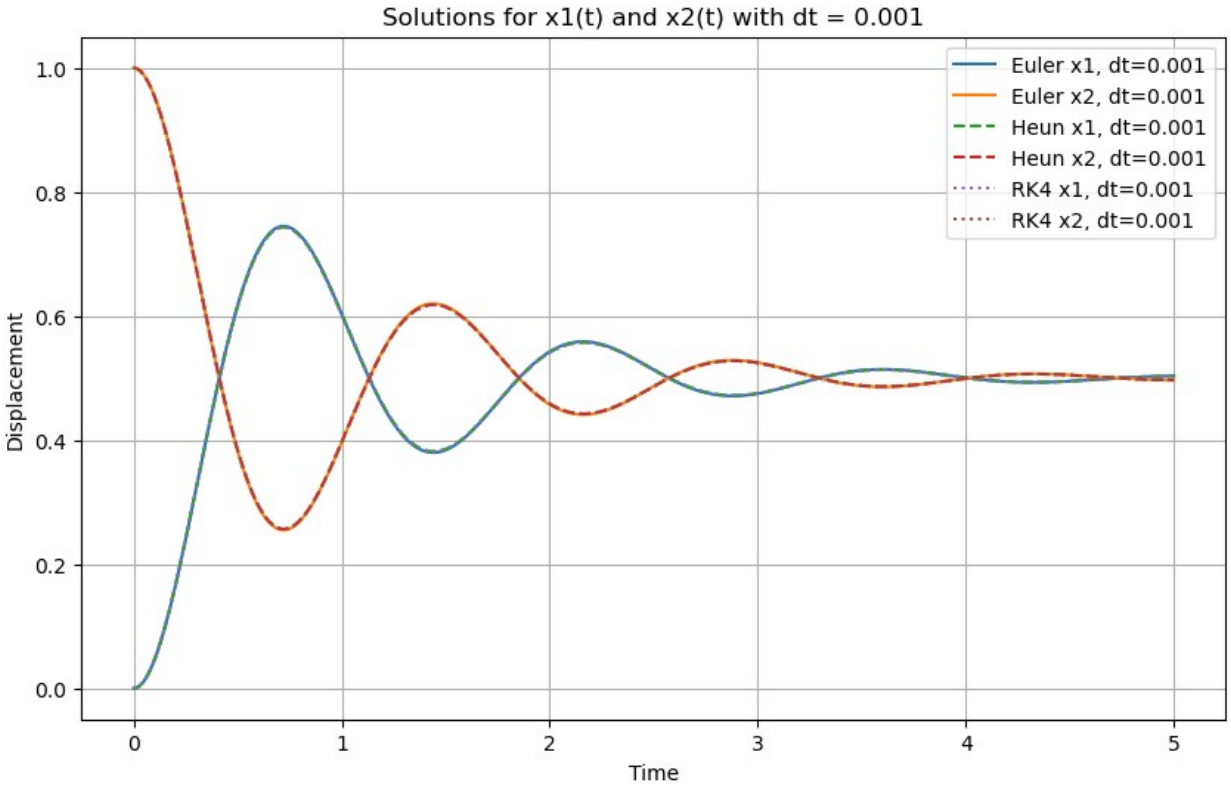

```
dt = 1:
  Euler method error: 5111.684571973921
  Heun's method error: 41753.94386323489
  4th-order Runge-Kutta error: 308610.39502344024
```



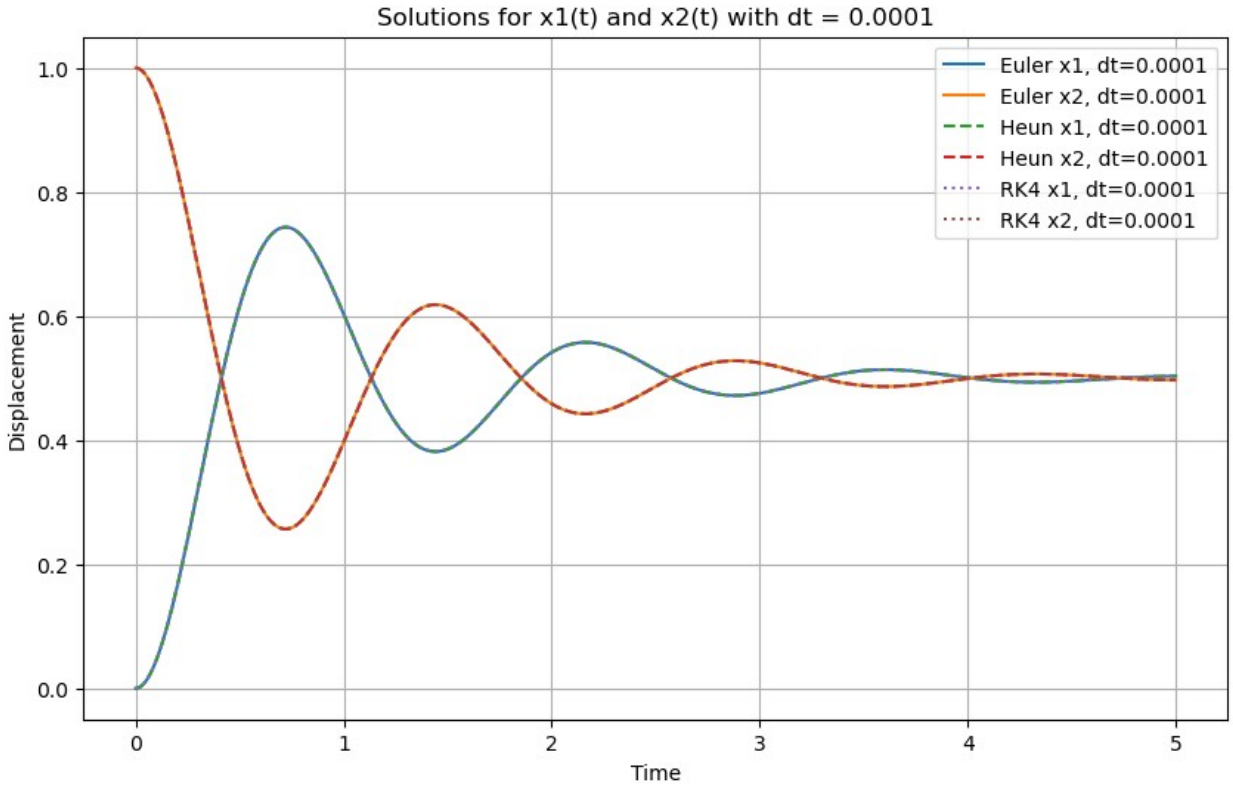

```
dt = 0.1:
  Euler method error: 1.8574438892622755
  Heun's method error: 0.013188724092078103
  4th-order Runge-Kutta error: 0.0001504055949161511
```




```
dt = 0.01:
  Euler method error: 0.005538861856692954
  Heun's method error: 0.00014770001376103955
  4th-order Runge-Kutta error: 1.0532216868749104e-08
```



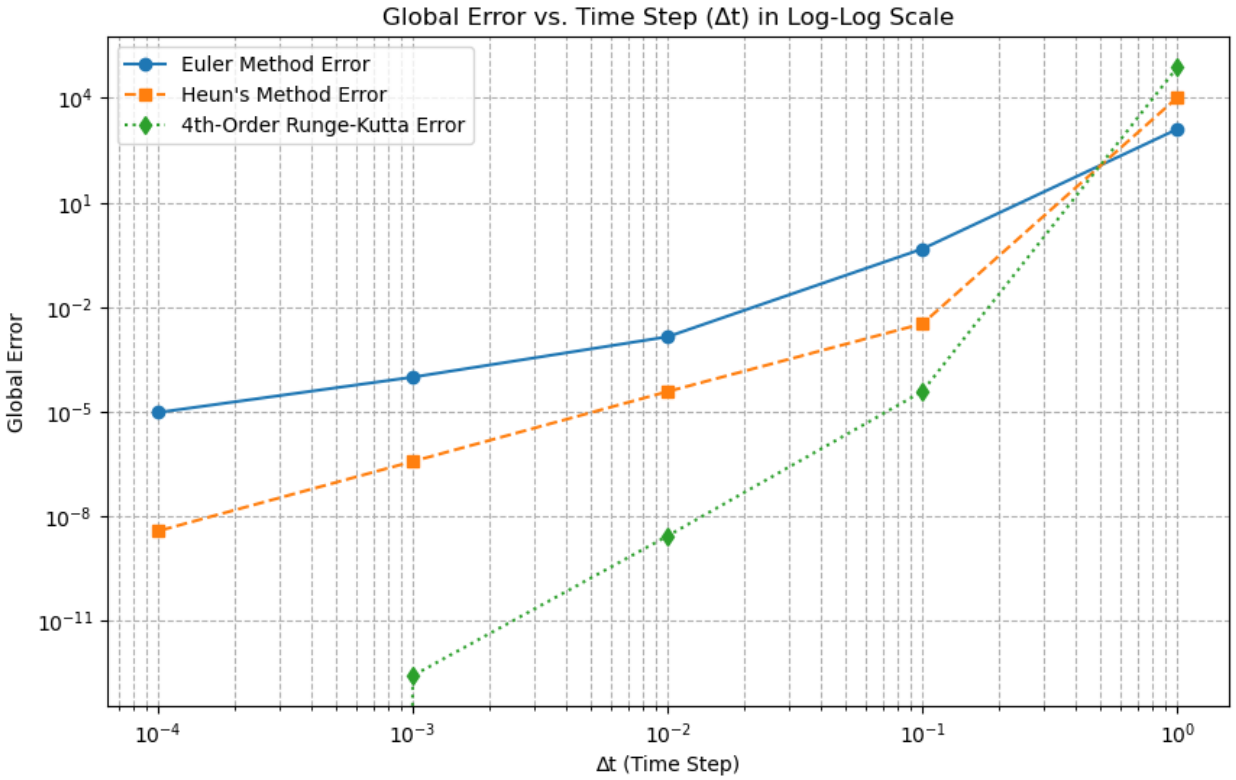

```
dt = 0.001:
  Euler method error: 0.00038302767044654277
  Heun's method error: 1.4516733944638419e-06
  4th-order Runge-Kutta error: 1.016206311654434e-12
```




```
dt = 0.0001:
  Euler method error: 3.6936974059225626e-05
  Heun's method error: 1.4486105610773247e-08
  4th-order Runge-Kutta error: 0.0

# Define the matrix B
B = np.array([
    [0, 1, 0, 0],
    [-10, -1, 10, 1],
    [0, 0, 0, 1],
    [10, 1, -10, -1]
])

# Initial conditions
X0 = np.array([0, 0, 1, 0])

# Time settings
T = 5
time_steps = [10**0, 10**-1, 10**-2, 10**-3, 10**-4]

# Function to compute the derivative dX/dt = B * X
def derivative(X):
    return B @ X

# Euler method
def euler_method(X0, dt, T):
```

```
    num_steps = int(T / dt)
    X = X0.copy()
    solution = [X]
    for _ in range(num_steps):
        X = X + dt * derivative(X)
        solution.append(X)
    return np.array(solution)

# Heun's method
def heun_method(X0, dt, T):
    num_steps = int(T / dt)
    X = X0.copy()
    solution = [X]
    for _ in range(num_steps):
        k1 = derivative(X)
        k2 = derivative(X + dt * k1)
        X = X + dt * 0.5 * (k1 + k2)
        solution.append(X)
    return np.array(solution)

# 4th-order Runge-Kutta method
def rk4_method(X0, dt, T):
    num_steps = int(T / dt)
    X = X0.copy()
    solution = [X]
    for _ in range(num_steps):
        k1 = derivative(X)
        k2 = derivative(X + 0.5 * dt * k1)
        k3 = derivative(X + 0.5 * dt * k2)
        k4 = derivative(X + dt * k3)
        X = X + (dt / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
        solution.append(X)
    return np.array(solution)

# Function to calculate global error for a given method
def calculate_global_error(true_solution, numerical_solution):
    return np.linalg.norm(true_solution - numerical_solution, ord=2) /
len(true_solution)

# Store errors for each method
euler_errors = []
heun_errors = []
rk4_errors = []

# Simulate and compute errors for each method and time step
for dt in time_steps:
    euler_solution = euler_method(X0, dt, T)
    heun_solution = heun_method(X0, dt, T)
    rk4_solution = rk4_method(X0, dt, T)
```

```
    # Using the RK4 solution with smallest dt as the "true" solution
    true_solution = rk4_method(X0, 10**-4, T)

    # Calculate global errors
    euler_error = calculate_global_error(true_solution[-1],
euler_solution[-1])
    heun_error = calculate_global_error(true_solution[-1],
heun_solution[-1])
    rk4_error = calculate_global_error(true_solution[-1],
rk4_solution[-1])

    # Store the errors
    euler_errors.append(euler_error)
    heun_errors.append(heun_error)
    rk4_errors.append(rk4_error)

# Plot global error vs. Δt in log-log scale
plt.figure(figsize=(10, 6))
plt.loglog(time_steps, euler_errors, 'o-', label="Euler Method Error")
plt.loglog(time_steps, heun_errors, 's--', label="Heun's Method
Error")
plt.loglog(time_steps, rk4_errors, 'd:', label="4th-Order Runge-Kutta
Error")
plt.xlabel('Δt (Time Step)')
plt.ylabel('Global Error')
plt.title('Global Error vs. Time Step (Δt) in Log-Log Scale')
plt.legend()
plt.grid(True, which="both", ls="--")
plt.show()
```

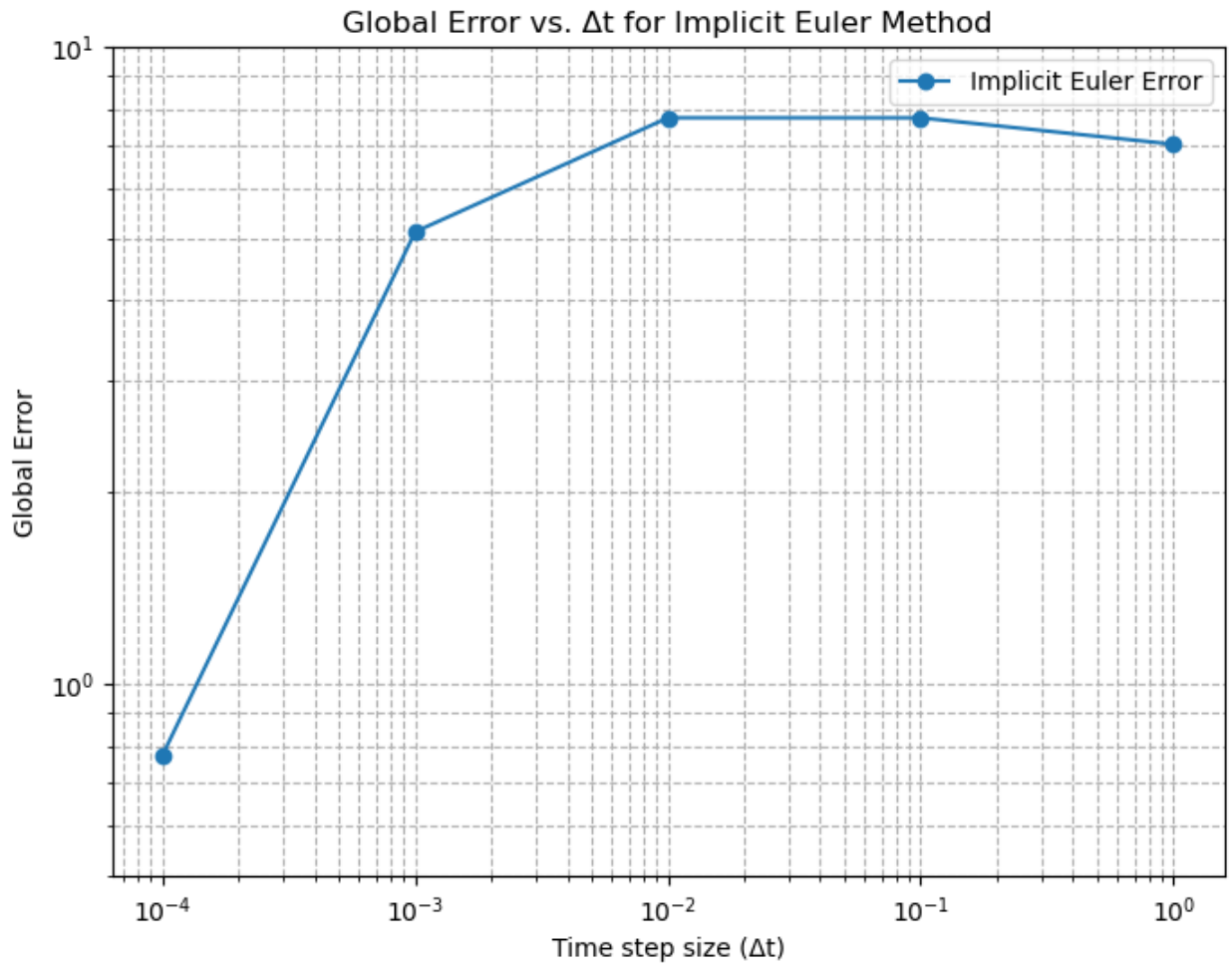

Global Error vs. Time Step (Δt) in Log-Log Scale
Euler Method Error
Heun's Method Error
4th-Order Runge-Kutta Error
Global Error
Δt (Time Step)
10^{4}
10^{1}
10^{-2}
10^{-5}
10^{-8}
10^{-11}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import solve
```

Problem 2, (4): solves the modified, stiff version of the problem with k=10^3 using the implicit Euler method and the 4th-order Runge-Kutta method

```
# New spring constant
k = 10**3

# Re-define the matrix B with updated spring constant
B = np.array([
    [0, 1, 0, 0],
    [-k, -1, k, 1],
    [0, 0, 0, 1],
    [k, 1, -k, -1]
])

# Initial conditions
X0 = np.array([0, 0, 1, 0])

# Time settings
T = 1
time_steps = [10**0, 10**-1, 10**-2, 10**-3, 10**-4]

# Function to compute the derivative dX/dt = B * X
def derivative(X):
    return B @ X

# Implicit Euler method
def implicit_euler_method(X0, dt, T):
    num_steps = int(T / dt)
    X = X0.copy()
    solution = [X]
    I = np.eye(4)  # Identity matrix for implicit calculation
    for _ in range(num_steps):
        X = solve(I - dt * B, X)  # Solving (I - dt*B) * X_next = X
        solution.append(X)
    return np.array(solution)

# 4th-order Runge-Kutta method
def rk4_method(X0, dt, T):
    num_steps = int(T / dt)
    X = X0.copy()
    solution = [X]
    for _ in range(num_steps):
        k1 = derivative(X)
        k2 = derivative(X + 0.5 * dt * k1)
```

```
        k3 = derivative(X + 0.5 * dt * k2)
        k4 = derivative(X + dt * k3)
        X = X + (dt / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
        solution.append(X)
    return np.array(solution)

# Function to calculate global error
def calculate_global_error(true_solution, numerical_solution):
    return np.linalg.norm(true_solution - numerical_solution, ord=2)

# Part (a): Implicit Euler - Plot global error vs. dt in log-log scale
errors_implicit_euler = []

# Using the RK4 solution with smallest dt as "true" solution
true_solution = rk4_method(X0, 10**-4, T)

for dt in time_steps:
    implicit_solution = implicit_euler_method(X0, dt, T)
    error = calculate_global_error(true_solution[-1],
implicit_solution[-1])
    errors_implicit_euler.append(error)

# Log-log plot for implicit Euler method errors
plt.figure(figsize=(8, 6))
plt.loglog(time_steps, errors_implicit_euler, '-o', label='Implicit
Euler Error')
plt.ylim([0.5, 1e1])
plt.xlabel('Time step size (Δt)')
plt.ylabel('Global Error')
plt.title('Global Error vs. Δt for Implicit Euler Method')
plt.grid(True, which="both", ls="--")
plt.legend()
plt.show()
```




```
# Part (b): Stability analysis for Runge-Kutta method
# Finding the maximum stable Δt
max_stable_dt = 0
stable_solution_found = False

for dt in time_steps:
    rk4_solution = rk4_method(X0, dt, T)
    if np.all(np.isfinite(rk4_solution)):
        max_stable_dt = dt
        stable_solution_found = True
    else:
        break

if stable_solution_found:
    print(f"Maximum stable Δt for Runge-Kutta method:
{max_stable_dt}")
else:
    print("Runge-Kutta method did not find a stable solution for any
of the given time steps.")
```

```
Maximum stable Δt for Runge-Kutta method: 0.0001
```