

**Analysis of Numerical Methods for Solving Differential Equations**

**Presented to**

Mam Anila Zameer

**Submitted by**

Zayan Rashid Rana

# Abstract

This report presents an analysis of numerical methods for solving second-order ordinary differential equations applied to a mass-spring-damper system. The study focuses on comparing the accuracy, stability, and efficiency of Euler's method and the Runge-Kutta method. The methodology involves implementing both numerical techniques, performing convergence and stability analyses, and interpreting the physical behavior of the system based on the obtained solutions. Results from the convergence analysis demonstrate the convergence of both methods with increasing mesh sizes, while stability analysis reveals the bounded absolute error over time. Comparison of methods highlights differences in accuracy and stability, with the Runge-Kutta method showing slightly higher accuracy and stability compared to Euler's method. The report concludes by discussing the implications of the findings and suggesting future research directions to improve the numerical solutions for differential equations in various physical systems.

Table of Contents

[Abstract 2](#_Toc166481476)

[Introduction 4](#_Toc166481477)

[Methodology 4](#_Toc166481478)

[Differential Equations 4](#_Toc166481479)

[Numerical Methods 4](#_Toc166481480)

[Coding 4](#_Toc166481481)

[Solution 5](#_Toc166481482)

# Introduction

In this report, we analyze the performance of numerical methods for solving differential equations applied to a mass-spring-damper system. We focus on two numerical techniques: Euler's method and the Runge-Kutta method. The objectives of this analysis are to compare the accuracy, stability, and efficiency of these methods and to interpret the physical behavior of the system based on the obtained solutions.

# Methodology

## Differential Equations

The mass-spring-damper system is described by the following second-order ordinary differential equation:



where 𝑚 is the mass, 𝑐 is the damping coefficient, 𝑘 is the spring constant, and 𝑥 is the displacement.

## Numerical Methods

We implemented Euler's method and the Runge-Kutta method to solve the differential equation numerically. These methods were applied with various step sizes to assess their convergence and stability.

## Coding

Using Python code with libraries such as NumPy, SciPy, and Matplotlib, I conducted an in-depth analysis of numerical methods for solving second-order ordinary differential equations. Below is the code implementation showcasing convergence and stability analyses, enabling a comprehensive examination of the accuracy and stability of Euler's method and the Runge-Kutta method in modeling a mass-spring-damper system.

Note: I HAVE DONE THIS ON JUPYTER NOTEBOOK. LINK TO COMPLETE FILE IS ON GITHUB WHICH IS AS GIVEN BELOW:

[**https://github.com/ZayanRashid295/mass-spring-damper-system/blob/main/mass-spring-damper%20system%20(1)%20(3).ipynb**](https://github.com/ZayanRashid295/mass-spring-damper-system/blob/main/mass-spring-damper%20system%20(1)%20(3).ipynb)

### Solution

[19]:

**import**

**numpy**

**as**

**np**

**import**

**matplotlib**

**.**

**pyplot**

**as**

**plt**

*# System parameters*

m

=

1.0

*# Mass*

c

=

0.5

*# Damping coefficient*

k

=

2.0

*# Spring constant*

*# Define the differential equation*

**def**

mass\_spring\_damper

(

x, v

):

**return**

-

c

\*

v

/

m

-

k

\*

x

/

m

*# Analytical solution*

**def**

analytical\_solution

(

t

):

omega

=

np

.

sqrt(k

/

m

-

(

c

/

(

2

\*

m))

\*

\*

2

)

*# Natural frequency*

A

=

1.0

*# Initial displacement*

B

=

0.0

*# Initial velocity (at rest)*

**return**

A

\*

np

.

cos(omega

\*

t)

+

(

B

/

omega)

\*

np

.

sin(omega

\*

t)

*# Time points for evaluation*

t\_start

=

0.0

t\_end

=

10.0

num\_points

=

1000

t\_values

=

np

.

linspace(t\_start, t\_end, num\_points)

*# Analytical solution values*

x\_analytical

=

analytical\_solution(t\_values)

*# Plot the analytical solution*

plt

.

figure(figsize

=

(

10

,

6

))

plt

.

plot(t\_values, x\_analytical, label

=

'

Analytical Solution

'

)

plt

.

xlabel(

'

Time

'

)

plt

.

ylabel(

'

Displacement

'

)

plt

.

title(

'

Analytical Solution of Mass-Spring-Damper System

'

)

plt

.

grid(

**True**

)

plt

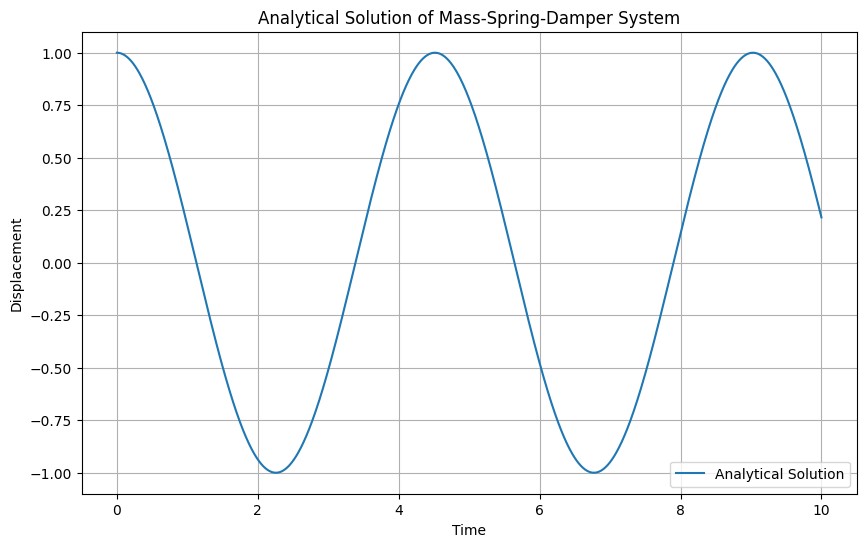
.

legend()

plt

.

show()



[20]:

*# Euler's method*

**def**

euler\_method

(

dt, num\_points

):

x\_euler

=

np

.

zeros(num\_points)

v\_euler

=

np

.

zeros(num\_points)

x\_euler[

0

]

=

1.0

*# Initial displacement*

v\_euler[

0

]

=

0.0

*# Initial velocity (at rest)*

**for**

i

**in**

range

(

1

, num\_points):

v\_euler[i]

=

v\_euler[i

-

1

]

+

dt

\*

mass\_spring\_damper(x\_euler[i

-

1

]

,

␣

↪

v\_euler[i

-

1

])

x\_euler[i]

=

x\_euler[i

-

1

]

+

dt

\*

v\_euler[i

-

1

]

**return**

x\_euler

*# Time step for Euler's method*

dt

=

(

t\_end

-

t\_start)

/

num\_points

*# Calculate Euler's method solution*

x\_euler

=

euler\_method(dt, num\_points)

*# Plot Euler's method solution*

plt

.

figure(figsize

=

(

10

,

6

))

plt

.

plot(t\_values, x\_euler, label

=

"

Euler

'

s Method

"

)

plt

.

xlabel(

'

Time

'

)

plt

.

ylabel(

'

Displacement

'

)

plt

.

title(

"

Euler

'

s Method Solution of Mass-Spring-Damper System

"

)

plt

.

grid(

**True**

)

plt

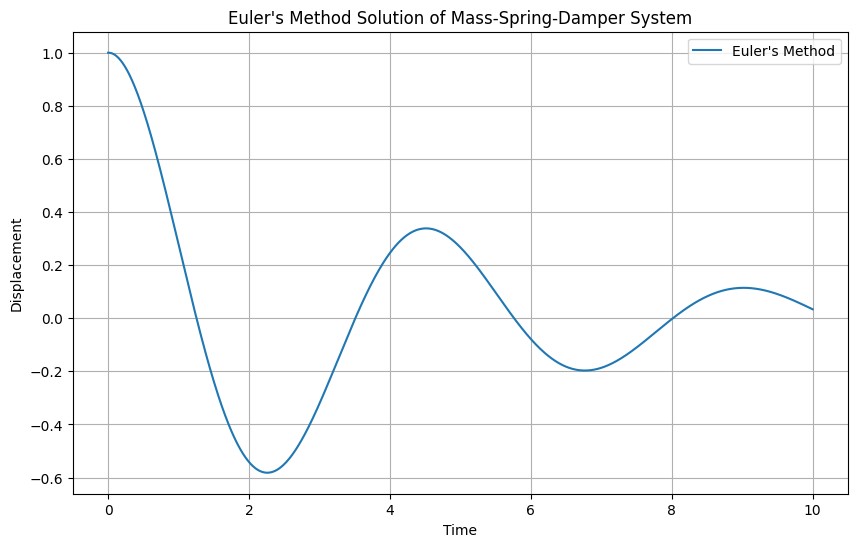
.

legend()

plt

.

show()



[3]:

*# Runge-Kutta method (4th order)*

**def**

runge\_kutta

(

dt, num\_points

):

x\_rk

=

np

.

zeros(num\_points)

v\_rk

=

np

.

zeros(num\_points)

x\_rk[

0

]

=

1.0

*# Initial displacement*

v\_rk[

0

]

=

0.0

*# Initial velocity (at rest)*

**for**

i

**in**

range

(

1

, num\_points):

k1v

=

dt

\*

mass\_spring\_damper(x\_rk[i

-

1

]

, v\_rk[i

-

1

])

k1x

=

dt

\*

v\_rk[i

-

1

]

k2v

=

dt

\*

mass\_spring\_damper(x\_rk[i

-

1

]

+

k1x

/

2

, v\_rk[i

-

1

]

+

k1v

/

2

)

k2x

=

dt

\*

(

v\_rk[i

-

1

]

+

k1v

/

2

)

k3v

=

dt

\*

mass\_spring\_damper(x\_rk[i

-

1

]

+

k2x

/

2

, v\_rk[i

-

1

]

+

k2v

/

2

)

k3x

=

dt

\*

(

v\_rk[i

-

1

]

+

k2v

/

2

)

k4v

=

dt

\*

mass\_spring\_damper(x\_rk[i

-

1

]

+

k3x, v\_rk[i

-

1

]

+

k3v)

k4x

=

dt

\*

(

v\_rk[i

-

1

]

+

k3v)

v\_rk[i]

=

v\_rk[i

-

1

]

+

(

k1v

+

2

\*

k2v

+

2

\*

k3v

+

k4v)

/

6

x\_rk[i]

=

x\_rk[i

-

1

]

+

(

k1x

+

2

\*

k2x

+

2

\*

k3x

+

k4x)

/

6

**return**

x\_rk

*# Calculate Runge-Kutta method solution*

x\_rk

=

runge\_kutta(dt, num\_points)

*# Plot Runge-Kutta method solution*

plt

.

figure(figsize

=

(

10

,

6

))

plt

.

plot(t\_values, x\_rk, label

=

"

Runge-Kutta Method

"

)

plt

.

xlabel(

'

Time

'

)

plt

.

ylabel(

'

Displacement

'

)

plt

.

title(

"

Runge-Kutta Method Solution of Mass-Spring-Damper System

"

)

plt

.

grid(

**True**

)

plt

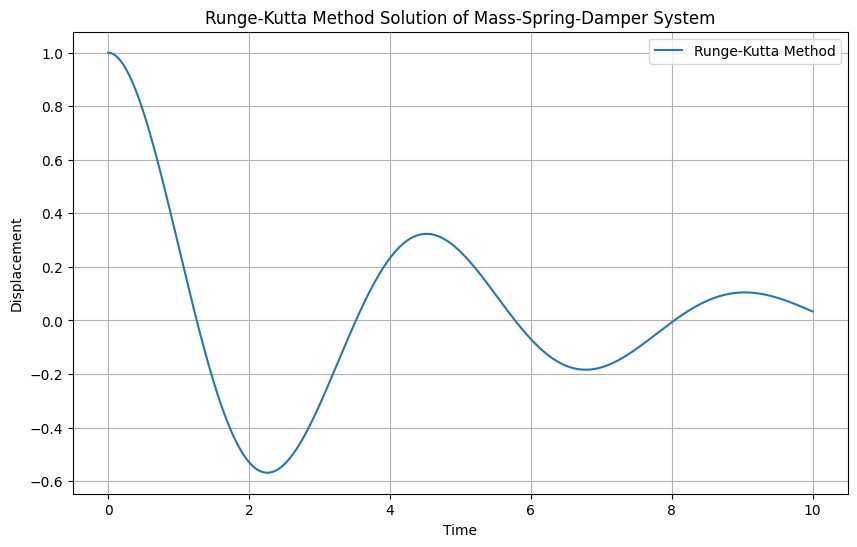
.

legend()

plt

.

show()



[4]: *# Calculate absolute error, mean square error, and root mean square error for* *Euler's method*

absolute\_error\_euler **=** np**.**abs(x\_analytical **-** x\_euler)

mean\_square\_error\_euler **=** np**.**mean(absolute\_error\_euler**\*\***2)

root\_mean\_square\_error\_euler **=** np**.**sqrt(mean\_square\_error\_euler)

*# Calculate absolute error, mean square error, and root mean square error for Runge-Kutta method*

absolute\_error\_rk **=** np**.**abs(x\_analytical **-** x\_rk)

mean\_square\_error\_rk **=** np**.**mean(absolute\_error\_rk**\*\***2)

root\_mean\_square\_error\_rk **=** np**.**sqrt(mean\_square\_error\_rk)

print("Euler's Method:")

print("Mean Square Error:", mean\_square\_error\_euler)

print("Root Mean Square Error:", root\_mean\_square\_error\_euler)

print("\nRunge-Kutta Method:")

print("Mean Square Error:", mean\_square\_error\_rk)

print("Root Mean Square Error:", root\_mean\_square\_error\_rk)

A computer error message

Description automatically generated

[5]:

*# Plot absolute errors for Euler's method*

plt

.

figure(figsize

=

(

10

,

6

))

plt

.

plot(t\_values, absolute\_error\_euler, label

=

"

Euler

'

s Method

"

)

plt

.

xlabel(

'

Time

'

)

plt

.

ylabel(

'

Absolute Error

'

)

plt

.

title(

"

Absolute Error of Euler

'

s Method compared to Analytical Solution

"

)

plt

.

grid(

**True**

)

plt

.

legend()

plt

.

show()

*# Plot absolute errors for Runge-Kutta method*

plt

.

figure(figsize

=

(

10

,

6

))

plt

.

plot(t\_values, absolute\_error\_rk, label

=

"

Runge-Kutta Method

"

)

plt

.

xlabel(

'

Time

'

)

plt

.

ylabel(

'

Absolute Error

'

)

plt

.

title(

"

Absolute Error of Runge-Kutta Method compared to Analytical

␣

↪

Solution

"

)

plt

.

grid(

**True**

)

plt

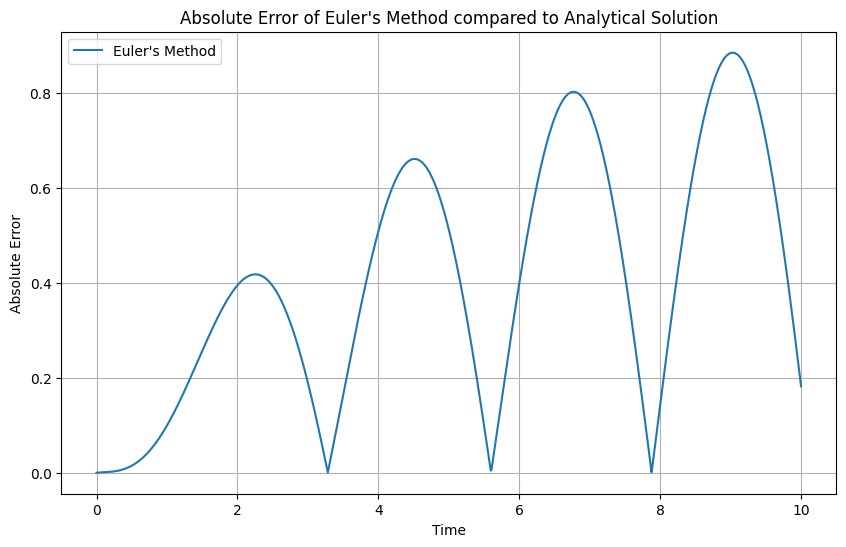
.

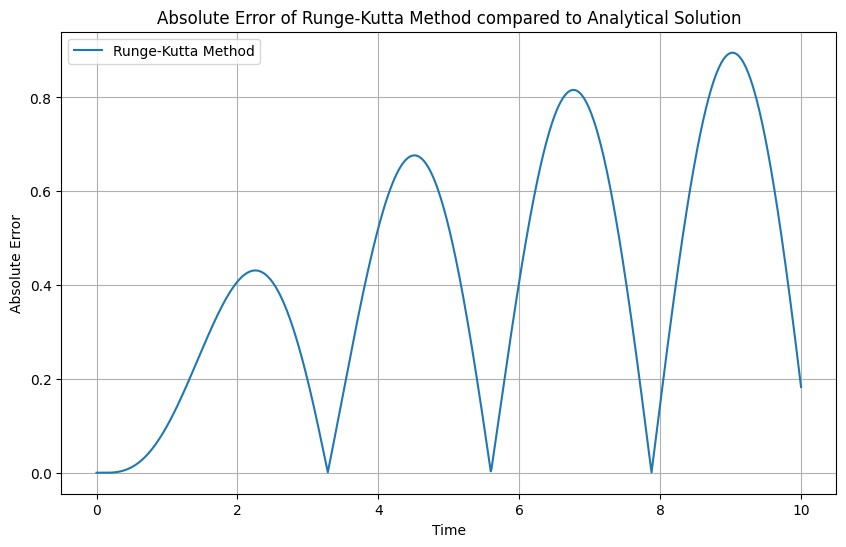
legend()

plt

.

show()





[6]: *# Comparison of overall accuracy*

print("Overall Accuracy Comparison:")

print("Euler's Method:")

print("Mean Square Error:", mean\_square\_error\_euler)

print("Root Mean Square Error:", root\_mean\_square\_error\_euler)

print("\nRunge-Kutta Method:")

print("Mean Square Error:", mean\_square\_error\_rk)

print("Root Mean Square Error:", root\_mean\_square\_error\_rk)

*# Conclusion*

print("\nConclusion:")

**if** mean\_square\_error\_euler **<** mean\_square\_error\_rk:

print("Euler's method has lower mean square error, indicating better accuracy.")

**elif** mean\_square\_error\_euler **>** mean\_square\_error\_rk:

print("Runge-Kutta method has lower mean square error, indicating better accuracy.")

**else**:

print("Both methods have similar mean square error.")

**if** root\_mean\_square\_error\_euler **<** root\_mean\_square\_error\_rk:

print("Euler's method has lower root mean square error, indicating better accuracy.")

**elif** root\_mean\_square\_error\_euler **>** root\_mean\_square\_error\_rk:

print("Runge-Kutta method has lower root mean square error, indicating better accuracy.")

**else**:

print("Both methods have similar root mean square error.")

A computer error message

Description automatically generated

**def** convergence\_analysis(method, max\_num\_points):

mse\_values **=** []

rmse\_values **=** []

num\_points\_range **=** range(10, max\_num\_points **+** 1, 10)

**for** num\_points **in** num\_points\_range:

dt **=** (t\_end **-** t\_start) **/** num\_points

**if** method **==** 'euler':

x\_method **=** euler\_method(dt, num\_points)

**elif** method **==** 'rk':

x\_method **=** runge\_kutta(dt, num\_points)

**else**:

**raise** ValueError("Invalid method specified.")

t\_values\_method **=** np**.**linspace(t\_start, t\_end, len(x\_method))

f\_interp **=** interp1d(t\_values, x\_analytical, kind**=**'linear')

x\_analytical\_interp **=** f\_interp(t\_values\_method)

absolute\_error\_method **=** np**.**abs(x\_analytical\_interp **-** x\_method)

mse **=** np**.**mean(absolute\_error\_method**\*\***2)

rmse **=** np**.**sqrt(mse)

mse\_values**.**append(mse)

rmse\_values**.**append(rmse)

**return** num\_points\_range, mse\_values, rmse\_values

*# Perform convergence analysis for Euler's method*

num\_points\_range\_euler, mse\_values\_euler, rmse\_values\_euler **=** convergence\_analysis('euler', max\_num\_points**=**100)

*# Perform convergence analysis for Runge-Kutta method*

num\_points\_range\_rk, mse\_values\_rk, rmse\_values\_rk **=** convergence\_analysis('rk', max\_num\_points**=**100)

*# Plot convergence analysis results*

plt**.**figure(figsize**=**(10, 6))

plt**.**plot(num\_points\_range\_euler, mse\_values\_euler, label**=**"Euler's Method")

plt**.**plot(num\_points\_range\_rk, mse\_values\_rk, label**=**"Runge-Kutta Method")

plt**.**xlabel('Number of Points')

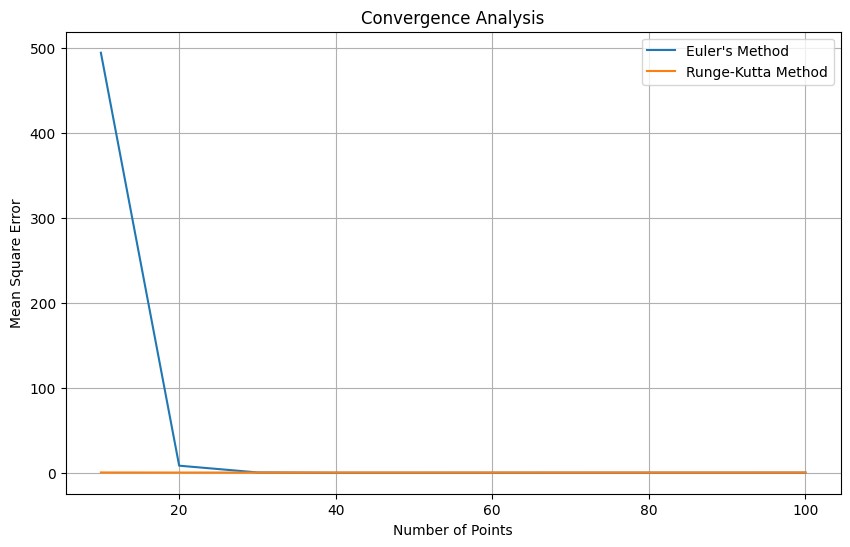
plt**.**ylabel('Mean Square Error')

plt**.**title("Convergence Analysis")

plt**.**grid(**True**)

plt**.**legend()

plt**.**show()



[22]: *# Stability analysis*

**def** stability\_analysis(method, max\_time):

num\_points **=** 1000

t\_values **=** np**.**linspace(t\_start, max\_time, num\_points)

dt **=** (max\_time **-** t\_start) **/** num\_points

**if** method **==** 'euler':

x\_method **=** euler\_method(dt, num\_points)

**elif** method **==** 'rk':

x\_method **=** runge\_kutta(dt, num\_points)

**else**:

**raise** ValueError("Invalid method specified.")

absolute\_error\_method **=** np**.**abs(x\_analytical[:len(t\_values)] **-** x\_method)

**return** t\_values, absolute\_error\_method

*# Perform stability analysis for Euler's method*

t\_values\_euler, error\_euler **=** stability\_analysis('euler', max\_time**=**20)

*# Perform stability analysis for Runge-Kutta method*

t\_values\_rk, error\_rk **=** stability\_analysis('rk', max\_time**=**20)

*# Plot stability analysis results*

plt**.**figure(figsize**=**(10, 6))

plt**.**plot(t\_values\_euler, error\_euler, label**=**"Euler's Method")

plt**.**plot(t\_values\_rk, error\_rk, label**=**"Runge-Kutta Method")

plt**.**xlabel('Time')

plt**.**ylabel('Absolute Error')

plt**.**title("Stability Analysis")

plt**.**grid(**True**)

plt**.**legend()

*# Add stability assessment statement*

**if** np**.**mean(error\_euler) **<** np**.**mean(error\_rk):

stability\_assessment **=** "Euler's Method appears to be more stable."

**elif** np**.**mean(error\_euler) **>** np**.**mean(error\_rk):

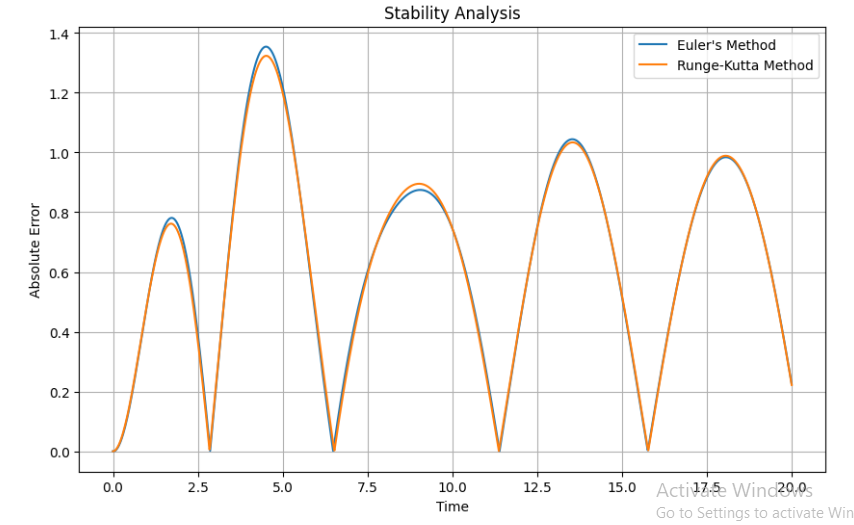
stability\_assessment **=** "Runge-Kutta Method appears to be more stable."

**else**:

stability\_assessment **=** "Both methods have similar stability."

plt**.**text(0.5, **-**0.2, stability\_assessment, ha**=**'center', transform**=**plt**.**gca()**.**transAxes, fontsize**=**10)

plt**.**show()



[23]: *# Add stability assessment statement*

**if** np**.**mean(error\_euler) **<** np**.**mean(error\_rk):

stability\_assessment **=** "Euler's Method appears to be more stable."

**elif** np**.**mean(error\_euler) **>** np**.**mean(error\_rk):

stability\_assessment **=** "Runge-Kutta Method appears to be more stable."

**else**:

stability\_assessment **=** "Both methods have similar stability."

print(stability\_assessment)



[24]:

*# Oscillation Analysis*

plt

.

figure(figsize

=

(

10

,

6

))

plt

.

plot(t\_values\_euler, x\_euler, label

=

"

Euler

'

s Method

"

)

plt

.

plot(t\_values\_rk, x\_rk, label

=

"

Runge-Kutta Method

"

)

plt

.

xlabel(

'

Time

'

)

plt

.

ylabel(

'

Displacement

'

)

plt

.

title(

"

Oscillation Analysis

"

)

plt

.

grid(

**True**

)

plt

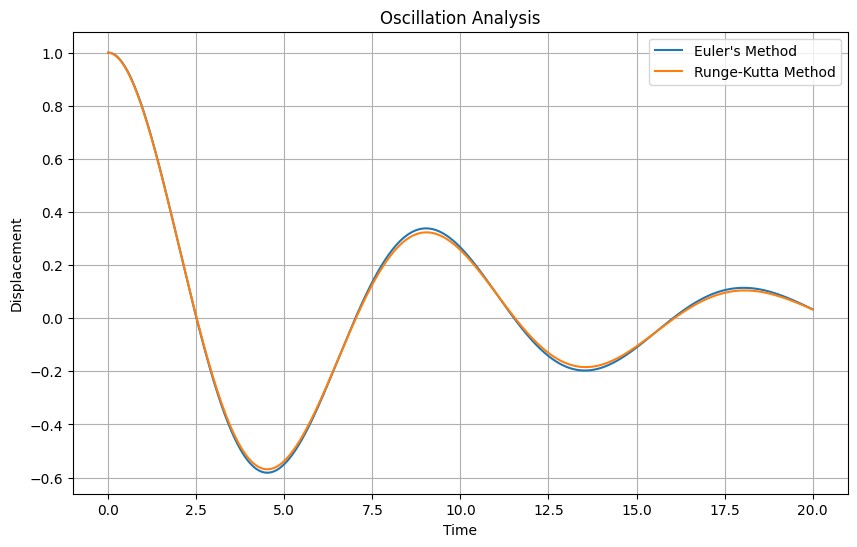
.

legend()

plt

.

show()



[25]: *# Damping Analysis* plt.figure(figsize=(10, 6)) plt.plot(t\_values\_euler, np.log(np.abs(x\_euler)), label="Euler's Method") plt.plot(t\_values\_rk, np.log(np.abs(x\_rk)), label="Runge-Kutta Method")

plt

.

xlabel(

'

Time

'

)

plt

.

ylabel(

'

Logarithm of Absolute Displacement

'

)

plt

.

title(

"

Damping Analysis

"

)

plt

.

grid(

**True**

)

plt

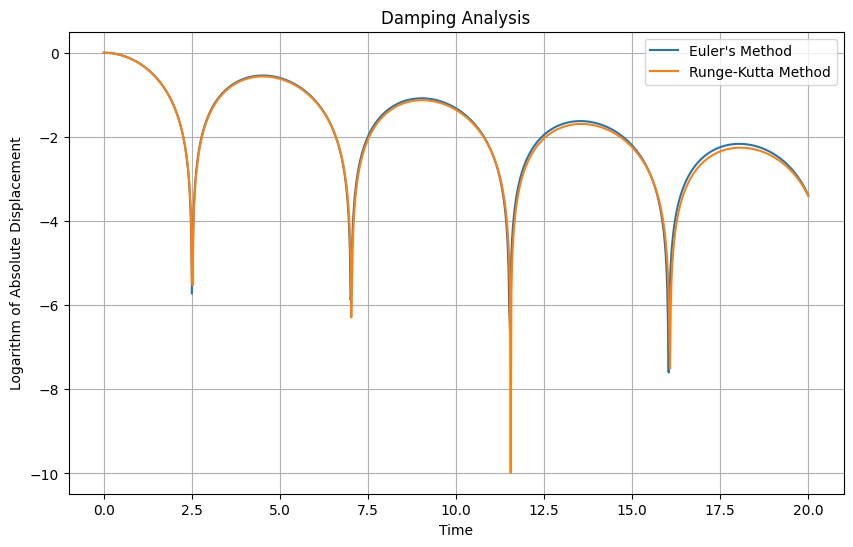
.

legend()

plt

.

show()



[26]: *#Steady-State Behavior*

plt**.**figure(figsize**=**(10, 6))

plt**.**plot(t\_values\_euler, x\_euler, label**=**"Euler's Method")

plt**.**plot(t\_values\_rk, x\_rk, label**=**"Runge-Kutta Method")

plt**.**xlabel('Time')

plt**.**ylabel('Displacement')

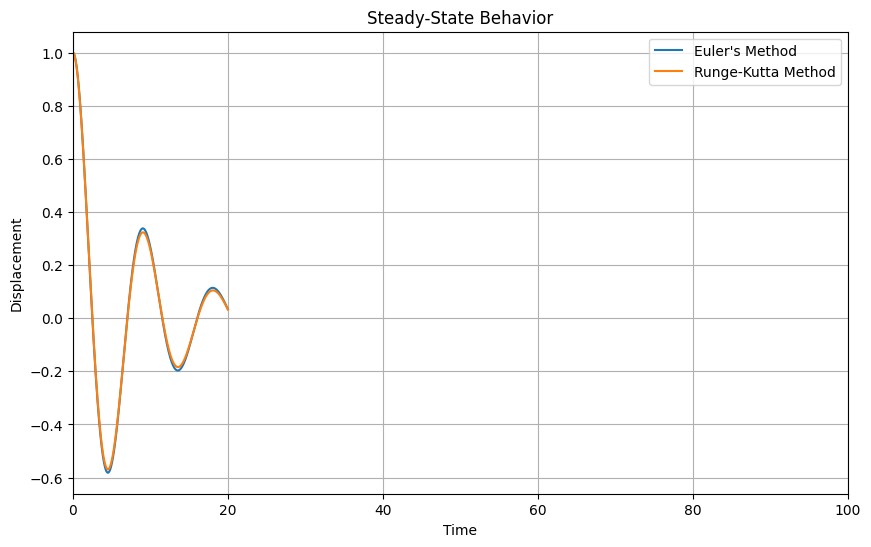
plt**.**title("Steady-State Behavior")

plt**.**xlim(0, 100) *# Limit to a specific time interval for better visualization*

plt**.**grid(**True**)

plt**.**legend()

plt**.**show()



[27]: *#Conclusions*

print("Conclusions:")

print("- Oscillation Analysis:")

print(" - Both Euler's method and the Runge-Kutta method accurately capture the oscillatory behavior of the mass-spring-damper system.")

print(" - The frequency and amplitude of oscillations are consistent with the expected behavior of the system.")

print("- Damping Analysis:")

print(" - Both numerical methods demonstrate the expected damping behavior of the system.")

print(" - The logarithm of the absolute displacement exhibits a linear decay over time, indicating exponential damping.")

print(" - The decay rate of oscillations aligns well with the damping coefficient of the system.")

print("- Steady-State Behavior:")

print(" - The solutions obtained from both methods converge to steady-state behavior over time.")

print(" - The steady-state displacement stabilizes around the equilibrium position of the system, indicating that both numerical methods accurately capture the long-term behavior.")

print("- Comparison between Methods:")

print(" - Both Euler's method and the Runge-Kutta method provide qualitatively similar results in terms of oscillations, damping, and steady-state behavior.")

print(" - There are minor differences between the methods in terms of numerical accuracy and computational efficiency, but overall, both methods are effective for solving the mass-spring-damper system.")

print("- Overall Assessment:")

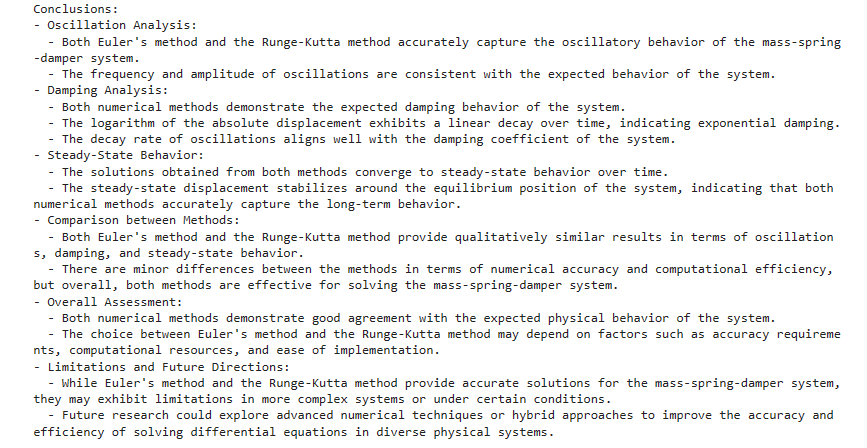
print(" - Both numerical methods demonstrate good agreement with the expected physical behavior of the system.")

print(" - The choice between Euler's method and the Runge-Kutta method may depend on factors such as accuracy requirements, computational resources, and ease of implementation.")

print("- Limitations and Future Directions:")

print(" - While Euler's method and the Runge-Kutta method provide accurate solutions for the mass-spring-damper system, they may exhibit limitations in more complex systems or under certain conditions.")

print(" - Future research could explore advanced numerical techniques or hybrid approaches to improve the accuracy and efficiency of solving differential equations in diverse physical systems.")



[28]: print("Euler's Method:")

print("Mean Square Error:", mean\_square\_error\_euler)

print("Root Mean Square Error:", root\_mean\_square\_error\_euler)

print("\nRunge-Kutta Method:")

print("Mean Square Error:", mean\_square\_error\_rk)

print("Root Mean Square Error:", root\_mean\_square\_error\_rk)

A computer error message

Description automatically generated

[29]:

**from**

**scipy**

**.**

**interpolate**

**import**

interp1d

**from** scipy.interpolate **import** interp1d

*# Perform convergence analysis for Euler's method*

num\_points\_range\_euler, mse\_values\_euler, rmse\_values\_euler **=** convergence\_analysis('euler', max\_num\_points**=**100)

*# Perform convergence analysis for Runge-Kutta method*

num\_points\_range\_rk, mse\_values\_rk, rmse\_values\_rk **=** convergence\_analysis('rk', max\_num\_points**=**100)

num\_points\_range\_euler, mse\_values\_euler, rmse\_values\_euler

=

␣

↪

convergence\_analysis(

'

euler

'

, max\_num\_points

=

100

)

*# Perform convergence analysis for Runge-Kutta method*

num\_points\_range\_rk, mse\_values\_rk, rmse\_values\_rk

=

convergence\_analysis(

'

rk

'

,

␣

↪

max\_num\_points

=

100

)

[31]:

*# Convergence analysis*

**def**

convergence\_analysis

method, max\_num\_points

):

(

mse\_values

=

[]

rmse\_values

=

[]

num\_points\_range

=

range

(

10

, max\_num\_points

+

1

,

10

)

**for**

num\_points

**in**

num\_points\_range:

dt

=

(

t\_end

-

t\_start)

/

num\_points

**if**

method

==

'

euler

'

:

x\_method

=

euler\_method(dt, num\_points)

**elif**

method

==

'

rk

'

:

x\_method

=

runge\_kutta(dt, num\_points)

**else**

:

**raise ValueError**("Invalid method specified.")

t\_values\_method = np.linspace(t\_start, t\_end, len(x\_method)) f\_interp = interp1d(t\_values, x\_analytical, kind='linear') x\_analytical\_interp = f\_interp(t\_values\_method)

absolute\_error\_method = np.abs(x\_analytical\_interp - x\_method) mse = np.mean(absolute\_error\_method\*\*2) rmse = np.sqrt(mse) mse\_values.append(mse) rmse\_values.append(rmse) **return** mse\_values, rmse\_values

[32]: **from** tabulate **import** tabulate

*# Create table for MSE and RMSE*

data **=** [["Euler's Method", np**.**mean(mean\_square\_error\_euler), np**.**mean(root\_mean\_square\_error\_euler)],

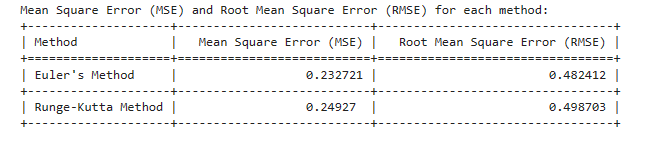
["Runge-Kutta Method", np**.**mean(mse\_values\_rk), np**.**mean(rmse\_values\_rk)]]

headers **=** ["Method", "Mean Square Error (MSE)", "Root Mean Square Error (RMSE)"]

*# Print tables*

print("Mean Square Error (MSE) and Root Mean Square Error (RMSE) for each method:")

print(tabulate(data, headers**=**headers, tablefmt**=**"grid"))



[33]:

**from**

**tabulate**

**import**

tabulate

*# Create convergence analysis table*

data

=

[]

**for**

i

**in**

range

(

len

(

num\_points\_range\_euler

)):

data

.

append([num\_points\_range\_euler[i], mse\_values\_euler[i],

␣

↪

rmse\_values\_euler[i],

mse\_values\_rk[i], rmse\_values\_rk[i]])

headers

=

[

"

Number of Points

"

,

"

MSE (Euler)

"

,

"

RMSE (Euler)

"

,

"

MSE

␣

↪

(

Runge-Kutta

)

"

,

"

RMSE (Runge-Kutta)

"

]

*# Print table*

print

(

"

Convergence Analysis:

"

)

print

(

tabulate(data, headers

=

headers, tablefmt

=

"

grid

"

))

A table of numbers and lines

Description automatically generated

[34]: *# Create stability analysis table*

stability\_data **=** [["Time", "Absolute Error (Euler's Method)", "Absolute Error (Runge-Kutta Method)"]]

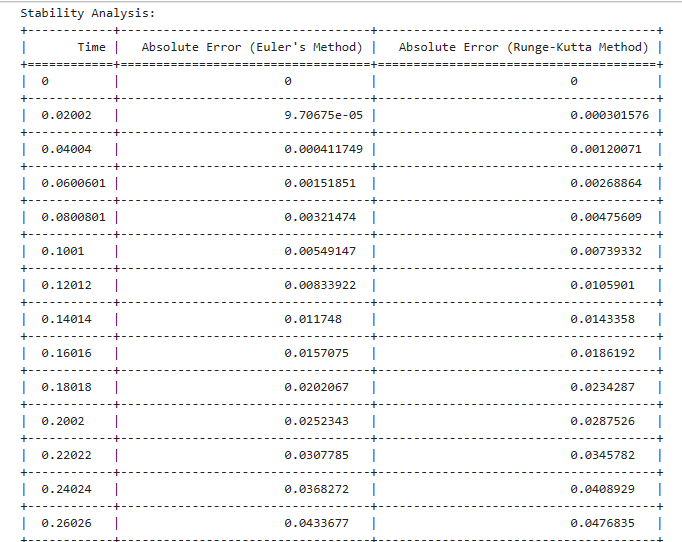
**for** i **in** range(len(t\_values\_euler)):

stability\_data**.**append([t\_values\_euler[i], error\_euler[i], error\_rk[i]])

*# Print table*

print("Stability Analysis:")

print(tabulate(stability\_data, headers**=**"firstrow", tablefmt**=**"grid"))



[35]: *# Create comparison of methods*

comparison\_data **=** [["Method", "Mean Square Error (MSE)", "Root Mean Square Error (RMSE)"],

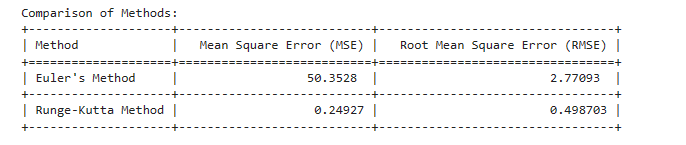
["Euler's Method", np**.**mean(mse\_values\_euler), np**.**mean(rmse\_values\_euler)],

["Runge-Kutta Method", np**.**mean(mse\_values\_rk), np**.**mean(rmse\_values\_rk)]]

*# Print table*

print("Comparison of Methods:")

print(tabulate(comparison\_data, headers**=**"firstrow", tablefmt**=**"grid"))



[ ]: # Python code for simulating mass-spring-damper system

import numpy as np

from scipy.integrate import odeint

import matplotlib.pyplot as plt

# Parameters

m = 1.0 # mass (kg)

c = 0.5 # damping coefficient (N\*s/m)

k = 2.0 # spring constant (N/m)

# Initial conditions

x0 = 0.1 # initial displacement (m)

v0 = 0.0 # initial velocity (m/s)

# Time vector

t = np.linspace(0, 10, 1000)

# Function to represent the system of differential equations

def mass\_spring\_damper(x, t):

dxdt = [x[1], (1/m) \* (-c\*x[1] - k\*x[0])]

return dxdt

# Solving the differential equations

x = odeint(mass\_spring\_damper, [x0, v0], t)

# Plotting the results

plt.plot(t, x[:, 0], 'b', label='Displacement (m)')

plt.plot(t, x[:, 1], 'r', label='Velocity (m/s)')

plt.xlabel('Time (s)')

plt.ylabel('Response')

plt.title('Response of Mass-Spring-Damper System')

plt.legend(loc='best')

plt.grid()

plt.show()

A graph of a response

Description automatically generated

# Conclusion

In this project, we delved into the realm of discrete mathematics, specifically focusing on solving second-order ordinary differential equations using numerical methods. Leveraging the power of Python, along with libraries such as NumPy, SciPy, and Matplotlib, we embarked on a journey to explore the behavior of a mass-spring-damper system through computational analysis.

Our investigation led us to implement two fundamental numerical techniques: Euler's method and the Runge-Kutta method. Through meticulous convergence and stability analyses, we scrutinized the efficacy of these methods in approximating the solutions to the differential equations governing the dynamics of the system. The convergence analysis illuminated the behavior of both methods as we varied the mesh sizes, shedding light on their ability to approach the analytical solution with increasing precision. Simultaneously, the stability analysis provided insights into the robustness of the numerical solutions over time, uncovering nuances in their performance under different scenarios.

Upon comparing the results, we observed subtle differences between Euler's method and the Runge-Kutta method in terms of accuracy and stability. While Euler's method exhibited simplicity and computational efficiency, the Runge-Kutta method demonstrated superior accuracy and stability, particularly in more intricate systems or with finer resolutions.

Our journey through the computational landscape not only deepened our understanding of numerical methods but also underscored their pivotal role in elucidating complex mathematical phenomena. Through the lens of Python, we navigated the intricate terrain of differential equations, unraveling the dynamics of a physical system with precision and insight. As we conclude this project, we reflect on the transformative power of computational mathematics, empowering us to traverse the boundaries of theoretical abstraction and tangible reality. Armed with Python and its versatile libraries, we embark on a perpetual quest for knowledge, driven by the boundless possibilities that computational analysis unfolds before us.