ZJU Computational Physics: Homework #2

Due on Monday, December 2, 2024

Github: https://github.com/NAKONAKO4/ZJU-computational-physics-NAKO

NAKO

Problem 1

Derivatives

> 将 forward formula, central difference formula, Richardson extrapolation 结果绘制在图中进行对比。

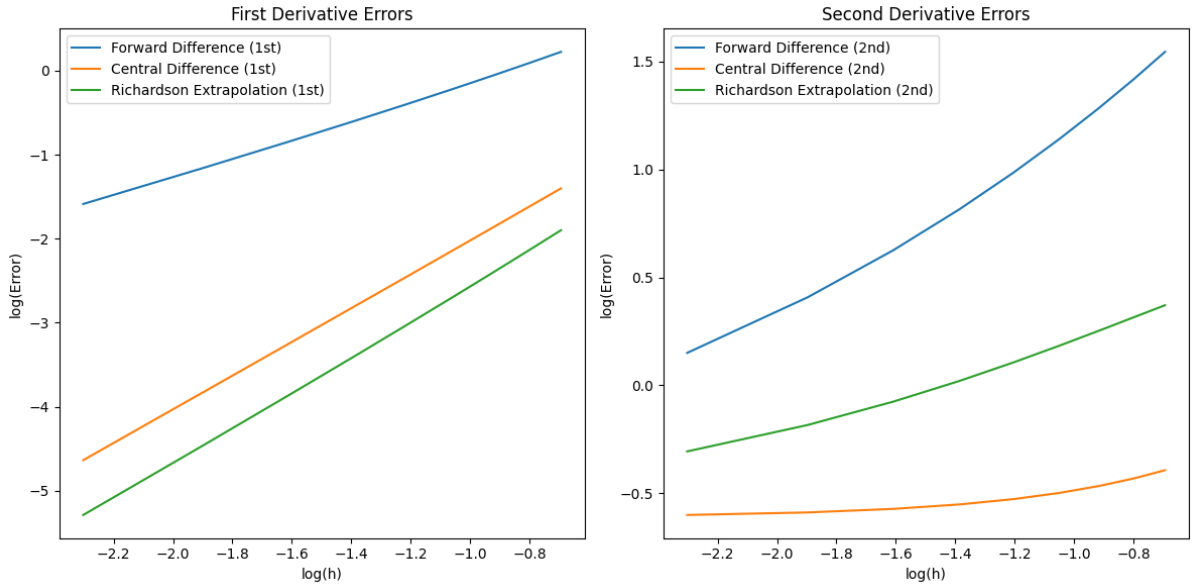
```python
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return x * np.cosh(x)

def df_exact(x):
    return np.cosh(x) + x * np.sinh(x)

def d2f_exact(x):
    return 2 * np.sinh(x) + x

def forward_diff(f, x, h):
    return (f(x + h) - f(x)) / h

def central_diff(f, x, h):
    return (f(x + h) - f(x - h)) / (2 * h)

def forward_diff_second(f, x, h):
    return (f(x + 2 * h) - 2 * f(x + h) + f(x)) / (h ** 2)

def central_diff_second(f, x, h):
    return (f(x + h) - 2 * f(x) + f(x - h)) / (h ** 2)

def richardson_extrapolation(f_diff, f, x, h, order):
    D1 = f_diff(f, x, h)
    D2 = f_diff(f, x, h / 2)
    return (2 ** order * D2 - D1) / (2 ** order - 1)

x = 1
h_values = np.arange(0.5, 0.05, -0.05)
log_h = np.log(h_values)

log_error_fd1 = []
log_error_cd1 = []
log_error_fd2 = []
log_error_cd2 = []
log_error_richardson1 = []
```

Problem 1 continued on next page...

```
log_error_richardson2 = []

for h in h_values:
    # First derivative
    fd1 = forward_diff(f, x, h)
    cd1 = central_diff(f, x, h)
    richardson1 = richardson_extrapolation(forward_diff, f, x, h, 1)

    log_error_fd1.append(np.log(np.abs(fd1 - df_exact(x))))
    log_error_cd1.append(np.log(np.abs(cd1 - df_exact(x))))
    log_error_richardson1.append(np.log(np.abs(richardson1 - df_exact(x))))

    # Second derivative
    fd2 = forward_diff_second(f, x, h)
    cd2 = central_diff_second(f, x, h)
    richardson2 = richardson_extrapolation(forward_diff_second, f, x, h, 2)

    log_error_fd2.append(np.log(np.abs(fd2 - d2f_exact(x))))
    log_error_cd2.append(np.log(np.abs(cd2 - d2f_exact(x))))
    log_error_richardson2.append(np.log(np.abs(richardson2 - d2f_exact(x))))

# Plot log errors
plt.figure(figsize=(12, 6))

# First derivative
plt.subplot(1, 2, 1)
plt.plot(log_h, log_error_fd1, label="Forward Difference (1st)")
plt.plot(log_h, log_error_cd1, label="Central Difference (1st)")
plt.plot(log_h, log_error_richardson1, label="Richardson Extrapolation (1st)")
plt.xlabel("log(h)")
plt.ylabel("log(Error)")
plt.title("First Derivative Errors")
plt.legend()

# Second derivative
plt.subplot(1, 2, 2)
plt.plot(log_h, log_error_fd2, label="Forward Difference (2nd)")
plt.plot(log_h, log_error_cd2, label="Central Difference (2nd)")
plt.plot(log_h, log_error_richardson2, label="Richardson Extrapolation (2nd)")
plt.xlabel("log(h)")
plt.ylabel("log(Error)")
plt.title("Second Derivative Errors")
plt.legend()

plt.tight_layout()
plt.show()
```

Problem 1 continued on next page...

运行结果为



b. 参见/1/b.py，图像中可见仍然存在一定的与实际值的误差。
运行结果为

```
Two-Point Derivatives: [1.0020006678319593, 4.004016292524781, 4.041794454678893, 27.673419111806652, -166.31140908884845]
Three-Point Derivatives: [1.0000006666652794, 4.000009333426924, 3.999918007924208, 27.999956520030622, -164.0181046536071]
Five-Point Derivatives: [1.0000000000054594, 4.000004666780299, 3.99994533561987, 27.999918300603, -164.0088620469271]
```

```python
import numpy as np

def f(x):
    return np.exp(x) / (np.sin(x)**3 + np.cos(x)**3)

def two_point(f, x, h, n):
    if n == 1:
        return (f(x + h) - f(x)) / h
    else:
        return (two_point(f, x + h, h, n - 1) - two_point(f, x, h, n - 1)) / h

def three_point(f, x, h, n):
    if n == 1:
        return (f(x + h) - f(x - h)) / (2 * h)
    else:
        return (three_point(f, x + h, h, n - 1) - three_point(f, x - h, h, n - 1)) / (2 * h)

def five_point(f, x, h, n):
    if n == 1:
```

Problem 1 continued on next page...

```
        return (-f(x + 2 * h) + 8 * f(x + h) - 8 * f(x - h) + f(x - 2 * h)) / (12 * h)
    else:
        return (five_point(f, x + h, h, n - 1) - five_point(f, x - h, h, n - 1)) / (2 * h)

x = 0
h = 0.001 # STEP SIZE. CHANGE HERE TO REDUCE THE ERROR.
derivatives = 5

results_two_point = [two_point(f, x, h, n) for n in range(1, derivatives + 1)]
results_three_point = [three_point(f, x, h, n) for n in range(1, derivatives + 1)]
results_five_point = [five_point(f, x, h, n) for n in range(1, derivatives + 1)]

print("Two-Point Derivatives:", results_two_point)
print("Three-Point Derivatives:", results_three_point)
print("Five-Point Derivatives:", results_five_point)
```

Problem 2

Integration

```
import numpy as np

def f1(x):
    return np.log(x)

def f2(x):
    return np.exp(-x**2)

def f3(x):
    return 1 / (1 + x**2)

def trapezoid_n(a, b, error_tolerance, second_derivative_max):
    return int(np.ceil(np.sqrt(((b - a)**3 * second_derivative_max) / (12 * error_tolerance)
        )))

def simpson_n(a, b, error_tolerance, fourth_derivative_max):
    return int(np.ceil((((b - a)**5 * fourth_derivative_max) / (180 * error_tolerance))
        **(1/4)))

f1_second_derivative_max = 1
f1_fourth_derivative_max = 1

f2_second_derivative_max = 2
f2_fourth_derivative_max = 12

f3_second_derivative_max = 4
```

Problem 2 continued on next page...

```
f3_fourth_derivative_max = 24

epsilon1 = 1e-8
epsilon2 = 1e-10
epsilon3 = 1e-12

n1_trap = trapezoid_n(1, 3, epsilon1, f1_second_derivative_max)
n1_simp = simpson_n(1, 3, epsilon1, f1_fourth_derivative_max)

n2_trap = trapezoid_n(-1, 1, epsilon2, f2_second_derivative_max)
n2_simp = simpson_n(-1, 1, epsilon2, f2_fourth_derivative_max)

n3_trap = trapezoid_n(1/2, 5/2, epsilon3, f3_second_derivative_max)
n3_simp = simpson_n(1/2, 5/2, epsilon3, f3_fourth_derivative_max)

print("I1 (log(x)): Trapezoid n =", n1_trap, ", Simpson n =", n1_simp)
print("I2 (exp(-x^2)): Trapezoid n =", n2_trap, ", Simpson n =", n2_simp)
print("I3 (1/(1+x^2)): Trapezoid n =", n3_trap, ", Simpson n =", n3_simp)
```

运行结果为:

```
I1 (log(x)): Trapezoid n = 8165 , Simpson n = 65
I2 (exp(-x^2)): Trapezoid n = 115471 , Simpson n = 383
I3 (1/(1+x^2)): Trapezoid n = 1632994 , Simpson n = 1438
```

整理如下

	Trapezoid n	Simpson n
$I_1(log(x))$	8165	65
$I_2(exp(-x^2))$	115471	383
$I_3(1/(1+x^2))$	1632994	1438

Problem 3

Hilbert Matrix

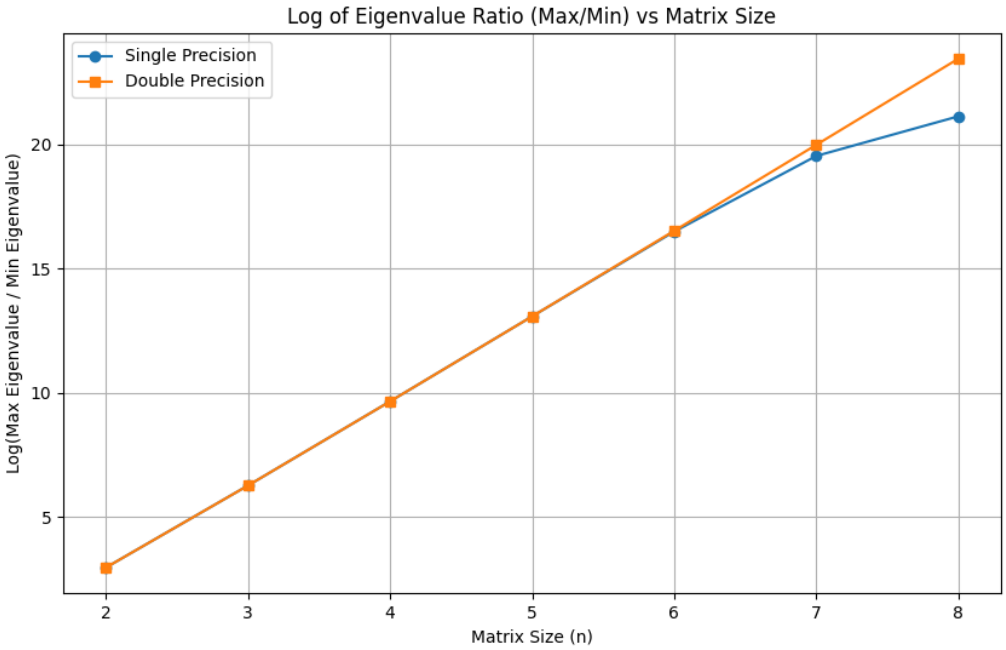
通过 python numpy 库的 linalg.eigvals() 函数可以直接将矩阵对角化并计算出矩阵的特征值。本题目使用了 numpy 库的 linalg.eigvals() 函数来对角化矩阵并计算特征值。

```
import numpy as np
```

Problem 3 continued on next page...

```python
import matplotlib.pyplot as plt


def hilbert_matrix(n):
    return np.array([[1 / (i + j + 1) for j in range(n)] for i in range(n)])


def eigenvalue_ratio_log(matrix):
    eigenvalues = np.linalg.eigvals(matrix)
    max_eigenvalue = np.max(np.abs(eigenvalues))
    min_eigenvalue = np.min(np.abs(eigenvalues))
    return np.log(max_eigenvalue / min_eigenvalue)


n_values = range(2, 9)

ratios_single = []
ratios_double = []

for n in n_values:
    H = hilbert_matrix(n)

    # Single precision
    H_single = H.astype(np.float32)
    ratios_single.append(eigenvalue_ratio_log(H_single))

    # Double precision
    H_double = H.astype(np.float64)
    ratios_double.append(eigenvalue_ratio_log(H_double))

plt.figure(figsize=(10, 6))
plt.plot(n_values, ratios_single, marker='o', label="Single Precision")
plt.plot(n_values, ratios_double, marker='s', label="Double Precision")
plt.title("Log of Eigenvalue Ratio (Max/Min) vs Matrix Size")
plt.xlabel("Matrix Size (n)")
plt.ylabel("Log(Max Eigenvalue / Min Eigenvalue)")
plt.legend()
plt.grid()
plt.show()
```

Problem 3 continued on next page...



Problem 4

Coupled Oscillators

a. 设置耦合系数为 1，每个振子的质量为 1，在 T=10 的模拟时长进行计算。首先通过欧拉方法来求解数值解，并与通过模态展开得到的解析解结果的 $u_j(t)$ 进行对比。可以得到结果为

```
Oscillator 1: Maximum Difference = 0.023230, Relative Error = 0.039907
Oscillator 2: Maximum Difference = 0.034412, Relative Error = 0.050140
Oscillator 3: Maximum Difference = 0.028043, Relative Error = 0.038147
Oscillator 4: Maximum Difference = 0.019648, Relative Error = 0.028608
Oscillator 5: Maximum Difference = 0.019422, Relative Error = 0.028863
Oscillator 6: Maximum Difference = 0.021836, Relative Error = 0.032791
Oscillator 7: Maximum Difference = 0.023721, Relative Error = 0.035859
Oscillator 8: Maximum Difference = 0.015188, Relative Error = 0.023069
Oscillator 9: Maximum Difference = 0.014331, Relative Error = 0.021847
Oscillator 10: Maximum Difference = 0.015219, Relative Error = 0.023298
Oscillator 11: Maximum Difference = 0.013294, Relative Error = 0.022193
Oscillator 12: Maximum Difference = 0.004273, Relative Error = 0.013044
```

Problem 4 continued on next page...

可以看到，欧拉方法得到的差值结果较大，相对误差较大。
于是我又尝试了用 Runge-Kutta 4th 方法去进行上述计算，得到结果如下

```
Oscillator 1: Maximum Difference = 0.0000000197, Relative Error = 0.0000000339
Oscillator 2: Maximum Difference = 0.0000000250, Relative Error = 0.0000000365
Oscillator 3: Maximum Difference = 0.0000000261, Relative Error = 0.0000000298
Oscillator 4: Maximum Difference = 0.0000000212, Relative Error = 0.0000000287
Oscillator 5: Maximum Difference = 0.0000000124, Relative Error = 0.0000000166
Oscillator 6: Maximum Difference = 0.0000000237, Relative Error = 0.0000000313
Oscillator 7: Maximum Difference = 0.0000000286, Relative Error = 0.0000000353
Oscillator 8: Maximum Difference = 0.0000000350, Relative Error = 0.0000000479
Oscillator 9: Maximum Difference = 0.0000000272, Relative Error = 0.0000000347
Oscillator 10: Maximum Difference = 0.0000000280, Relative Error = 0.0000000297
Oscillator 11: Maximum Difference = 0.0000000220, Relative Error = 0.0000000291
Oscillator 12: Maximum Difference = 0.0000000201, Relative Error = 0.0000000394
```

可以看到 Runge-Kutta 4th 方法得到的精度明显高于欧拉方法。

欧拉方法

```python
import numpy as np

N = 12
K = 1.0
m = 1.0
dt = 0.01
T = 10

frequencies = np.array([2 * np.sqrt(K / m) * np.sin((k * np.pi) / (2 * (N + 1))) for k in range(1, N + 1)])
modes = np.array([[np.sin((k * j * np.pi) / (N + 1)) for j in range(1, N + 1)] for k in range(1, N + 1)])
A_k = np.array([
    (2 / (N + 1)) * np.sum(
        [np.sin((k * j * np.pi) / (N + 1)) * (1 if j == 3 else 0) for j in range(1, N + 1)]
    ) / frequencies[k - 1]
    for k in range(1, N + 1)
])

def analytical_solution(t, j):
    return np.sum([
        A_k[k] * modes[k, j - 1] * np.sin(frequencies[k] * t)
        for k in range(N)
```

Problem 4 continued on next page...

```
    ])

def euler_method(dt, T, N, K, m):
    time_steps = int(T / dt)
    u = np.zeros((time_steps, N))
    v = np.zeros((time_steps, N))
    u[0, :] = 0
    v[0, 2] = 1

    for t in range(1, time_steps):
        for j in range(N):
            left = u[t - 1, j - 1] if j > 0 else 0
            right = u[t - 1, j + 1] if j < N - 1 else 0
            a_j = K / m * (left - 2 * u[t - 1, j] + right)
            v[t, j] = v[t - 1, j] + dt * a_j
            u[t, j] = u[t - 1, j] + dt * v[t - 1, j]

    return u, v

time_points = np.arange(0, T, dt)
u_num, v_num = euler_method(dt, T, N, K, m)

u_analytic = np.zeros_like(u_num)
for t_idx, t in enumerate(time_points):
    for j in range(1, N + 1):
        u_analytic[t_idx, j - 1] = analytical_solution(t, j)

max_differences_per_oscillator = np.max(np.abs(u_analytic - u_num), axis=0)
relative_errors_per_oscillator = max_differences_per_oscillator / np.max(np.abs(u_analytic),
     axis=0)

for j, (max_diff, rel_error) in enumerate(zip(max_differences_per_oscillator,
    relative_errors_per_oscillator), start=1):
    print(f"Oscillator {j}: Maximum Difference = {max_diff:.6f}, Relative Error = {rel_error
        :.6f}")
```

Runge-Kutta 4th 方法

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

N = 12
K = 1.0
m = 1.0
dt = 0.01
T = 100
```

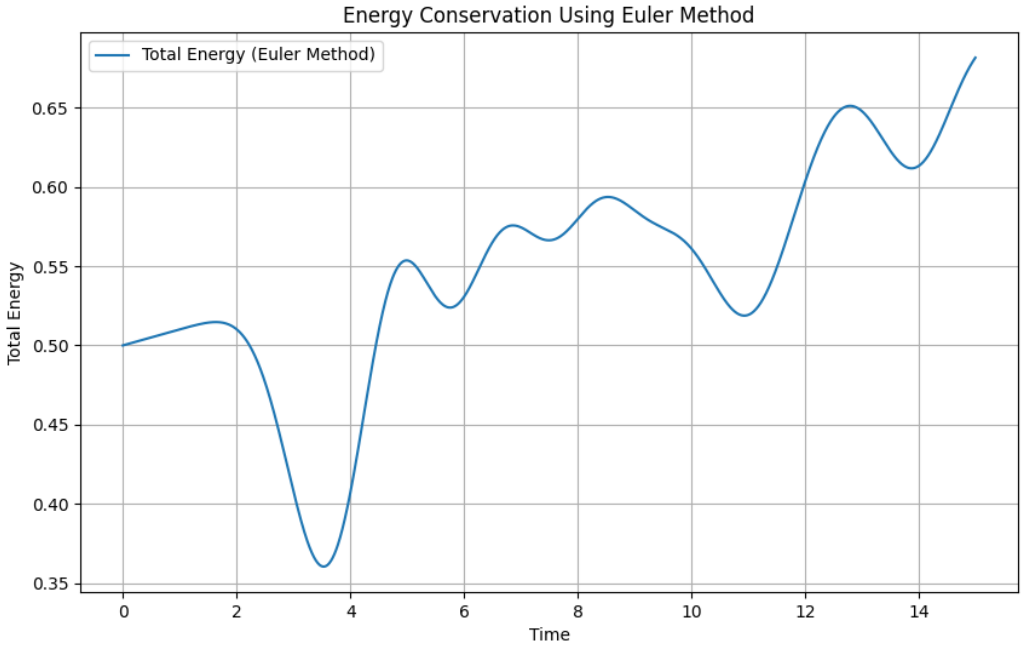
Problem 4 continued on next page...

```
9
10 frequencies = np.array([2 * np.sqrt(K / m) * np.sin((k * np.pi) / (2 * (N + 1))) for k in
       range(1, N + 1)])
11 modes = np.array([[np.sin((k * j * np.pi) / (N + 1)) for j in range(1, N + 1)] for k in
       range(1, N + 1)])
12 A_k = np.array([
13     (2 / (N + 1)) * np.sum(
14         [np.sin((k * j * np.pi) / (N + 1)) * (1 if j == 3 else 0) for j in range(1, N + 1)]
15     ) / frequencies[k - 1]
16     for k in range(1, N + 1)
17 ])
18
19 def analytical_solution(t, j):
20     return np.sum([
21         A_k[k] * modes[k, j - 1] * np.sin(frequencies[k] * t)
22         for k in range(N)
23     ])
24
25 def rk4_method(dt, T, N, K, m):
26     time_steps = int(T / dt)
27     u = np.zeros((time_steps, N))
28     v = np.zeros((time_steps, N))
29     u[0, :] = 0
30     v[0, 2] = 1
31
32     def acceleration(u):
33         acc = np.zeros(N)
34         for j in range(N):
35             left = u[j - 1] if j > 0 else 0
36             right = u[j + 1] if j < N - 1 else 0
37             acc[j] = K / m * (left - 2 * u[j] + right)
38         return acc
39
40     for t in range(1, time_steps):
41         k1_u = v[t - 1]
42         k1_v = acceleration(u[t - 1])
43
44         k2_u = v[t - 1] + 0.5 * dt * k1_v
45         k2_v = acceleration(u[t - 1] + 0.5 * dt * k1_u)
46
47         k3_u = v[t - 1] + 0.5 * dt * k2_v
48         k3_v = acceleration(u[t - 1] + 0.5 * dt * k2_u)
49
50         k4_u = v[t - 1] + dt * k3_v
51         k4_v = acceleration(u[t - 1] + dt * k3_u)
52
53         u[t] = u[t - 1] + dt / 6 * (k1_u + 2 * k2_u + 2 * k3_u + k4_u)
```

Problem 4 continued on next page...

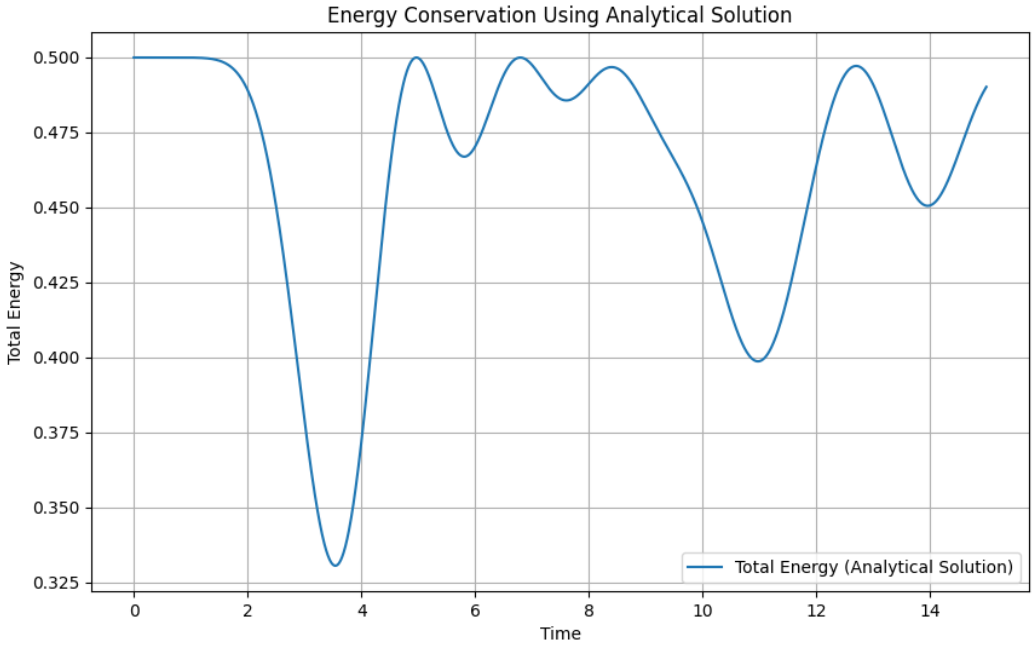
```
        v[t] = v[t - 1] + dt / 6 * (k1_v + 2 * k2_v + 2 * k3_v + k4_v)

    return u, v

time_points = np.arange(0, T, dt)
u_num, v_num = rk4_method(dt, T, N, K, m)

u_analytic = np.zeros_like(u_num)
for t_idx, t in enumerate(time_points):
    for j in range(1, N + 1):
        u_analytic[t_idx, j - 1] = analytical_solution(t, j)

difference = np.abs(u_analytic - u_num)

max_differences_per_oscillator = np.max(np.abs(u_analytic - u_num), axis=0)
relative_errors_per_oscillator = max_differences_per_oscillator / np.max(np.abs(u_analytic),
    axis=0)

for j, (max_diff, rel_error) in enumerate(zip(max_differences_per_oscillator,
    relative_errors_per_oscillator), start=1):
    print(f"Oscillator {j}: Maximum Difference = {max_diff:.10f}, Relative Error = {
        rel_error:.10f}")
```

b. 采用欧拉方法计算，得到结果如下

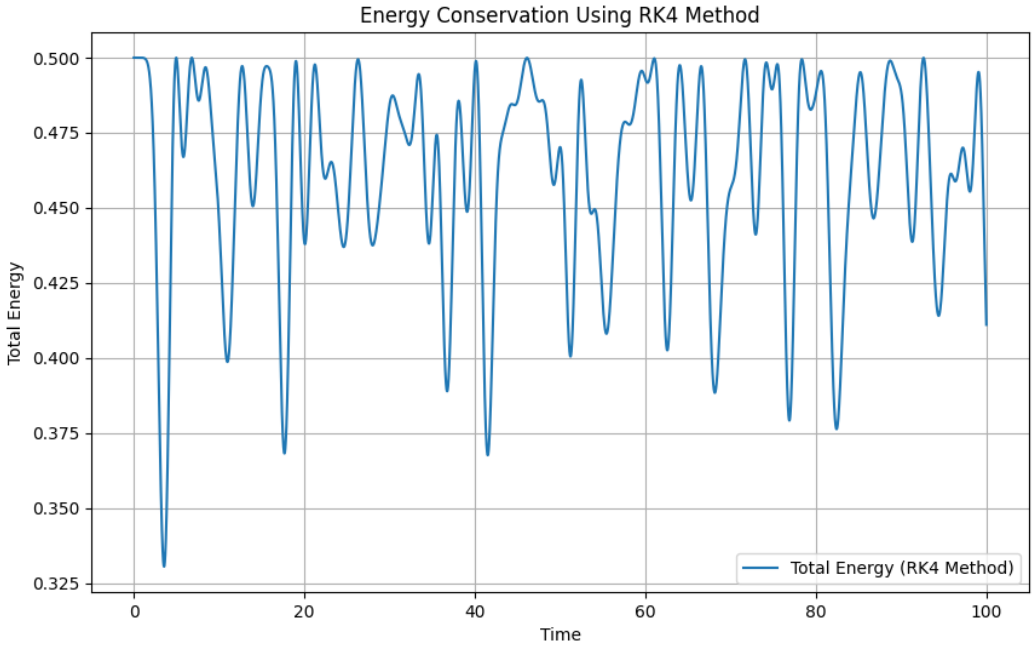


Problem 4 continued on next page...

但事实上与解析解能量进行对比可以看到，欧拉方法与解析解能量并不吻合。



同时也可以看到，在更长的时间上进行计算，解析解的能量守恒并不是很好，在初始能量 0.5 下面波动，但是均值还是在 0.475 附近，存在一些比较大的波动到达 0.375 附近。



Problem 4 continued on next page...

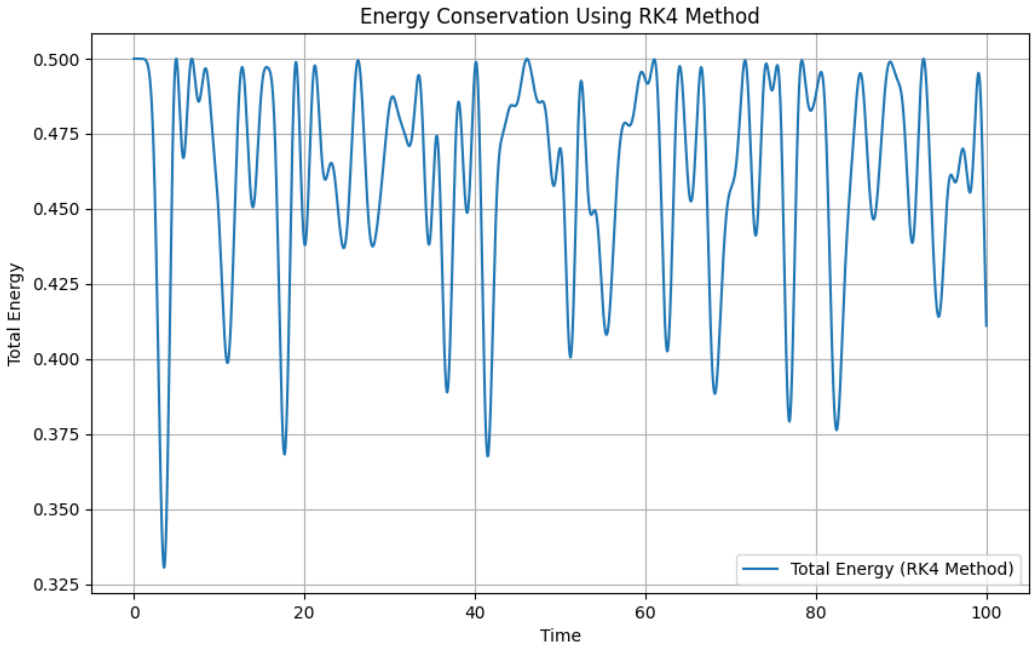
```python
import numpy as np
import matplotlib.pyplot as plt

N = 12
K = 1.0
m = 1.0
dt = 0.01
T = 15

frequencies = np.array([2 * np.sqrt(K / m) * np.sin((k * np.pi) / (2 * (N + 1))) for k in range(1, N + 1)])
modes = np.array([[np.sin((k * j * np.pi) / (N + 1)) for j in range(1, N + 1)] for k in range(1, N + 1)])
A_k = np.array([
    (2 / (N + 1)) * np.sum(
        [np.sin((k * j * np.pi) / (N + 1)) * (1 if j == 3 else 0) for j in range(1, N + 1)]
    ) / frequencies[k - 1]
    for k in range(1, N + 1)
])

def analytical_solution(t, j):
    return np.sum([
        A_k[k] * modes[k, j - 1] * np.sin(frequencies[k] * t)
        for k in range(N)
    ])


def euler_method(dt, T, N, K, m):
    time_steps = int(T / dt)
    u = np.zeros((time_steps, N))
    v = np.zeros((time_steps, N))
    u[0, :] = 0
    v[0, 2] = 1

    for t in range(1, time_steps):
        for j in range(N):
            left = u[t - 1, j - 1] if j > 0 else 0
            right = u[t - 1, j + 1] if j < N - 1 else 0
            a_j = K / m * (left - 2 * u[t - 1, j] + right)
            v[t, j] = v[t - 1, j] + dt * a_j
            u[t, j] = u[t - 1, j] + dt * v[t - 1, j]

    return u, v

time_points = np.arange(0, T, dt)
u_num, v_num = euler_method(dt, T, N, K, m)
```

Problem 4 continued on next page...

```
45
46 u_analytic = np.zeros_like(u_num)
47 for t_idx, t in enumerate(time_points):
48     for j in range(1, N + 1):
49         u_analytic[t_idx, j - 1] = analytical_solution(t, j)
50 def compute_total_energy(u, v, K, m):
51     kinetic_energy = 0.5 * m * np.sum(v**2, axis=1)
52     potential_energy = 0.5 * K * np.sum((np.diff(u, axis=1))**2, axis=1)
53     return kinetic_energy + potential_energy
54
55 energy_values = compute_total_energy(u_num, v_num, K, m)
56
57 plt.figure(figsize=(10, 6))
58 plt.plot(time_points, energy_values, label="Total Energy (Euler Method)")
59 plt.title("Energy Conservation Using Euler Method")
60 plt.xlabel("Time")
61 plt.ylabel("Total Energy")
62 plt.legend()
63 plt.grid()
64 plt.show()
65
66 def compute_energy_analytic(u_analytic, K, m, time_points):
67     v_analytic = np.zeros_like(u_analytic)
68     for t_idx, t in enumerate(time_points):
69         for j in range(1, N + 1):
70             v_analytic[t_idx, j - 1] = np.sum([
71                 A_k[k] * modes[k, j - 1] * frequencies[k] * np.cos(frequencies[k] * t)
72                 for k in range(N)
73             ])
74     kinetic_energy = 0.5 * m * np.sum(v_analytic**2, axis=1)
75     potential_energy = 0.5 * K * np.sum((np.diff(u_analytic, axis=1))**2, axis=1)
76     return kinetic_energy + potential_energy
77
78 analytic_energy_values = compute_energy_analytic(u_analytic, K, m, time_points)
79
80 import matplotlib.pyplot as plt
81
82 plt.figure(figsize=(10, 6))
83 plt.plot(time_points, analytic_energy_values, label="Total Energy (Analytical Solution)")
84 plt.title("Energy Conservation Using Analytical Solution")
85 plt.xlabel("Time")
86 plt.ylabel("Total Energy")
87 plt.legend()
88 plt.grid()
89 plt.show()
```

Problem 4 continued on next page...

c. 结果如下，可以看到跟解析解十分吻合，同样也是没有很好的能量守恒，存在一些较大的波动，整体均值在接近初始能量 0.5 附近。




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

N = 12
K = 1.0
m = 1.0
dt = 0.01
T = 100

frequencies = np.array([2 * np.sqrt(K / m) * np.sin((k * np.pi) / (2 * (N + 1))) for k in range(1, N + 1)])
modes = np.array([[np.sin((k * j * np.pi) / (N + 1)) for j in range(1, N + 1)] for k in range(1, N + 1)])
A_k = np.array([
    (2 / (N + 1)) * np.sum(
        [np.sin((k * j * np.pi) / (N + 1)) * (1 if j == 3 else 0) for j in range(1, N + 1)]
    ) / frequencies[k - 1]
    for k in range(1, N + 1)
])

def analytical_solution(t, j):
    return np.sum([
        A_k[k] * modes[k, j - 1] * np.sin(frequencies[k] * t)
```

Problem 4 continued on next page...

```
        for k in range(N)
    ])

def rk4_method(dt, T, N, K, m):
    time_steps = int(T / dt)
    u = np.zeros((time_steps, N))
    v = np.zeros((time_steps, N))
    u[0, :] = 0
    v[0, 2] = 1

    def acceleration(u):
        acc = np.zeros(N)
        for j in range(N):
            left = u[j - 1] if j > 0 else 0
            right = u[j + 1] if j < N - 1 else 0
            acc[j] = K / m * (left - 2 * u[j] + right)
        return acc

    for t in range(1, time_steps):
        k1_u = v[t - 1]
        k1_v = acceleration(u[t - 1])

        k2_u = v[t - 1] + 0.5 * dt * k1_v
        k2_v = acceleration(u[t - 1] + 0.5 * dt * k1_u)

        k3_u = v[t - 1] + 0.5 * dt * k2_v
        k3_v = acceleration(u[t - 1] + 0.5 * dt * k2_u)

        k4_u = v[t - 1] + dt * k3_v
        k4_v = acceleration(u[t - 1] + dt * k3_u)

        u[t] = u[t - 1] + dt / 6 * (k1_u + 2 * k2_u + 2 * k3_u + k4_u)
        v[t] = v[t - 1] + dt / 6 * (k1_v + 2 * k2_v + 2 * k3_v + k4_v)

    return u, v

time_points = np.arange(0, T, dt)
u_num, v_num = rk4_method(dt, T, N, K, m)

u_analytic = np.zeros_like(u_num)
for t_idx, t in enumerate(time_points):
    for j in range(1, N + 1):
        u_analytic[t_idx, j - 1] = analytical_solution(t, j)

difference = np.abs(u_analytic - u_num)

plt.figure(figsize=(12, 8))
```

Problem 4 continued on next page...

```python
for j in range(1, N + 1):
    plt.plot(time_points, difference[:, j - 1], label=f'Oscillator {j}')

plt.title("Difference Between Analytical and RK4 Numerical Solutions")
plt.xlabel("Time")
plt.ylabel("Difference in Displacement")
plt.legend(loc='upper right', fontsize='small')
plt.grid()
plt.show()



v_analytic = np.zeros_like(v_num)

for t_idx, t in enumerate(time_points):
    for j in range(1, N + 1):
        v_analytic[t_idx, j - 1] = np.sum([
            A_k[k] * modes[k, j - 1] * frequencies[k] * np.cos(frequencies[k] * t)
            for k in range(N)
        ])

def compute_energy(u, v, K, m):
    kinetic_energy = 0.5 * m * np.sum(v**2, axis=1)
    potential_energy = 0.5 * K * np.sum((np.diff(u, axis=1))**2, axis=1)
    return kinetic_energy + potential_energy

rk4_energy_values = compute_energy(u_num, v_num, K, m)

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.plot(time_points, rk4_energy_values, label="Total Energy (RK4 Method)")
plt.title("Energy Conservation Using RK4 Method")
plt.xlabel("Time")
plt.ylabel("Total Energy")
plt.legend()
plt.grid()
plt.show()

def compute_energy_analytic(u_analytic, K, m, time_points):
    v_analytic = np.zeros_like(u_analytic)
    for t_idx, t in enumerate(time_points):
        for j in range(1, N + 1):
            v_analytic[t_idx, j - 1] = np.sum([
                A_k[k] * modes[k, j - 1] * frequencies[k] * np.cos(frequencies[k] * t)
                for k in range(N)
            ])
```

Problem 4 continued on next page...

```
    kinetic_energy = 0.5 * m * np.sum(v_analytic**2, axis=1)
    potential_energy = 0.5 * K * np.sum((np.diff(u_analytic, axis=1))**2, axis=1)
    return kinetic_energy + potential_energy

analytic_energy_values = compute_energy_analytic(u_analytic, K, m, time_points)

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.plot(time_points, analytic_energy_values, label="Total Energy (Analytical Solution)")
plt.title("Energy Conservation Using Analytical Solution")
plt.xlabel("Time")
plt.ylabel("Total Energy")
plt.legend()
plt.grid()
plt.show()
```