Homework 5 of Computational Mathematics*

Chang, Yung-Hsuan

111652004

Department of Applied Mathematics

May 24, 2024

*For further information (such as codes), please refer to https://github.com/eiken59/CM_HW/tree/main/HW5.

Problem 1. Show that each of the following initial-value problems has a unique solution and find the solution. Can Theorem 5.4 be applied in each case?

a. $y' = t^{-2}(\sin 2t - 2ty), \quad 1 \le t \le 2, \quad y(1) = 2$; and

b. $y' = -y + t\sqrt{y}, \quad 2 \le t \le 3, \quad y(2) = 2$.

Solution.

a. The domain of $f(t, y) = \dfrac{\sin 2t - 2ty}{t^2}$ is $D = [1, 2] \times \mathbb{R}$. It is clear that f is continuous on D. Fix a $t \in [1, 2]$. Then, for $y_1, y_2 \in \mathbb{R}$,

$$\begin{aligned}|f(t, y_1) - f(t, y_2)| &= \frac{2t|y_1 - y_2|}{t^2} \\ &\le 2|y_1 - y_2|,\end{aligned}$$

which implies f satisfies a Lipschitz condition in the variable y on D with a Lipschitz constant 2. By Theorem 5.4, the initial-value problem has a unique solution. Using calculus, we have

$$\begin{aligned}t^2 y' + 2ty &= \sin 2t \\ t^2 y &= -\frac{1}{2}\cos 2t + C \\ \overset{y(1)=2}{\Longrightarrow} \qquad y &= \frac{-\cos 2t + 4 + \cos 2}{2t^2}.\end{aligned}$$

b. We can find that it is a Bernoulli's equation in the form of

$$y' + (1) \cdot y = t \cdot y^{1/2}.$$

Hence, we can find the solution as follows:

$$\begin{aligned}y' + (1) \cdot y &= t \cdot y^{1/2} \\ y' y^{-1/2} + y^{1/2} &= t \\ (\text{Let } u = y^{1/2}) \qquad 2u' + u &= t.\end{aligned}$$

It is clear that the homogeneous solution is $u_h = Ce^{-t/2}$. The non-homogeneous solution will be $u_n = -t + 2$. Hence, it has a unique solution; the general solution of the original equation is

$y_g = (Ce^{-t/2} - t + 2)^2$. By the assumption of the initial valule, we have $y = \left(e\sqrt{2} \cdot e^{-t/2} - t + 2\right)^2$. Since it does not satisfies a Lipschitz condition on $D = [2,3] \times \mathbb{R}$ (here $y \geq 0$), Theorem 5.4 cannot be applied in this case. $\square$

Problem 2. For each choice of $f(t, y)$ given in parts (a)-(d):

i. Does f satisfy a Lipschitz condition on $D = \{(t, y) \mid 0 \leq t \leq 1, -\infty < y < \infty\}$?

ii. Can Theorem 5.6 be used to show that the initial-value problem

$$y' = f(t, y), \quad 0 \leq t \leq 1, \quad y(0) = 1$$

is well-posed?

a. $f(t, y) = e^{t-y}$; and

b. $f(t, y) = \dfrac{1 + y}{1 + t}$.

Solution.

a. Fix a $t \in [0, 1]$. Then, for $y_1, y_2 \in \mathbb{R}$ with $y_1 > y_2$,

$$\begin{aligned} \left|e^{t-y_1} - e^{t-y_2}\right| &\geq \left|e^{-y_1} - e^{-y_2}\right| \\ &\geq e^{-y_2} \end{aligned}$$

is unbounded, which implies that f does not satisfy a Lipschitz condition on D in the variable y.

We cannot use Theorem 5.6 here since f does not satisfy a Lipschitz condition.

b. Fix a $t \in [0, 1]$. Then, for $y_1, y_2 \in \mathbb{R}$,

$$\left|\frac{1 + y_1}{1 + t} - \frac{1 + y_2}{1 + t}\right| \leq |y_1 - y_2|,$$

which implies f satisfies a Lipschitz condition in the variable y on D with Lipschitz constant 1.

Since f is continuous on D, by Theorem 5.6, the initial-value problem is well-posed. □

Problem 3. Use Euler's method to approximate the solutions for each of the following initial-value problems.

a. $y' = \dfrac{2-2ty}{t^2+1}, \quad 0 \le t \le 1, \quad y(0) = 1$ with $h = 0.1$; and

b. $y' = \dfrac{y^2}{1+t}, \quad 1 \le t \le 2, \quad y(1) = -\dfrac{1}{\ln 2}$ with $h = 0.1$.

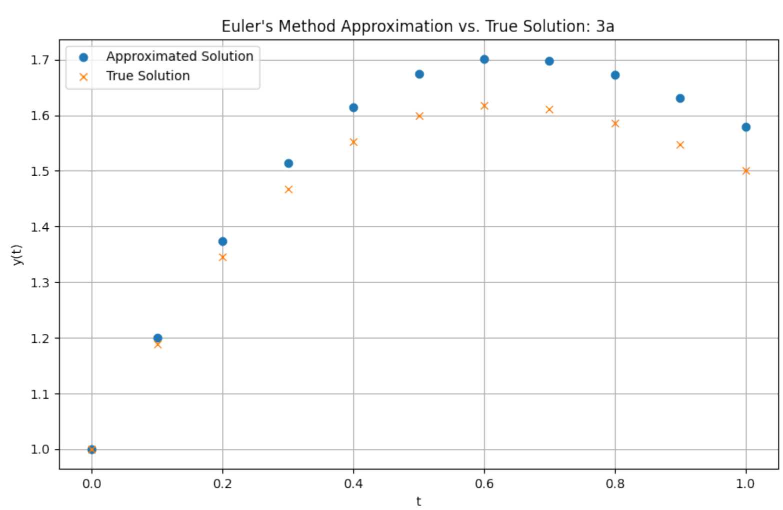
Show that the actual solutions are indeed $y(t) = \dfrac{2t+1}{t^2+1}$ and $y(t) = \dfrac{-1}{\ln(t+1)}$, respectively. Plot the errors between your numerical solutions and the exact solutions. Draw your conclusion regarding to the order of error with respect to the time step dt.

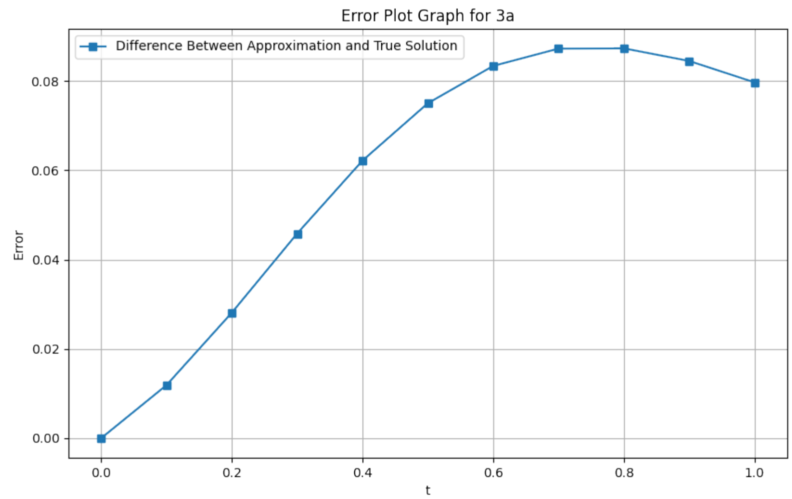
Solution.

a. We directly differentiate the function and see whether it is a solution. Let $y(t) = \dfrac{2t+1}{t^2+1}$. It is clear that $y(0) = 1$, and we have

$$\begin{aligned} y'(t) &= \frac{2(t^2+1)-(2t+1)(2t)}{(t^2+1)^2} \\ &= \frac{2}{t^2+1} - \frac{2t}{t^2+1}\frac{2t+1}{t^2+1} \\ &= \frac{2-2ty}{t^2+1}. \end{aligned}$$

The graph of the approximated solution and the actual solution can be seen below. The graph of error can also be seen below.

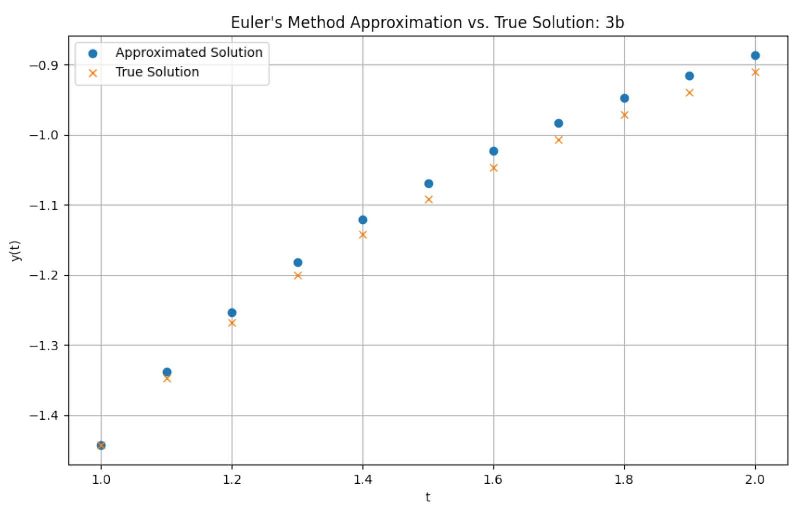


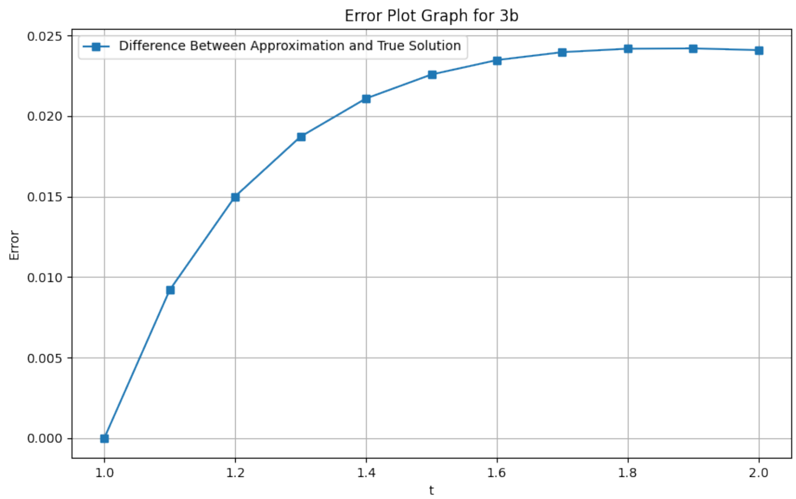


b. We directly differentiate the function and see whether it is a solution. Let $y(t) = \dfrac{-1}{\ln(t+1)}$. It is clear that $y(1) = -\dfrac{1}{\ln 2}$, and we have

$$\begin{aligned} y'(t) &= \frac{1/(t+1)}{(\ln(t+1))^2} \\ &= \frac{y^2}{1+t}. \end{aligned}$$

The graph of the approximated solution and the actual solution can be seen below. The graph of error can also be seen below.





Error Plot Graph for 3b
Difference Between Approximation and True Solution
Error
t
0.025
0.020
0.015
0.010
0.005
0.000
1.0
1.2
1.4
1.6
1.8
2.0

$\square$

```python
import math
import matplotlib.pyplot as plt

def plot_arrays(x_axis, y_1s, y_2s, title):
    plt.figure(figsize=(10, 6))
    plt.plot(x_axis, y_1s, 'o', label='Approximated Solution')
    plt.plot(x_axis, y_2s, 'x', label='True Solution')

    plt.xlabel('t')
    plt.ylabel('y(t)')
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.savefig(f"P{title[-2:]}.png", transparent=True)

def plot_errors(x_axis, y_1s, y_2s, title):
    plt.figure(figsize=(10, 6))
    y_diff = [abs(y1 - y2) for y1, y2 in zip(y_1s, y_2s)]
    plt.plot(x_axis, y_diff, 's-', label='Difference Between Approximation and True Solution')

    for i in range(len(x_axis)):
        plt.annotate(f'{abs(y_1s[i] - y_2s[i]):.2e}', (x_axis[i], min(y_1s[i], y_2s[i])),
                     textcoords="offset points", xytext=(0,-15), ha='center')

    plt.xlabel('t')
    plt.ylabel('Error')
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.savefig(f"P{title[-2:]}e.png", transparent=True)

def euler_method(f, true_y, alpha, a, b, h):
    N = int((b - a) / h)
    y_0 = alpha
    approx_soln_list = [y_0]
    real_soln_list = [true_y(a)]
    t_values = [a]

    for i in range(1, N + 1):
        t_i = a + i * h
        t_values.append(t_i)
        y_0 += h * f(t_values[-2], y_0)
        approx_soln_list.append(y_0)
        real_soln_list.append(true_y(t_i))

    return t_values, approx_soln_list, real_soln_list

def f_a(t, y):
    return (2 - 2 * t * y) / (t * t + 1)

def true_y_a(t):
    return (2 * t + 1) / (t * t + 1)

def f_b(t, y):
    return y * y / (1 + t)

def true_y_b(t):
    return -1 / math.log(t + 1)


t_values_a, approx_soln_a, real_soln_a = euler_method(f_a, true_y_a, 1, 0, 1, 0.1)

plot_arrays(t_values_a, approx_soln_a, real_soln_a, "Euler's Method Approximation vs. True Solution: 3a")
plot_errors(t_values_a, approx_soln_a, real_soln_a, "Error Plot Graph for 3a")


t_values_b, approx_soln_b, real_soln_b = euler_method(f_b, true_y_b, -1 / math.log(2), 1, 2, 0.1)

plot_arrays(t_values_b, approx_soln_b, real_soln_b, "Euler's Method Approximation vs. True Solution: 3b")
plot_errors(t_values_b, approx_soln_b, real_soln_b, "Error Plot Graph for 3b")
```

The code for Problem 3

Problem 4. Given the initial-value problem

$$y' = -y + t + 1, \quad 0 \le t \le 5, \quad y(0) = 1$$

with exact solution $y(t) = e^{-t} + t$.

a. Approximate $y(5)$ using Euler's method with $h = 0.2$, $h = 0.1$, and $h = 0.05$.

b. Determine the optimal value of h to use in computing $y(5)$, assuming $\delta = 10^{-6}$ and that Eq. (5.14) is valid.

Solution.

a. Using Euler's method with

$$\omega_{i+1} = \omega_i + h\,(-y + hi + 1), \quad i = 1, 2, \ldots, \frac{5-0}{h}.$$

By Python, we have

$$y(5) \approx 5.003777893186297 \quad \text{when } h = 0.2;$$

$$y(5) \approx 5.005153775207321 \quad \text{when } h = 0.1;$$

$$y(5) \approx 5.005920529220334 \quad \text{when } h = 0.05;$$

b. By assuming (5.14) is true and $\delta = 10^{-6}$, the minimal value of $E(h)$ occurs when

$$h = \sqrt{\frac{2 \cdot 10^{-6}}{\max_{t \in [0,5]} |e^{-t}|}} = \sqrt{2 \cdot 10^{-6}} \approx 1.4142135623731 \times 10^{-3}.$$

□

```python
def euler_method(f, h, a, b):
    y_0 = 1
    N = int((b - a) / h)
    for i in range(N):
        y_0 += h * f(a + h * i, y_0)
    return y_0

def f_4(t, y):
    return -y + t + 1

for h in [0.2, 0.1, 0.05]:
    print(f"With h = {h}, the approximation is {euler_method(f_4, h, 0, 5)}.")
```

The code for Problem 4a

Problem 5. Use Taylor's method of order two to approximate the solutions for each of the following initial-value problems.

a. $y' = \dfrac{1+t}{1+y}, \quad 1 \le t \le 2, \quad y(1) = 2$ with $h = 0.5$; and

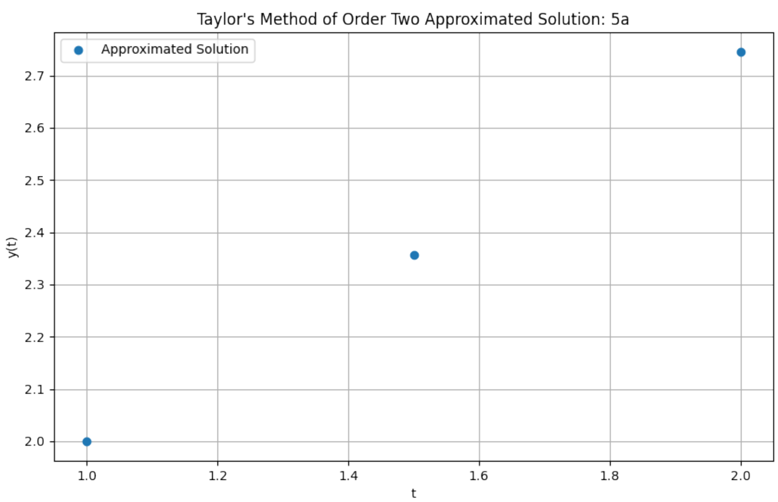
b. $y' = -y + t\sqrt{y}, \quad 2 \le t \le 3, \quad y(2) = 2$ with $h = 0.25$.

Solution.

a. By calculus,

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{1+t}{1+y} = \frac{1}{1+y} - \frac{(1+t)^2}{(1+y)^3}.$$

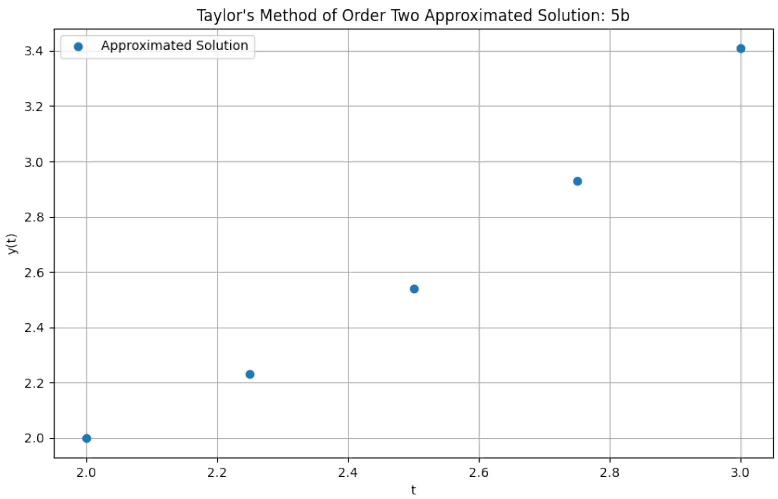
By Python, we have the following result:



b. By calculus,

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(-y + t\sqrt{y}\right) = y + \sqrt{y} - \frac{3}{2}t\sqrt{y} + \frac{t^2}{2}.$$

By Python, we have the following result:



□

```
import math
import matplotlib.pyplot as plt

def plot_array(x_axis, y_axis, title):
    plt.figure(figsize=(10, 6))
    plt.plot(x_axis, y_axis, 'o', label='Approximated Solution')
    plt.xlabel('t')
    plt.ylabel('y(t)')
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.savefig(f"P{title[-2:]}.png", transparent=True)

def taylor_method(n, f_family, alpha, a, b, h):
    N = int((b - a) / h)
    y_0 = alpha
    approx_soln_list = [y_0]
    t_values = [a]

    for i in range(1, N + 1):
        T = 0
        for ii in range(n):
            T += h ** ii * f_family[ii](t_values[-1], approx_soln_list[-1]) / math.factorial(ii + 1)
        approx_soln_list.append(approx_soln_list[-1] + h * T)
        t_values.append(a + h * i)

    return t_values, approx_soln_list

def f_a(t, y):
    return (1 + t) / (1 + y)

def f_b(t, y):
    return -y + t * y ** 0.5

def Df_a(t, y):
    return 1 / (1 + y) - (1 + t) ** 2 / (1 + y) ** 3

def Df_b(t, y):
    return y + y * 0.5 - 3 * t * y ** 0.5 / 2 + t ** 2 / 2

f_a_family = [f_a, Df_a]
f_b_family = [f_b, Df_b]


t_values_a, approx_soln_a = taylor_method(2, f_a_family, 2, 1, 2, 0.5)

plot_array(t_values_a, approx_soln_a, "Taylor's Method of Order Two Approximated Solution: 5a")

t_values_b, approx_soln_b = taylor_method(2, f_b_family, 2, 2, 3, 0.25)

plot_array(t_values_b, approx_soln_b, "Taylor's Method of Order Two Approximated Solution: 5b")
```

The code for Problem 5

Problem 6. Given the initial-value problem

$$y' = \frac{2}{t}y + t^2 e^t, \quad 1 \leq t \leq 2, \quad y(1) = 0$$

with exact solution $y(t) = t^2(e^t - e)$.

a. Use Taylor's method of order two with $h = 0.1$ to approximate the solution, and compare it with the actual values of y.

b. Use the answers generated in part (a) and linear interpolation to approximate y at the following values, and compare them to the actual values of y.

   i. $y(1.04)$;

   ii. $y(1.55)$; and

   iii. $y(1.97)$.

c. Use Taylor's method of order four with $h = 0.1$ to approximate the solution, and compare it with the actual values of y.

d. Use the answers generated in part (c) and piecewise cubic Hermite interpolation to approximate y at the following values, and compare them to the actual values of y.

   i. $y(1.04)$;

   ii. $y(1.55)$; and

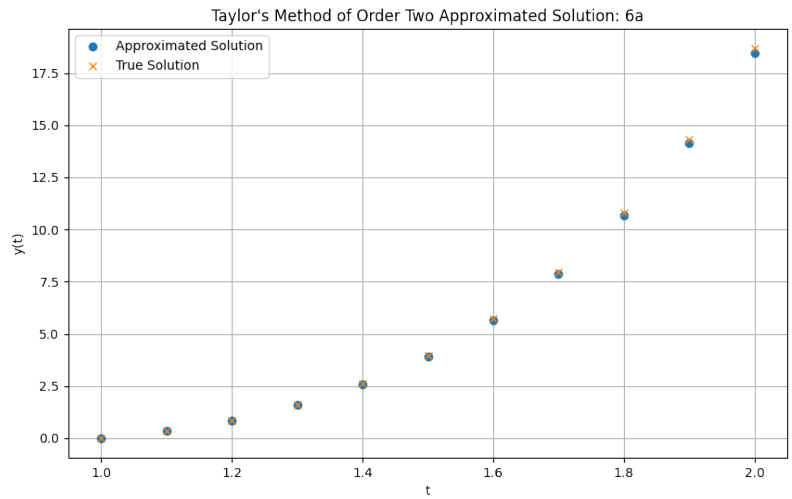
   iii. $y(1.97)$.

Solution.

a. By calculus,

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{2}{t}y + t^2 e^t\right) &= -\frac{2}{t^2}y + \frac{2}{t}y' + 2te^t + t^2 e^t \\ &= \frac{2y}{t^2} + (4t + t^2)e^t.\end{aligned}$$

By Python, we have the following result:



b. By (a), we have the following table:

t	Approximation	Real Value
1	0	0
1.1	0.3397852286	0.3459198765
1.2	0.8521434493	0.8666425358
1.3	1.5817695052	1.6072150782
1.4	2.5809966497	2.6203595512
1.5	3.9109845593	3.9676662942
1.6	5.6430810358	5.7209615256
1.7	7.8603816039	7.9638734778
1.8	10.6595144804	10.7936246605
1.9	14.1526820904	14.3230815359
2.0	18.4699944826	18.6830970819

Using linear interpolation, we have

$$
\begin{aligned}
y(1.04) &= 0.4 \cdot y(1.0) + 0.6 \cdot y(1.1) \\
&= 0.2038711371, \\
y(1.55) &= 0.5 \cdot y(1.5) + 0.5 \cdot y(1.6) \\
&= 4.7770327976, \\
y(1.97) &= 0.3 \cdot y(1.9) + 0.7 \cdot y(2.0) \\
&= 17.1748007649,
\end{aligned}
$$

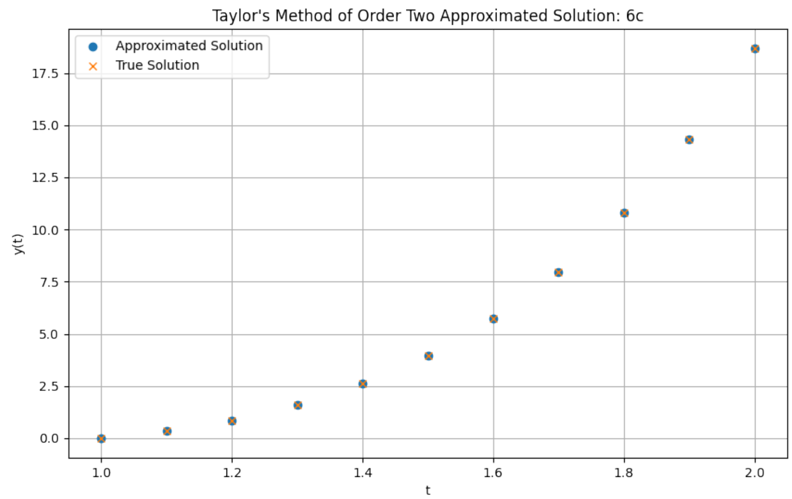
where the real values are

$$
\begin{aligned}
y(1.04) &= 0.2075519259, \\
y(1.55) &= 4.8443139099, \\
y(1.97) &= 17.3750924181.
\end{aligned}
$$

c. By calculus,

$$
\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{2y}{t^2} + (4t + t^2)e^t\right) &= \frac{-4y}{t^3} + \frac{2y'}{t^2} + (t^2 + 6t + 4)e^t \\
&= (t^2 + 6t + 6)e^t, \\
\frac{\mathrm{d}}{\mathrm{d}t}\left((t^2 + 6t + 6)e^t\right) &= (t^2 + 8t + 12)e^t.
\end{aligned}
$$

By Python, we have the following result:



d. By (c), we have the following table:

t	Approximation	Real Value
1	0	0
1.1	0.3459126888	0.3459198765
1.2	0.8666257293	0.8666425358
1.3	1.6071858864	1.6072150782
1.4	2.6203148428	2.6203595512
1.5	3.9676025389	3.9676662942
1.6	5.7208747556	5.7209615256
1.7	7.9637592441	7.9638734778
1.8	10.7934779832	10.7936246605
1.9	14.3228968484	14.3230815359
2.0	18.6828681680	18.6830970819

Using linear interpolation, we have

$$
\begin{aligned}
y(1.04) &= 0.4 \cdot y(1.0) + 0.6 \cdot y(1.1) \\
&= 0.2075476133, \\
y(1.55) &= 0.5 \cdot y(1.5) + 0.5 \cdot y(1.6) \\
&= 4.8442386472, \\
y(1.97) &= 0.3 \cdot y(1.9) + 0.7 \cdot y(2.0) \\
&= 17.3748767721,
\end{aligned}
$$

where the real values are

$$
\begin{aligned}
y(1.04) &= 0.2075519259, \\
y(1.55) &= 4.8443139099, \\
y(1.97) &= 17.3750924181.
\end{aligned}
$$

$\square$

```python
import math
import matplotlib.pyplot as plt

def plot_arrays(x_axis, y_1s, y_2s, title):
    plt.figure(figsize=(10, 6))
    plt.plot(x_axis, y_1s, 'o', label='Approximated Solution')
    plt.plot(x_axis, y_2s, 'x', label='True Solution')

    plt.xlabel('t')
    plt.ylabel('y(t)')
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.savefig(f"P{title[-2:]}.png", transparent=True)

def taylor_method(n, f_family, real_y, alpha, a, b, h):
    N = int((b - a) / h)
    y_0 = alpha
    approx_soln_list = [y_0]
    real_soln_list = [y_0]
    t_values = [a]

    for i in range(1, N + 1):
        T = 0
        for ii in range(n):
            T += h ** ii * f_family[ii](t_values[-1], approx_soln_list[-1]) / math.factorial(ii + 1)
        approx_soln_list.append(approx_soln_list[-1] + h * T)
        t_values.append(a + h * i)
        real_soln_list.append(real_y(a + h * i))

    return t_values, approx_soln_list, real_soln_list

def f(t, y):
    return 2 * y / t + t ** 2 * math.exp(t)

def Df(t, y):
    return 2 * y / t ** 2 + (4 * t + t ** 2) * math.exp(t)

def D2f(t, y):
    return (t ** 2 + 6 * t + 6) * math.exp(t)

def D3f(t, y):
    return (t ** 2 + 8 * t + 12) * math.exp(t)

def y(t):
    return t ** 2 * (math.exp(t) - math.e)

f_family = [f, Df, D2f, D3f]


t_values_a, approx_soln_a, real_soln_a = taylor_method(2, f_family, y, 0, 1, 2, 0.1)

plot_arrays(t_values_a, approx_soln_a, real_soln_a, "Taylor's Method of Order Two Approximated Solution: 6a")

for i, j, k in zip(t_values_a, approx_soln_a, real_soln_a):
    print(f"${i :.1f}$ & ${j :.10f}$ & ${k :.10f}$ \\\\")
    print("\hline")

t_values_c, approx_soln_c, real_soln_c = taylor_method(4, f_family, y, 0, 1, 2, 0.1)

plot_arrays(t_values_c, approx_soln_c, real_soln_c, "Taylor's Method of Order Two Approximated Solution: 6c")

for i, j, k in zip(t_values_c, approx_soln_c, real_soln_c):
    print(f"${i :.1f}$ & ${j :.10f}$ & ${k :.10f}$ \\\\")
    print("\hline")

for soln_list in [approx_soln_a, approx_soln_c, real_soln_c]:
    print(f"{0.4 * soln_list[0] + 0.6 * soln_list[1] :.10f}")
    print(f"{0.5 * soln_list[5] + 0.5 * soln_list[6] :.10f}")
    print(f"{0.3 * soln_list[9] + 0.7 * soln_list[10] :.10f}")
```

The code for Problem 6

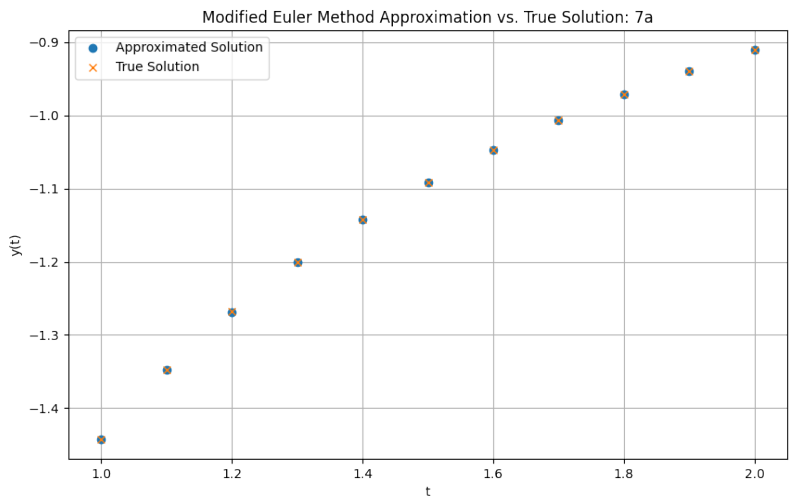
Problem 7. Use the modified Euler method to approximate the solutions to each of the following initial-value problems, and compare the results to the actual values.

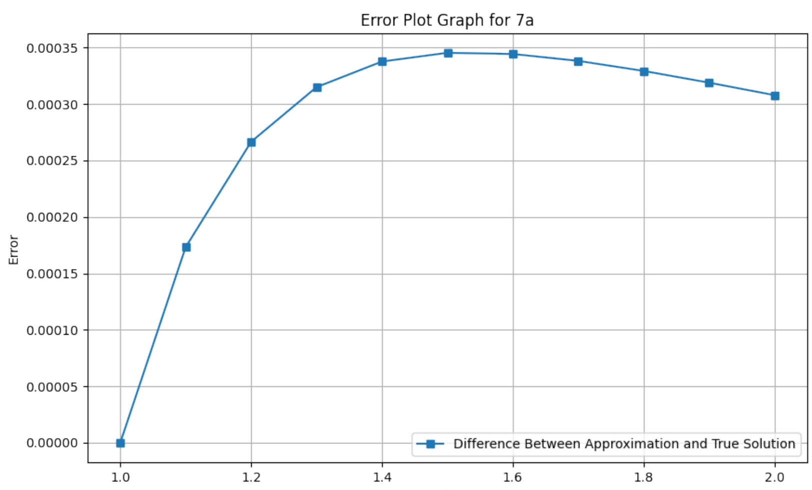
a. $y' = \dfrac{y^2}{1+t}$, $\quad 1 \le t \le 2$, $\quad y(1) = -\dfrac{1}{\ln 2}$ with $h = 0.1$; actual solution $y = \dfrac{-1}{\ln(t+1)}$; and

b. $y' = \dfrac{y^2+y}{t}$, $\quad 1 \le t \le 3$, $\quad y(1) = -2$ with $h = 0.2$; actual solution $y(t) = \dfrac{2t}{1-2t}$.

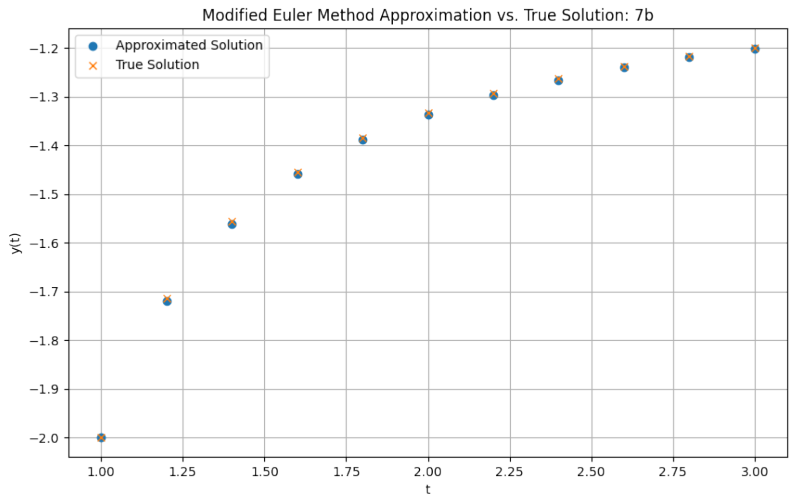
Solution. By Python, we have the following results:

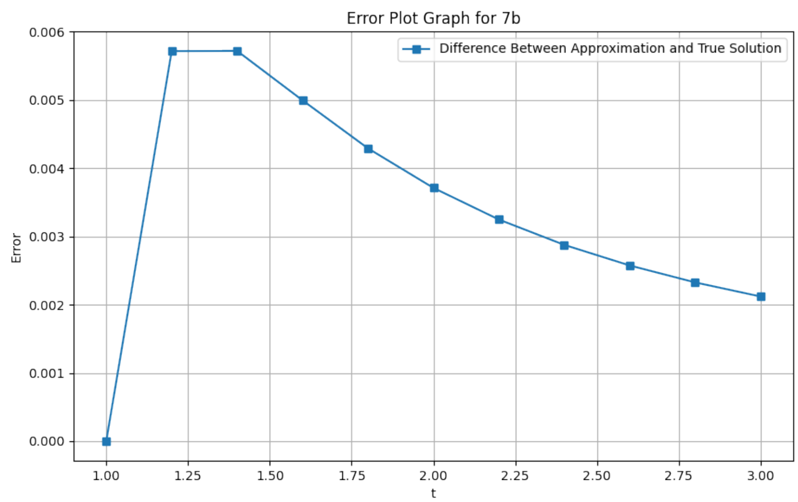
a. The first graph is with approximated solution and the actual solution; the second graph is the error plot graph. In the error plot graph, I used lines to connect each pair of adjacent dots to increase visual clarity.





b. The first graph is with approximated solution and the actual solution; the second graph is the error plot graph. In the error plot graph, I used lines to connect each pair of adjacent dots to increase visual clarity.





□

```python
import math
import matplotlib.pyplot as plt

def plot_arrays(x_axis, y_1s, y_2s, title):
    plt.figure(figsize=(10, 6))
    plt.plot(x_axis, y_1s, 'o', label='Approximated Solution')
    plt.plot(x_axis, y_2s, 'x', label='True Solution')
    plt.xlabel('t')
    plt.ylabel('y(t)')
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.savefig(f"P{title[-2:]}.png", transparent=True)

def plot_errors(x_axis, y_1s, y_2s, title):
    plt.figure(figsize=(10, 6))
    y_diff = [abs(y1 - y2) for y1, y2 in zip(y_1s, y_2s)]
    plt.plot(x_axis, y_diff, 's-', label='Difference Between Approximation and True Solution')

    plt.xlabel('t')
    plt.ylabel('Error')
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.savefig(f"P{title[-2:]}e.png", transparent=True)

def modified_euler_method(f, true_y, alpha, a, b, h):
    N = int((b - a) / h)
    y_0 = alpha
    approx_soln_list = [y_0]
    real_soln_list = [true_y(a)]
    t_values = [a]

    for i in range(1, N + 1):
        t_i = a + i * h
        t_values.append(t_i)
        y_0 += h * (f(t_values[-2], y_0) + f(t_i, y_0 + h * f(t_values[-2], y_0))) / 2
        approx_soln_list.append(y_0)
        real_soln_list.append(true_y(t_i))

    return t_values, approx_soln_list, real_soln_list

def f_a(t, y):
    return y * y / (1 + t)

def true_y_a(t):
    return -1 / math.log(t + 1)

def f_b(t, y):
    return (y * y + y) / t

def true_y_b(t):
    return 2 * t / (1 - 2 * t)


t_values_a, approx_soln_a, real_soln_a = modified_euler_method(f_a, true_y_a, -1 / math.log(2), 1, 2, 0.1)

plot_arrays(t_values_a, approx_soln_a, real_soln_a, "Modified Euler Method Approximation vs. True Solution: 7a")
plot_errors(t_values_a, approx_soln_a, real_soln_a, "Error Plot Graph for 7a")

t_values_b, approx_soln_b, real_soln_b = modified_euler_method(f_b, true_y_b, -2, 1, 3, 0.2)

plot_arrays(t_values_b, approx_soln_b, real_soln_b, "Modified Euler Method Approximation vs. True Solution: 7b")
plot_errors(t_values_b, approx_soln_b, real_soln_b, "Error Plot Graph for 7b")
```

The code for Problem 7

Problem 8. Show that the difference method

$$\omega_0 = \alpha$$

$$\omega_{i+1}\omega_i + a_1\, f(t_i, \omega_i) + a_2\, f(t_i + \alpha_2, \omega_1 + \delta_2\, f(t_i, \omega_i)),$$

for each $i = 0, 1, 2, \ldots, N-1$, cannot have local truncation error $\mathcal{O}(h^3)$ for any choice of constants a_1, a_2, α_2, and δ_2.

Solution.

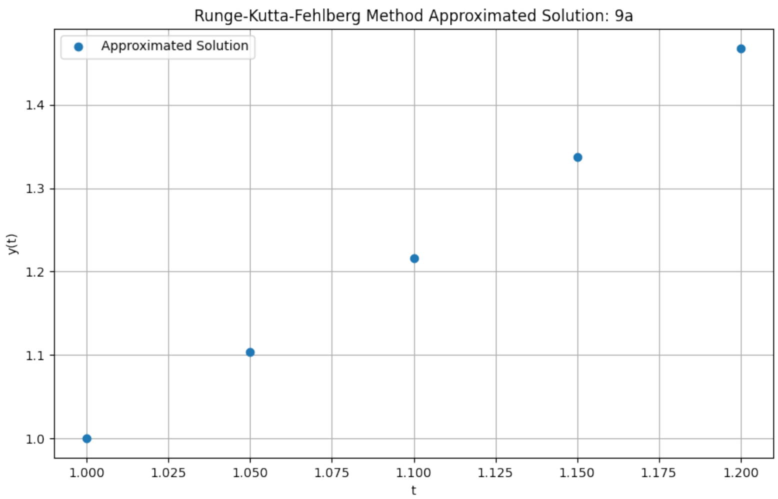
Problem 9. Use the Runge-Kutta-Fehlberg algorithm with tolerance $TOL = 10^{-4}$ to approximate the solution to the following initial-value problems.

a. $y' = \left(\frac{y}{t}\right)^2 + \frac{y}{t}, \quad 1 \le t \le 1.2, \quad y(1) = 1$ with $hmax = 0.05$ and $hmin = 0.02$; and

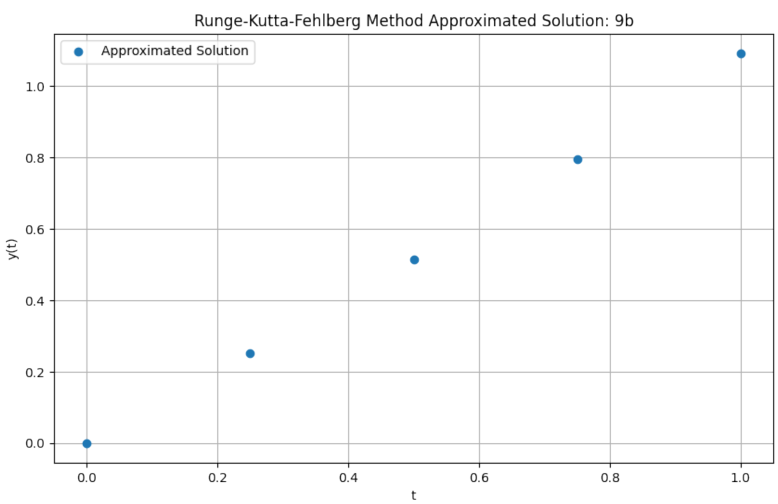
b. $y' = \sin t + e^{-t}, \quad 0 \le t \le 1, \quad y(0) = 0$ with $hmax = 0.25$ and $hmin = 0.02$.

Solution. The code is provided after the two graphs.

a.



b.



□

```python
import math
import matplotlib.pyplot as plt

def plot_array(x_axis, y_axis, title):
    plt.figure(figsize=(10, 6))
    plt.plot(x_axis, y_axis, 'o', label='Approximated Solution')
    plt.xlabel('t')
    plt.ylabel('y(t)')
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.savefig(f"P{title[-2:]}.png", transparent=True)

def runge_kutta_fehlberg_method(f, alpha, TOL, a, b, hmax, hmin):
    h = hmax
    y_0 = alpha
    approx_soln_list = [y_0]
    t_values = [a]

    FLAG = True

    while(FLAG):
        t = t_values[-1]
        k_1 = h * f(t, y_0)
        k_2 = h * f(t + h / 4, y_0 + k_1 / 4)
        k_3 = h * f(t + 3 * h / 8, y_0 + 3 * k_1 / 32 + 9 * k_2 / 32)
        k_4 = h * f(t + 12 * h / 13, y_0 + 1932 * k_1 / 2197 - 7200 * k_2 / 2197 + 7296 * k_3 / 2197)
        k_5 = h * f(t + h, y_0 + 439 * k_1 / 216 - 8 * k_2 + 3680 * k_3 / 513 - 845 * k_4 / 4104)
        k_6 = h * f(t + h / 2, y_0 - 8 * k_1 / 27 + 2 * k_2 - 3544 * k_3 / 2565 + 1859 * k_4 / 4104 - 11 * k_5 / 40)

        R = abs(k_1 / 360 - 128 * k_3 / 4275 - 2197 * k_4 / 75240 + k_5 / 50 + 2 * k_6 / 55) / h

        if R <= TOL:
            t += h
            t_values.append(t)
            y_0 += 25 * k_1 / 216 + 1408 * k_3 / 2565 + 2197 * k_4 / 4104 - k_5 / 5.
            approx_soln_list.append(y_0)

        delta = 0.84 * pow(TOL / R, 0.25)
        if delta <= 0.1:
            h = 0.1 * h
        elif delta >= 4:
            h = 4 * h
        else:
            h = delta * h

        if h > hmax:
            h = hmax

        if t >= b:
            FLAG = False
        elif t + h > b:
            h = b - t
        elif h < hmin:
            FLAG = False
            print("Minimum h exceeded.")

    return t_values, approx_soln_list

def f_a(t, y):
    return y * y / (t * t) + y / t

def f_b(t, y):
    return math.sin(t) + math.exp(-t)

t_values_a, approx_soln_a = runge_kutta_fehlberg_method(f_a, 1, 10 ** -4, 1, 1.2, 0.05, 0.02)

print(approx_soln_a)
plot_array(t_values_a, approx_soln_a, "Runge-Kutta-Fehlberg Method Approximated Solution: 9a")

t_values_b, approx_soln_b = runge_kutta_fehlberg_method(f_b, 0, 10 ** -4, 0, 1, 0.25, 0.02)

print(approx_soln_b)
plot_array(t_values_b, approx_soln_b, "Runge-Kutta-Fehlberg Method Approximated Solution: 9b")
```

The code for Problem 9

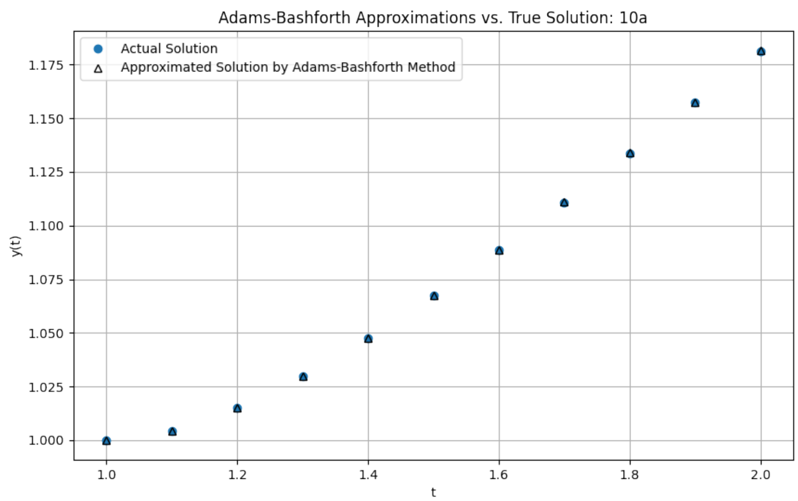
Problem 10. Use each of the Adams-Bashforth methods to approximate the solutions to the following initial-value problems. In each case use starting values obtained from the Runge-Kutta method of order four. Compare the results to the actual values.

a. $y' = \dfrac{y}{t} - \left(\dfrac{y}{t}\right)^2, \quad 1 \le t \le 2, \quad y(1) = 1$ with $h = 0.1$; actual solution $y(t) = \dfrac{t}{1 + \ln(t)}$; and

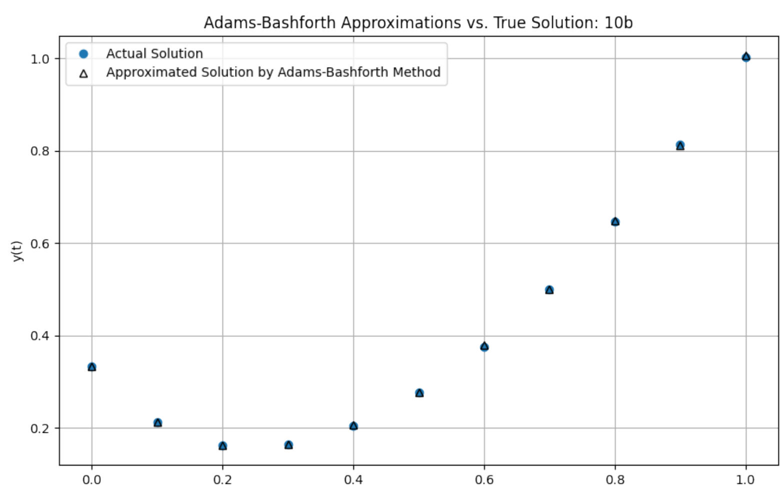
b. $y' = -5y + 5t^2 + 2t, \quad 0 \le t \le 1, \quad y(0) = \dfrac{1}{3}$ with $h = 0.1$; actual solution $y(t) = t^2 + \dfrac{e^{-5t}}{3}$.

Solution. The code is provided after the two graphs.

a.



b.



□

```python
import math
import matplotlib.pyplot as plt

def plot_arrays(x_axis, y_1s, y_2s, title):
    plt.figure(figsize=(10, 6))
    plt.plot(x_axis, y_2s, 'o', label='Actual Solution')
    plt.plot(x_axis, y_1s, '^', color="black", markerfacecolor='none', label='Approximated Solution by Adams-Bashforth Method')

    plt.xlabel('t')
    plt.ylabel('y(t)')
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.savefig(f"P{title[-3:]}.png", transparent=True)

def adams_bashforth_method(f, true_y, alpha, a, b, h):
    N = int((b - a) / h)
    approx_soln_list = alpha.copy()
    real_soln_list = alpha.copy()
    t_values = [a, a + h, a + 2 * h, a + 3 * h]

    for i in range(4, N + 1):
        w = approx_soln_list[-4:]
        t = t_values[-4:]
        approx_soln_list.append(w[3] + h * (55 * f(t[3], w[3]) - 59 * f(t[2], w[2]) + 37 * f(t[1], w[1]) - 9 * f(t[0], w[0])) / 24)
        t_i = a + i * h
        t_values.append(t_i)
        real_soln_list.append(true_y(t_i))

    return t_values, approx_soln_list, real_soln_list

def runge_kutta_method(f, true_y, alpha, a, b, h):
    N = int((b - a) / h)
    approx_soln_list = [alpha]
    real_soln_list = [alpha]
    t_values = [a]

    for i in range(1, N + 1):
        t = t_values[-1]
        w = approx_soln_list[-1]
        if i < 1:
            approx_soln_list.append(true_y(t))
        else:
            k_1 = h * f(t, w)
            k_2 = h * f(t + h / 2, w + k_1 / 2)
            k_3 = h * f(t + h / 2, w + k_2 / 2)
            k_4 = h * f(t + h, w + k_3)
            approx_soln_list.append(w + (k_1 + 2 * k_2 + 2 * k_3 + k_4) / 6)
        t_i = a + i * h
        real_soln_list.append(true_y(t_i))
        t_values.append(t_i)

    return t_values, approx_soln_list, real_soln_list

def f_a(t, y):
    return y / t - y * y / (t * t)

def true_y_a(t):
    return t / (1 + math.log(t))

def f_b(t, y):
    return -5 * y + 5 * t * t + 2 * t

def true_y_b(t):
    return t * t + math.exp(-5 * t) / 3

t_values_a, approx_soln_a_RK, real_soln_a = runge_kutta_method(f_a, true_y_a, 1, 1, 2, 0.1)

t_values_a, approx_soln_a_AB, real_soln_a = adams_bashforth_method(f_a, true_y_a, approx_soln_a_RK[0:4], 1, 2, 0.1)

plot_arrays(t_values_a, approx_soln_a_AB, real_soln_a, "Adams-Bashforth Approximations vs. True Solution: 10a")

t_values_b, approx_soln_b_RK, real_soln_b = runge_kutta_method(f_b, true_y_b, 1 / 3, 0, 1, 0.1)

t_values_b, approx_soln_b_AB, real_soln_b = adams_bashforth_method(f_b, true_y_b, approx_soln_b_RK[0:4], 0, 1, 0.1)

plot_arrays(t_values_b, approx_soln_b_AB, real_soln_b, "Adams-Bashforth Approximations vs. True Solution: 10b")
```

The code for Problem 10

Problem 11. The initial-value problem

$$y' = e^y, \quad 0 \le t \le 0.2, \quad y(0) = 1$$

has solution $y(t) = 1 - \ln(1 - et)$. Applying the three-step Adams-Moulton method to this problem is equivalent to finding the fixed point ω_{i+1} of

$$g(\omega) = \omega_i + \frac{h}{24}\left(9e^{\omega} + 19e^{\omega_i} - 5e^{\omega_{i-1}} + e^{\omega_{i-2}}\right).$$

a. With $h = 0.01$, obtain ω_{i+1} by functional iteration for $i = 2, \ldots, 19$ using exact starting values ω_0, ω_1, and ω_2. At each step use ω_i to initially approximate ω_{i+1}.

b. Will Newton's method speed the convergence over functional iteration?

Solution.

Problem 12. Derive the Adams-Bashforth three-step method by the following method. Set

$$y(t_{i+1}) = t(t_i) + ah\, f(t_i, y(t_i)) + bh\, f(t_{i-1}, y(t_{i-1})) + ch\, f(t_{i-2}, y(t_{i-2})).$$

Expand $y(t_{i+1})$, $f(t_{i-2}, y(t_{i-2}))$, and $f(t_{i-1}, y(t_{i-1}))$ in Taylor series about $(t_i, y(t_i))$, and equate the coefficients of h, h^2, and h^3 to obtain a, b, and c.

Solution.

Problem 13. Use the Adams variable step-size predictor-corrector algorithm with $TOL = 10^{-4}$ to approximate the solutions to the following initial-value problems:

a. $y' = \sin t + e^{-t}, \quad 0 \le t \le 1, \quad y(0) = 0$ with $hmax = 0.2$ and $hmin = 0.01$; and

b. $y' = -ty + \dfrac{4t}{y}, \quad 0 \le t \le 1, \quad y(0) = 1$ with $hmax = 0.2$ and $hmin = 0.01$.

Solution.

Problem 14. Let $P(t)$ be the number of individuals in a population at time t, measured in years. If the average birth rate b is constant and the average death rate d is proportional to the size of the population (due to overcrowding), then the growth rate of the population is given by the logistic equation

$$\frac{\mathrm{d}P}{\mathrm{d}t}(t) = b\,P(t) - k(P(t))^2,$$

where $d = k\,P(t)$. Suppose $P(0) = 50976$, $b = 2.9 \times 10^{-2}$, and $k = 1.4 \times 10^{-7}$. Find the population after 5 years using the extrapolation method (based on the Euler method and the midpoint method) with times step $h = 0.1$. Justify the order of truncation error from your numerical answers.

Solution.

Problem 15. Suppose the swinging pendulum described in the lead example of this chapter is 2 ft long and that $g = 32.17$ ft/s^2. With $h = 0.1$ s, compare the angle θ obtained for the following two initial-value problems at $t = 0, 1, 2$.[a]

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} + \frac{g}{L}\sin\theta = 0, \quad \theta(0) = \frac{\pi}{6}, \quad \theta'(0) = 0.$$

You shall use Adams fourth-order predictor-corrector algorithm to obtain your numerical answer.

[a] I read this as "find values of $\theta(1)$ and $\theta(2)$ given $\theta(0) = \pi/6$."

Solution.

Problem 16. Consider the differential equation

$$y' = f(t, y), \quad a \le t \le b, \quad y(a) = \alpha.$$

a. Show that

$$y'(t_i) = \frac{-3y(t_y) + 4y(t_{i+1} - y(t_{i+2}))}{2h} + \frac{h^2}{3} y'''(\xi_1)$$

for some $\xi \in (t_i, t_{i+2})$.

b. Part (a) suggests the difference method

$$\omega_{i+2} = 4\omega_{i+1} - 3\omega_i - 2h\, f(t_i, \omega_i), \quad \text{for } i = 0, 1, 2 \ldots, N-2.$$

Use this method to solve

$$y' - 1 - y, \quad 0 \le t \le 1, \quad y(0) = 0$$

with $h = 0.1$. Use the starting values $\omega_0 = 0$ and $\omega_1 = y(t_1) = 1 - e^{-0.1}$.

c. Repeat part (b) with $h = 0.01$ and $\omega_1 = 1 - e^{-0.01}$.

d. Analyze this method for consistency, stability, and convergence.

Solution.

Problem 17. Given the multistep method

$$\omega_{i+1} = -\frac{3}{2}\omega_i + 3\omega_{i-1} - \frac{1}{2}\omega_{i-2} + 3g\, f(t_i, \omega_i), \quad \text{for } i = 2, 3, \ldots, N-1$$

with starting values ω_0, ω_1, and ω_2:

a. Find the local truncation error.

b. Comment on consistency, stability, and convergence.

Solution.

Problem 18. Discuss consistency, stability, and convergence for the implicit trapezoidal method

$$\omega_{i+1} = \omega_i + \frac{h}{2}\left(f(t_{i+1}, \omega_{i+1}) + f(t_i, \omega_i)\right), \quad \text{for } i = 0, 1, 2, \ldots, N-1$$

with $\omega_0 = \alpha$ applied to the differential equation

$$y' = f(t, y), \quad a \leq t \leq b, \quad y(a) = \alpha.$$

Solution.

Problem 19. Show that the fourth-order Runge-Kutta method,

$$k_1 = h\, f(t_i, \omega_i),$$

$$k_2 = h\, f(t_i + h/2, \omega_i + k_1/2),$$

$$k_3 = h\, f(t_i + h/2, \omega_i + k_2/2),$$

$$k_4 = h\, f(t_i + h, \omega_i + k_3),$$

$$\omega_{i+1} = \omega_i + \frac{1}{6}\,(k_1 + 2k_2 + 2k_3 + k_4)$$

when applied to the differential equation $y' = \lambda y$, can be written in the form

$$\omega_{i+1} = \left(1 + h\lambda + \frac{1}{2}(h\lambda)^2 + \frac{1}{6}(h\lambda)^3 + \frac{1}{24}(h\lambda)^4\right)\omega_i.$$

Solution. We have $f(t, w) = \lambda w$. Hence, we have

$$
\begin{aligned}
k_1 &= h\lambda\omega_i,\\
k_2 &= h\lambda(\omega_i + k_1/2)\\
&= h\lambda\omega_i + (h\lambda)^2\omega_i/2,\\
k_3 &= h\lambda(\omega_i + k_2/2)\\
&= h\lambda\omega_i + h\lambda(h\lambda\omega_i + (h\lambda)^2\omega_i/2)/2\\
&= h\lambda\omega_i + (h\lambda)^2\omega_i/2 + (h\lambda)^3\omega_i/4,\\
k_4 &= h\lambda(\omega_i + k_3)\\
&= h\lambda\omega_i + (h\lambda)^2\omega_i + (h\lambda)^3\omega_i/2 + (h\lambda)^4\omega_i/4.
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
\omega_{i+1} &= \omega_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \\
&= \omega_i + \frac{1}{6}[(h\lambda\omega_i) \\
&\qquad\qquad + 2\left(h\lambda\omega_i + (h\lambda)^2\omega_i/2\right) \\
&\qquad\qquad + 2\left(h\lambda\omega_i + (h\lambda)^2\omega_i/2 + (h\lambda)^3\omega_i/4\right) \\
&\qquad\qquad + \left(h\lambda\omega_i + (h\lambda)^2\omega_i + (h\lambda)^3\omega_i/2 + (h\lambda)^4\omega_i/4\right)] \\
&= \omega_i + \frac{1}{6}\left(6 \cdot h\lambda\omega_i + 3 \cdot (h\lambda)^2\omega_i + 1 \cdot (h\lambda)^3\omega_i + \frac{1}{4} \cdot (h\lambda)^4\omega_i\right) \\
&= \omega_i\left(1 + h\lambda + \frac{1}{2}(h\lambda)^2 + \frac{1}{6}(h\lambda)^3 + \frac{1}{24}(h\lambda)^4\right).
\end{aligned}
$$

□