Homework 5 of Computational Mathematics[*]

Chang, Yung-Hsuan
111652004
Department of Applied Mathematics

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

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

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

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

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 the following:

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. $\square$

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 $h = 0.1$.

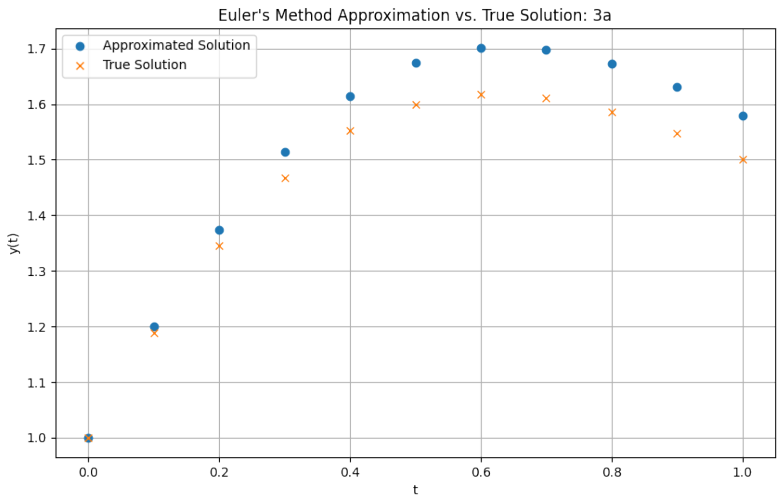
Show that the actual solutions are indeed $y(t) = \dfrac{2t+1}{t^2+1}$ and $y(t) = \dfrac{-1}{\ln(t+1)}$, respectively. Plot the errors between your numerical solutions and the exact solutions. Draw your conclusion regarding to the order of error with respect to the time step dt.

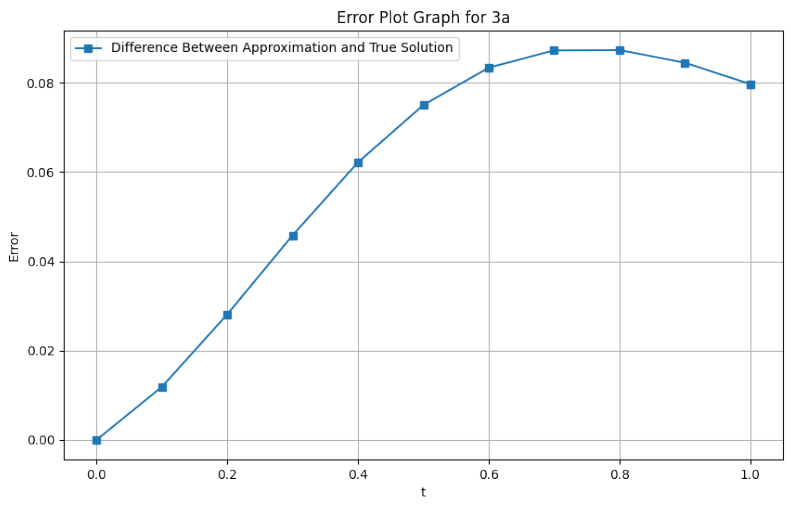
Solution.

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.

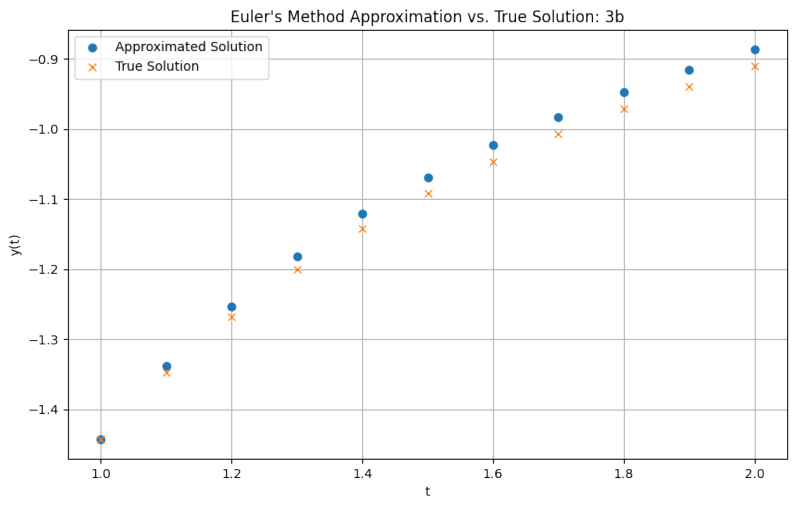


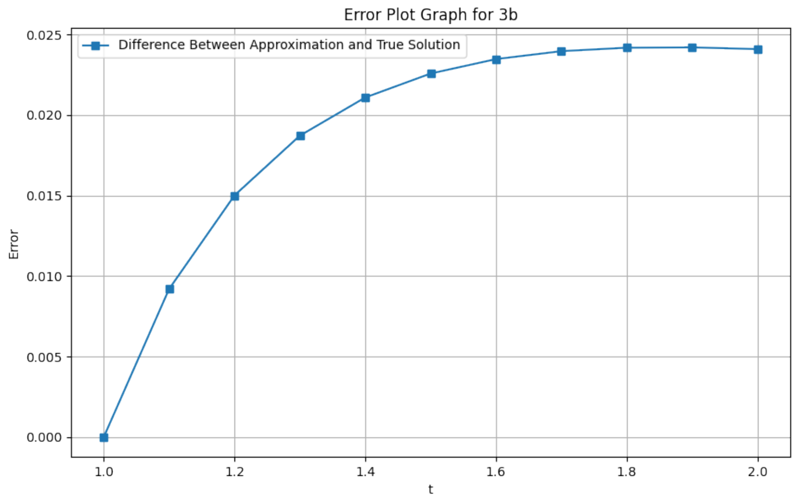


b. We directly differentiate the function and see whether it is a solution. Let $y(t) = \dfrac{-1}{\ln(t+1)}$. It is clear that $y(1) = -\dfrac{1}{\ln 2}$, and we have

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

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
0.000
0.005
0.010
0.015
0.020
0.025
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

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

By Python, we have

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

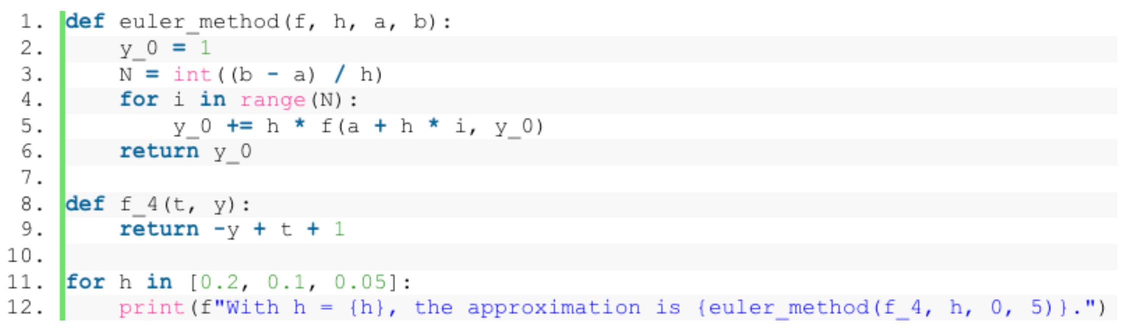
$$y(5) \approx 5.005153775207321 \quad \text{when } h = 0.1;$$

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

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

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

□



```
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

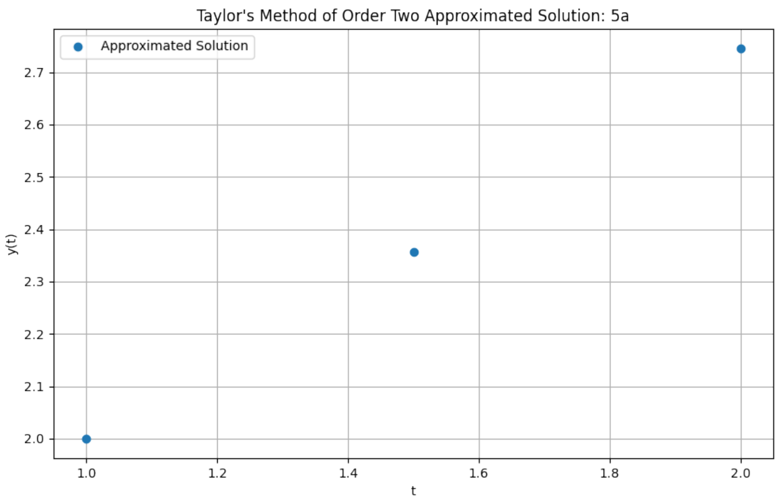
b. $y' = -y + t\sqrt{y}, \quad 2 \le t \le 3, \quad y(2) = 2$ with $h = 0.25$.

Solution.

a. By calculus,

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

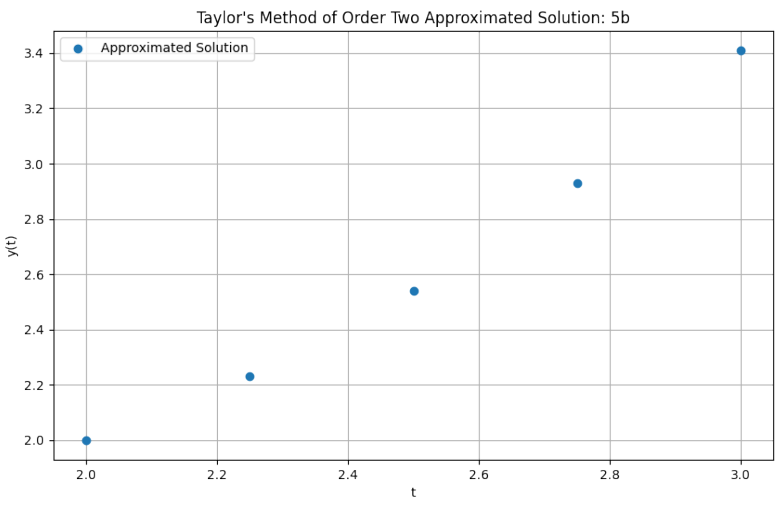
By Python, we have the following result:



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:



□

```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 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

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

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

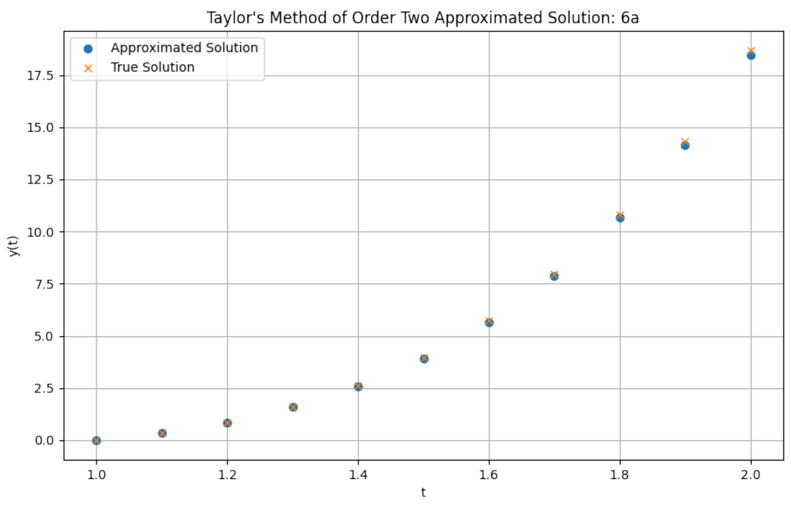
   iii. $y(1.97)$.

Solution.

a. By calculus,

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

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

$$\begin{aligned} y(1.04) &= 0.6 \cdot y(1.0) + 0.4 \cdot y(1.1) \\ &= 0.1359140914, \\ y(1.55) &= 0.5 \cdot y(1.5) + 0.5 \cdot y(1.6) \\ &= 4.7770327976, \\ y(1.97) &= 0.3 \cdot y(1.9) + 0.7 \cdot y(2.0) \\ &= 17.1748007649, \end{aligned}$$

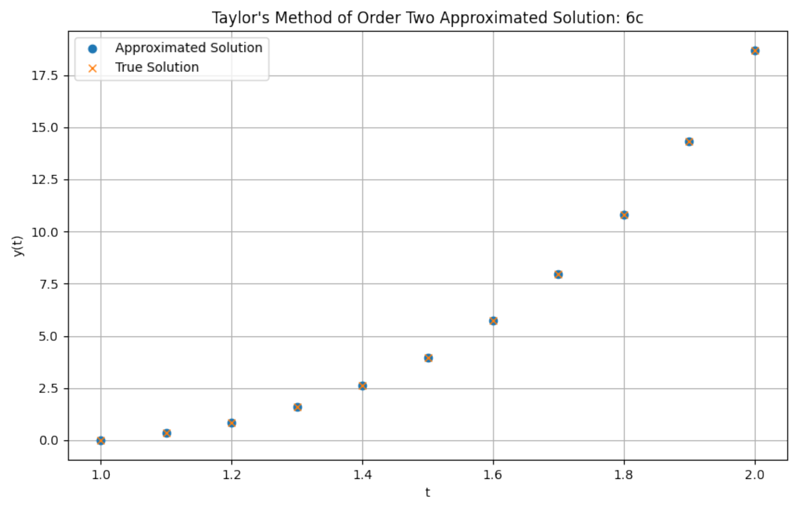
where the real values are

$$\begin{aligned} y(1.04) &= 0.1199874971, \\ y(1.55) &= 4.7886350208, \\ y(1.97) &= 17.2792984356. \end{aligned}$$

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 cubic Hermite polynomial, we have

$$H_3(1.04) = 0.1199703835,$$

$$H_3(1.55) = 4.7885271557,$$

$$H_3(1.97) = 17.2790404208,$$

where the real values are

$$y(1.04) = 0.1199874971,$$

$$y(1.55) = 4.7886350208,$$

$$y(1.97) = 17.2792984356.$$

$\square$

```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 Four 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]:
    print(f"{0.6 * soln_list[0] + 0.4 * 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}")

for t in [1.04, 1.55, 1.97]:
    print(f"{y(t):.10f}")

def cubic_hermite_interpolation(tuple1, tuple2, x):
        x1, fx1, dfx1 = tuple1
        x2, fx2, dfx2 = tuple2

        df = (fx2-fx1)/(x2-x1)
        ddf1 = (df - dfx1)/(x2-x1)
        ddf2 = (dfx2 - df)/(x2-x1)
        dddf = (ddf2 - ddf1)/(x2-x1)

        return fx1 + (x-x1)*dfx1 + (x-x1)**2 * ddf1 + (x-x1)**2 * (x-x2)*dddf

print(f"1.04: {cubic_hermite_interpolation((1, 0, f(1, 0)), (1.1, approx_soln_c[1], f(1.1, approx_soln_c[1])), 1.04): .10f}")
print(f"1.55: {cubic_hermite_interpolation((1.5, approx_soln_c[5], f(1.5, approx_soln_c[5])), \
    (1.6, approx_soln_c[6], f(1.6, approx_soln_c[6])), 1.55): .10f}")
print(f"1.97: {cubic_hermite_interpolation((1.9, approx_soln_c[9], f(1.9, approx_soln_c[9])), \
    (2, approx_soln_c[10], f(2, approx_soln_c[10])), 1.97): .10f}")
```

The code for Problem 6

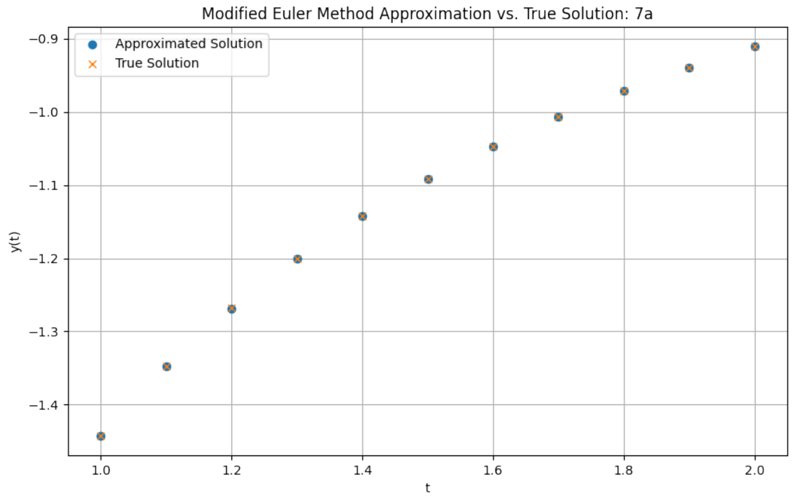
Problem 7. Use the modified Euler method to approximate the solutions to each of the following initial-value problems, and compare the results to the actual values.

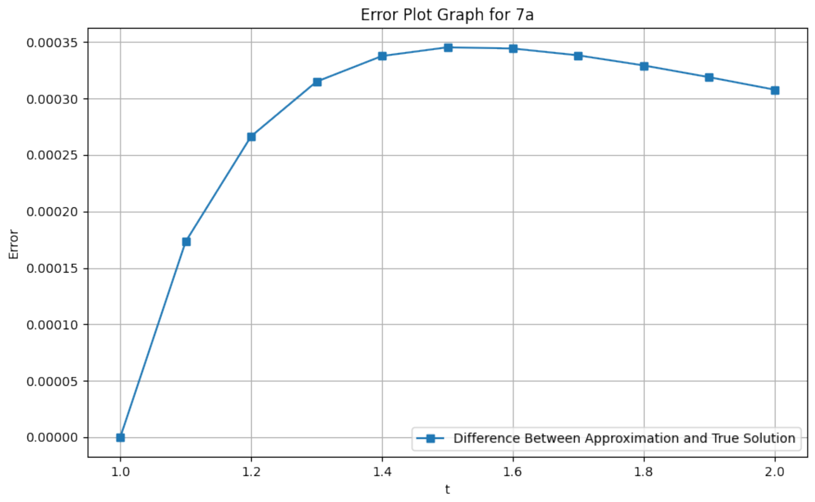
a. $y' = \dfrac{y^2}{1+t}, \quad 1 \le t \le 2, \quad y(1) = -\dfrac{1}{\ln 2}$ with $h = 0.1$; actual solution $y = \dfrac{-1}{\ln(t+1)}$; and

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

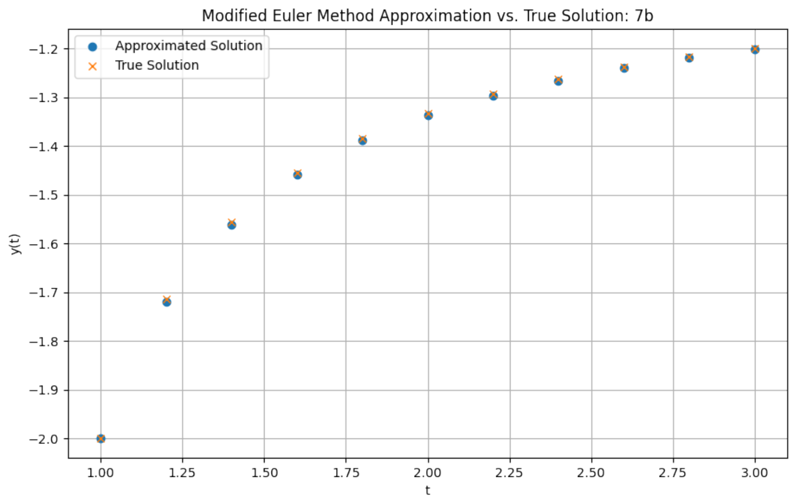
Solution. By Python, we have the following results:

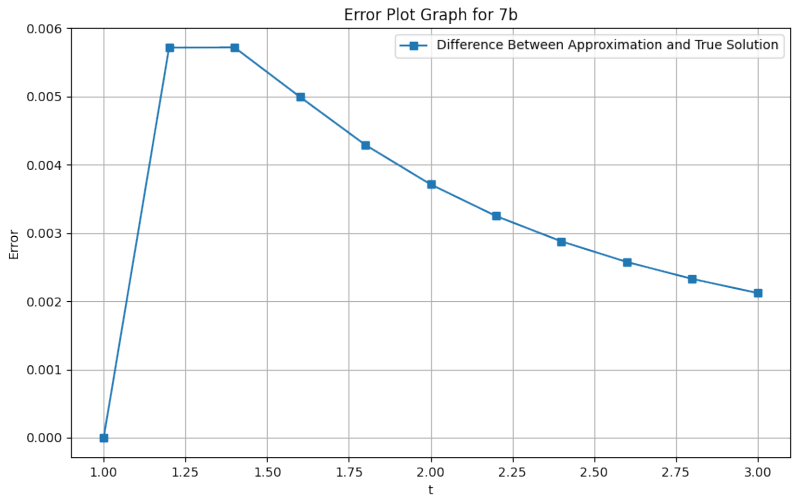
a. The first graph is with approximated solution and the actual solution; the second graph is the error plot graph. In the error plot graph, I used lines to connect each pair of adjacent dots to increase visual clarity.





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. Let $i \in \{0, 1, 2, \ldots, N-1\}$. The local truncation error at the $(i+1)$st step is

$$\tau_{i+1}(h) = \frac{y_{i+1} - y_i}{h} - \frac{a_1\, f(t_i, y_i) + a_2\, f(t_i + \alpha_2, \omega_1 + \delta_2\, f(t_i, y_i))}{h}.$$

Since

$$y' = f(t, y)$$

and

$$y'' = f'(t, y),$$

we have

$$y_{i+1} = y_i + h\, f(t_i, y_i) + \frac{h^2}{2} f'(t_i, y_i) + \frac{h^3}{6} f''(t_i, y_i) + \mathcal{O}(h^4).$$

Hence,

$$\begin{aligned}
\tau_{i+1}(h) &= \frac{y_{i+1} - y_i}{h} - \frac{a_1\, f(t_i, y_i) + a_2\, f(t_i + \alpha_2, \omega_1 + \delta_2\, f(t_i, y_i))}{h} \\
&= f(t_i, y_i) + \frac{h}{2} f'(t_i, y_i) + \frac{h^2}{6} f''(t_i, y_i) + \mathcal{O}(h^3) - \frac{a_1\, f(t_i, y_i) + a_2\, f(t_i + \alpha_2, \omega_1 + \delta_2\, f(t_i, y_i))}{h}
\end{aligned}$$

Since $\frac{h^2}{6} f''(t_i, y_i)$ will never be vanished, the truncation error cannot be with order $\mathcal{O}(h^3)$ (at most $\mathcal{O}(h^2)$). □

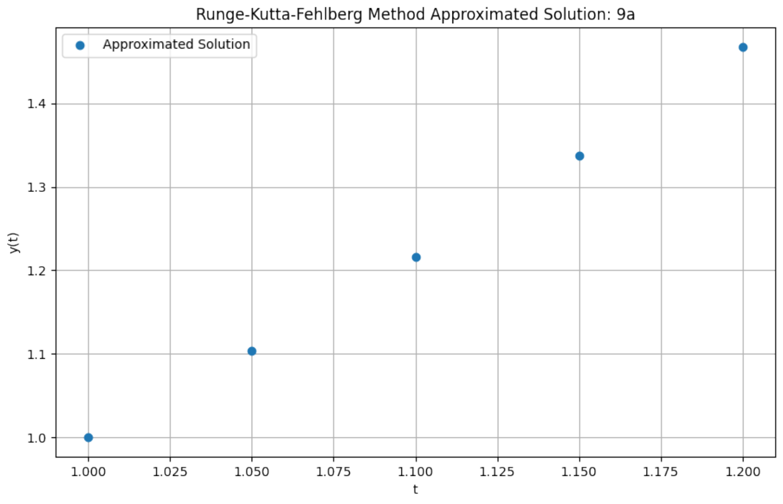
Problem 9. Use the Runge-Kutta-Fehlberg algorithm with tolerance $TOL = 10^{-4}$ to approximate the solution to the following initial-value problems.

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

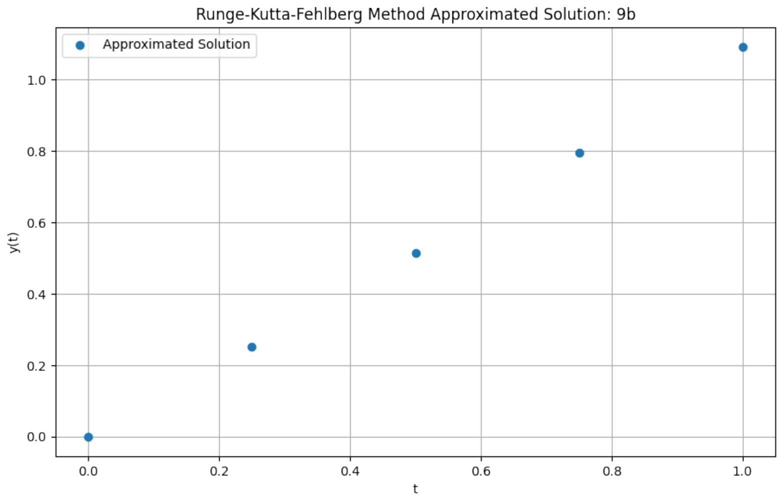
b. $y' = \sin t + e^{-t}, \quad 0 \le t \le 1, \quad y(0) = 0$ with $hmax = 0.25$ and $hmin = 0.02$.

Solution. The code is provided after the two graphs.

a.



b.



$\square$

```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

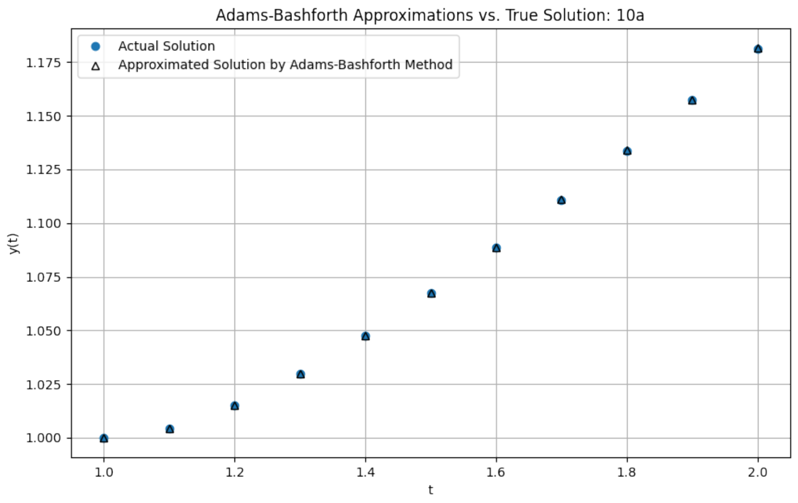
Problem 10. Use each of the Adams-Bashforth methods to approximate the solutions to the following initial-value problems. In each case use starting values obtained from the Runge-Kutta method of order four. Compare the results to the actual values.

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

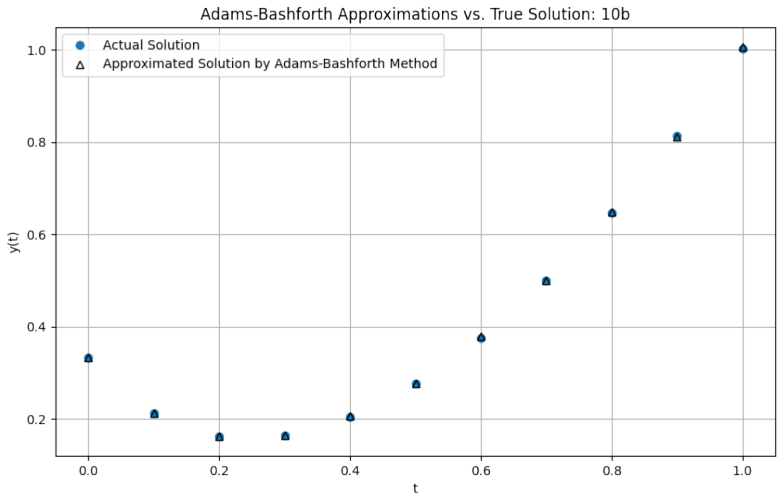
b. $y' = -5y + 5t^2 + 2t, \quad 0 \le t \le 1, \quad y(0) = \frac{1}{3}$ with $h = 0.1$; actual solution $y(t) = t^2 + \frac{e^{-5t}}{3}$.

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

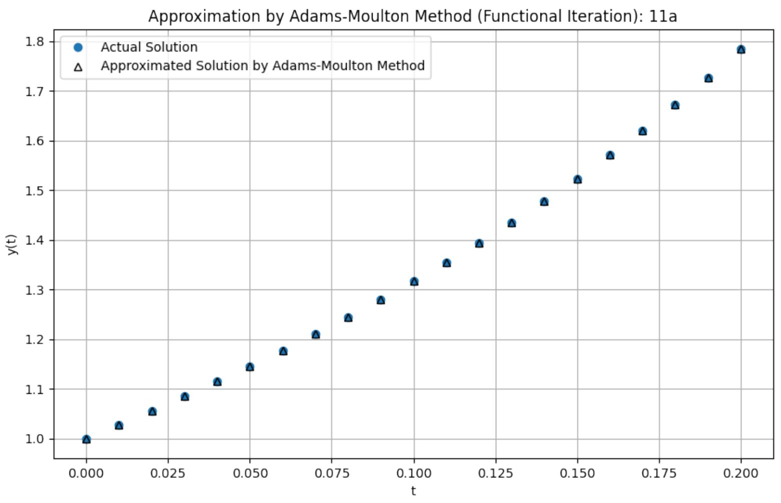
$$g(\omega) = \omega_i + \frac{h}{24}\left(9e^{\omega} + 19e^{\omega_i} - 5e^{\omega_{i-1}} + e^{\omega_{i-2}}\right).$$

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.

a. I use fixed-point iteration with accuracy 10^{-8} for each ω_{i+1}, $i = 2, 3, \ldots, 19$. The solution is approximated as follows:



b. Since fix-point iteration converges linearly and Newton's method converges quadratically, Newton's method will speed the convergence over funcitonal iteration. $\square$

```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_2s, 'o', label='Actual Solution')
    plt.plot(x_axis, y_1s, '^', color="black", markerfacecolor='none', label='Approximated Solution by Adams-Moulton 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 true_y(t):
    return 1 - math.log(1 - math.e * t)

omega_list = [true_y(0.01 * i) for i in range(3)]

def fix_point_iteration(f):
    omega = [omega_list[-j] for j in range(3, 0, -1)]
    new_omega = f(omega_list[-1], omega)
    old_omega = -100
    while abs(old_omega - new_omega) > 1e-8:
        old_omega = new_omega
        new_omega = f(new_omega, omega)
    return new_omega

def g(w, W):
    [w2, w1, w0] = W
    return w0 + 0.01 * (9 * math.exp(w) + 19 * math.exp(w0) - 5 * math.exp(w1) + math.exp(w2)) / 24

for _ in range(3, 21):
    omega_list.append(fix_point_iteration(g))

t_values = [0.01 * i for i in range(21)]
real_y_values = [true_y(0.01 * i) for i in range(21)]
plot_arrays(t_values, omega_list, real_y_values,  "Approximation by Adams-Moulton Method (Functional Iteration): 11a")
```

The code for Problem 11 a

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

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

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. Since $y'(t) = f(t, y(t))$,

$$y(t_{i+1}) = y(t_i) + ahy'(t_i) + bhy'(t_{i-1}) + chy'(t_{i-2}),$$

where

$$\begin{aligned}
y(t_{i+1}) &= y(t_i + h) \\
&= y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(t_i) + \frac{h^3}{6}y'''(t_i) + \cdots, \\
y'(t_{i-1}) &= y'(t_i - h) \\
&= y'(t_i) - hy''(t_i) + \frac{h^2}{2}y'''(t_i) - \cdots, \\
y'(t_{i-2}) &= y'(t_i - 2h) \\
&= y'(t_i) - 2hy''(t_i) + \frac{4h^2}{2}y'''(t_i) - \cdots.
\end{aligned}$$

Hence, we have

$$\begin{aligned}
hy'(t_i) &= ahy'(t_i) + bhy'(t_i) + chy'(t_i) \\
\frac{h^2}{2}y''(t_i) &= -ah^2y''(t_i) - 2chy''(t_i) \\
\frac{h^3}{6}y'''(t_i) &= \frac{bh^3}{2}y'''(t_i) + \frac{4ch^3}{2}y'''(t_i),
\end{aligned}$$

which yields $(a, b, c) = \left(\frac{23}{12}, -\frac{4}{3}, \frac{5}{12}\right)$. $\square$

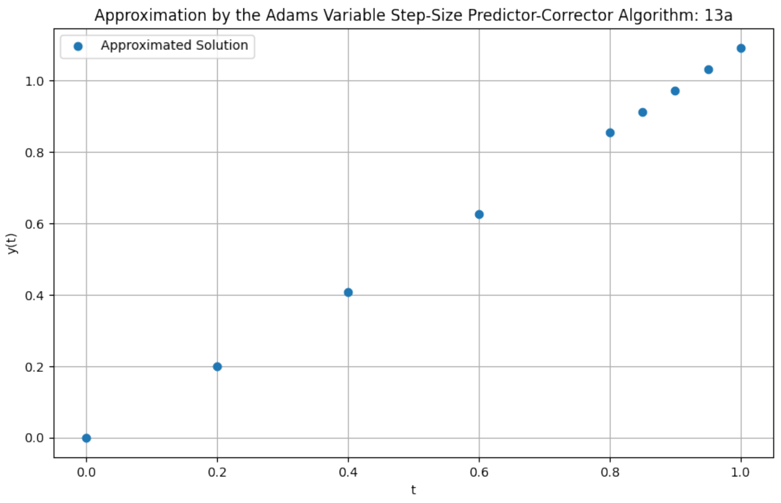
Problem 13. Use the Adams variable step-size predictor-corrector algorithm with $TOL = 10^{-4}$ to approximate the solutions to the following initial-value problems:

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

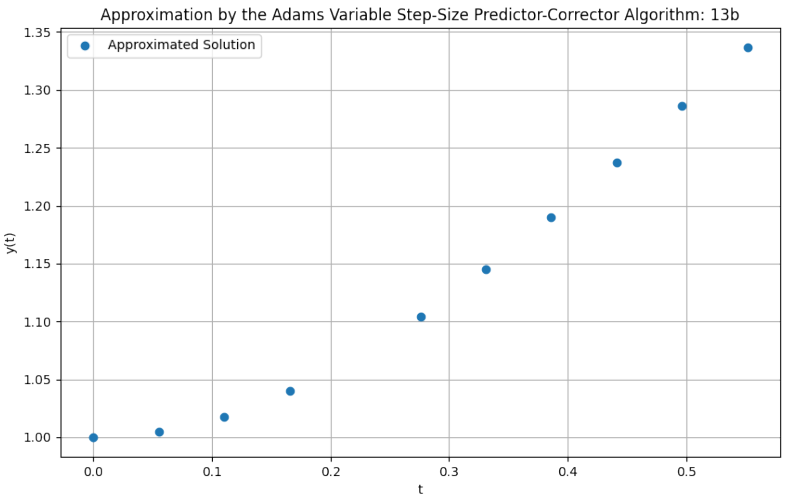
b. $y' = -ty + \dfrac{4t}{y}, \quad 0 \leq t \leq 1, \quad y(0) = 1$ with $hmax = 0.2$ and $hmin = 0.01$.

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[-3:]}.png", transparent=True)

def adams_variable_step_size_predictor_corrector(f, t, y, b, h_min, h_max, tol, N):
    def rk4(f, t, y, h):
        k1 = h*f(t, y)
        k2 = h*f(t + h/2, y + k1/2)
        k3 = h*f(t + h/2, y + k2/2)
        k4 = h*f(t + h, y + k3)
        t=t+h
        return t,y + (k1 + 2*k2 + 2*k3 + k4)/6

    AB_correctors = [
        [2, 3, -1],
        [12, 23, -16, 5],
        [24, 55, -59, 37, -9],
        [720, 1901, -2774, 2616, -1274, 251]
    ]
    AM_correctors=[
        [2, 1, 1],
        [12, 5, 8, -1],
        [24, 9, 19, -5,1],
        [720, 251, 646, -264, 106,-19]
    ]

    corrector = AB_correctors[N-2]
    time = [t]
    y_values = [y]
    result=[]
    last=False
    h=h_max

    for _i in range(1, N):
        t_i,y_i=rk4(f, time[_i-1], y_values[_i-1],h)
        y_values.append(y_i)
        time.append(t_i)
    nflag=True
    i=N
    t=time[len(time)-1]+h

    while h != 0:
        fix = 0

        for j in range(N):
            fix += corrector[j+1] * f(time[i-j-1], y_values[i-j-1])
        wp = y_values[i-1] + h/corrector[0]*fix


        coeffs = AM_correctors[N-2]
        fix = coeffs[1]*f(t, wp)

        for j in range(len(coeffs)-2):
            fix += coeffs[j+2] * f(time[i-j-1], y_values[i-j-1])
        wc = y_values[i-1] + h/coeffs[0]*fix

        sigma=19*abs(wc-wp)/(270*h)

        if(sigma<=tol):
            y_values.append(wc)
            time.append(t)
            if nflag:
                for j in range(N):
                    result.append([time[i-N+j],y_values[i-N+j]])
            else:
                result.append([time[i],y_values[i]])
            if last:
                # print('last')
                result.append([time[i],y_values[i]])
                break
```

The code for Problem 13

```python
            else:
                i=i+1

                nflag=False
                if sigma<=(0.1*tol) or time[i-1]+h>b:
                    q=(tol/(2*sigma)) ** (0.25)
                    if q>4:
                        h=4*h
                    else:
                        h=q*h

                    if h>h_max:
                        h=h_max

                    if time[i-1]+4*h>=b:
                        h=(b-time[i-1])/4
                        last=True

                    for _i in range(-1, N-2):
                        t_i,y_i=rk4(f, time[i+_i], y_values[i+_i],h)
                        y_values.append(y_i)
                        time.append(t_i)
                    nflag=True

                elif time[i-1]+4*h>=b:
                    h=(b-time[i-1])/4
                    last=True

                    for _i in range(-1, N-2):
                        t_i,y_i=rk4(f, time[i+_i], y_values[i+_i],h)
                        y_values.append(y_i)
                        time.append(t_i)
                    nflag=True
                    i=i+N-1
        else:
            q=(tol/(2*sigma))**0.25

            if(q<0.1):
                h=0.1*h
            else:
                h=q*h

            if h<h_min:
                break
            else :
                if nflag:
                    i=i-(N-1)

                for _i in range(-1, N-2):
                        t_i,y_i=rk4(f, time[i+_i], y_values[i+_i],h)
                        if((i+_i+1)>=len(y_values)):
                            y_values.append(y_i)
                            time.append(t_i)

                        else:
                            y_values[i+_i+1]=y_i
                            time[i+_i+1]=t_i
                nflag=True
                i=i+N-1
        t=time[len(time)-1]+h
    t_values, approx_soln_list = [result[i][0] for i in range(len(result))], [result[i][1] for i in range(len(result))]
    return t_values, approx_soln_list


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

def f_b(t, y):
    return - t * y + 4 * t / y

t_values_a, approximation_a = adams_variable_step_size_predictor_corrector(f_a, 0, 0, 1, 0.01, 0.2, 1e-4, 4)

plot_array(t_values_a, approximation_a, "Approximation by the Adams Variable Step-Size Predictor-Corrector Algorithm: 13a")

t_values_b, approximation_b = adams_variable_step_size_predictor_corrector(f_b, 0, 1, 1, 0.01, 0.2, 1e-4, 4)

plot_array(t_values_b, approximation_b, "Approximation by the Adams Variable Step-Size Predictor-Corrector Algorithm: 13b")
```

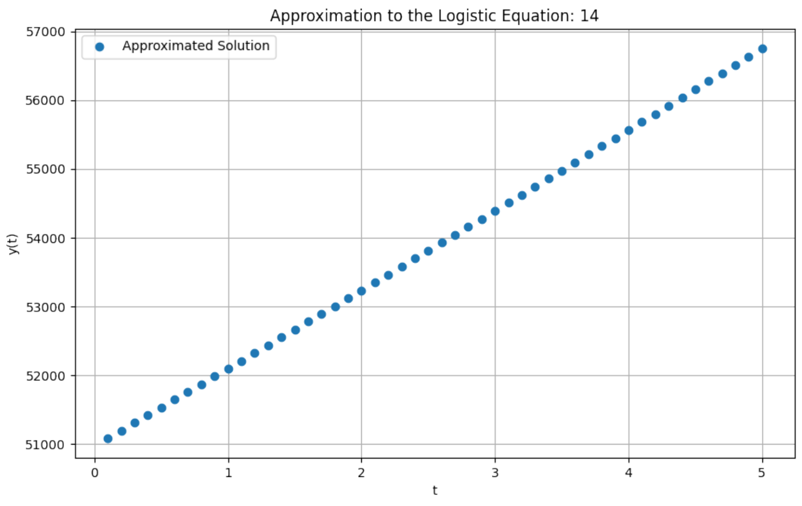
The code for Problem 13 (Contidued)

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

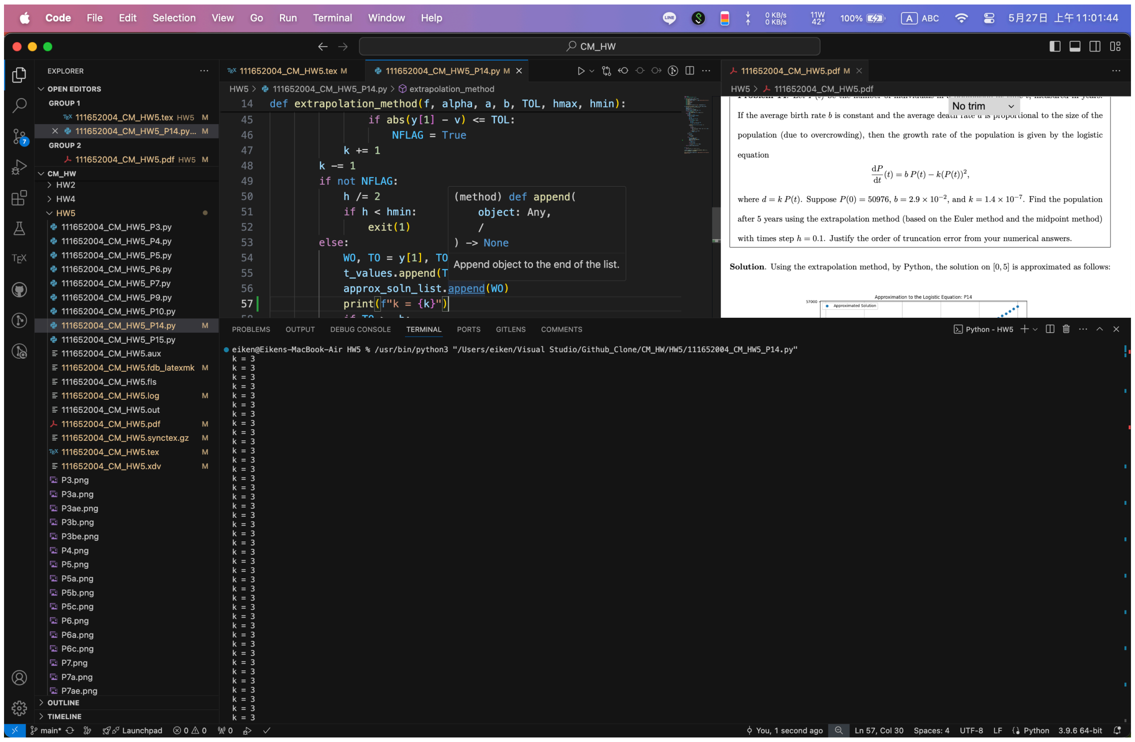
$$\frac{\mathrm{d}P}{\mathrm{d}t}(t) = b\,P(t) - k(P(t))^2,$$

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. Using the extrapolation method, by Python, the solution on $[0, 5]$ is approximated as follows:



Hence $y(5) = 56751.0367687$. Since $k = 3$ in the code, the order of truncation error is $\mathcal{O}(h^6)$. $\square$



```python
def extrapolation_method(f, alpha, a, b, TOL, hmax, hmin):
                if abs(y[1] - v) <= TOL:
                    NFLAG = True
            k += 1
        k -= 1
        if not NFLAG:
            h /= 2
            if h < hmin:
                exit(1)
        else:
            W0, T0 = y[1], T0
            t_values.append(T
            approx_soln_list.append(W0)
            print(f"k = {k}")
```

```
(method) def append(
    object: Any,
    /
) -> None
Append object to the end of the list.
```

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. Using the extrapolation method, by Python, the solution on $[0, 5]$ is approximated as follows:

```
eiken@Eikens-MacBook-Air HW5 % /usr/bin/python3 "/Users/eiken/Visual Studio/Github_Clone/CM_HW/HW5/111652004_CM_HW5_P14.py"
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
k = 3
```

k for each iteration is 3

```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 extrapolation_method(f, alpha, a, b, TOL, hmax, hmin):
    NK = [0, 2, 4, 6, 8, 12, 16, 24, 32]
    t_values = []
    approx_soln_list = []
    Q = [[0] for _ in range(8)]
    TO, WO, h, FLAG = a, alpha, hmax, True
    for i in range(1, 8):
        for j in range(1, i + 1):
            Q[i].append((NK[i + 1] / NK[j]) ** 2)
    while (FLAG):
        y = [0]
        k, NFLAG = 1, False
        while(k <= 8 and (not NFLAG)):
            HK = h / NK[k]
            T = TO
            W2 = WO
            W3 = W2 + HK * f(T, W2)
            T = TO + HK
            for j in range(1, NK[k]):
                W1 = W2
                W2 = W3
                W3 = W1 + 2 * HK * f(T, W2)
                T = TO + (j + 1) * HK

            y.append((W3 + W2 + HK * f(T, W3)) / 2)
            if k >= 2:
                j = k
                v = y[1]
                while(j >= 2):
                    y[j - 1] = y[j] + (y[j] - y[j - 1]) / (Q[k - 1][j - 1] - 1)
                    j -= 1
                if abs(y[1] - v) <= TOL:
                    NFLAG = True
            k += 1
        k -= 1
        if not NFLAG:
            h /= 2
            if h < hmin:
                exit(1)
        else:
            WO, TO = y[1], TO + h
            t_values.append(TO)
            approx_soln_list.append(WO)
            if TO >= b:
                FLAG = False
            elif TO + h > b:
                h = b - TO
            elif k <= 3 and h < 0.5 * hmax:
                h *= 2
    return t_values, approx_soln_list


def f(t, y):
    return 2.9e-2 * y - 1.4e-7 * y ** 2

t_values, approx_soln = extrapolation_method(f, 50976, 0, 5, 1e-9, 0.1000000001, 0.999999999)

plot_array(t_values, approx_soln, "Approximation to the Logistic Equation: P14")
```

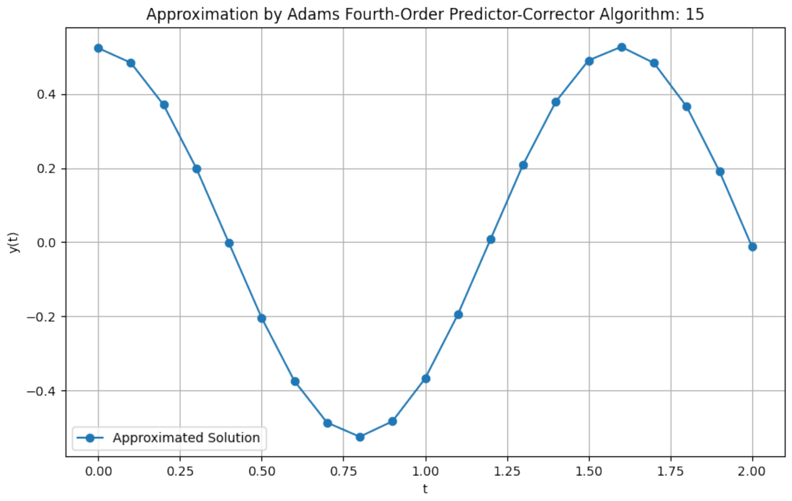
The code for Problem 14

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 $t = 0, 1, 2$:

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

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

Solution. The solutions is approximated as follows:



For values approximated at $t = 0, 1, 2$:

t	Approximation
0	0.523599
1	-0.368034
2	-0.012316

$\square$

```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 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 adams_fourth_order_predictor_correction_method(f, alpha, a, b, N):
    h = (b - a) / N
    t_0 = a
    y_0 = alpha
    approx_soln_list = [y_0]
    t_values = [t_0]
    m = len(alpha)

    for i in range(1, N + 1):
        if i < 4:
            k = [[], [], [], []]
            for j in range(m):
                k[0].append(h * f[j](t_values[-1], approx_soln_list[-1]))
            for j in range(m):
                k[1].append(h * f[j](t_values[-1] + h / 2, (approx_soln_list[-1][0] + k[0][0] / 2, approx_soln_list[-1][1] + k[0][1] / 2)))
            for j in range(m):
                k[2].append(h * f[j](t_values[-1] + h / 2, (approx_soln_list[-1][0] + k[1][0] / 2, approx_soln_list[-1][1] + k[1][1] / 2)))
            for j in range(m):
                k[3].append(h * f[j](t_values[-1] + h, (approx_soln_list[-1][0] + k[2][0], approx_soln_list[-1][1] + k[2][1])))
            w = []
            for j in range(m):
                w.append(approx_soln_list[-1][j] + (k[0][j] + 2 * k[1][j] + 2 * k[2][j] + k[3][j]) / 6)
            approx_soln_list.append(w)
            t = a + i * h
            t_values.append(t)
        else:
            w = approx_soln_list[-4:]
            t = t_values[-4:]
            t_values.append(a + i * h)
            w_0_list = []
            W = []
            for j in range(m):
                w_0 = w[3][j] + h * (55 * f[j](t[3], w[3]) - 59 * f[j](t[2], w[2]) + 37 * f[j](t[1], w[1]) - 9 * f[j](t[0], w[0])) / 24
                w_0_list.append(w_0)
            for j in range(m):
                w_0 = w[3][j] + h * (9 * f[j](t_values[-1], w_0_list) + 19 * f[j](t[3], w[3]) - 5 * f[j](t[2], w[2]) + f[j](t[1], w[1])) / 24
                W.append(w_0)
            approx_soln_list.append(W)

    return t_values, approx_soln_list

def phi(t, y):
    return -32.14 / 2 * math.sin(y[0])

def psi(t, y):
    return y[1]

f = [psi, phi]

t_list, y_approx_list = adams_fourth_order_predictor_correction_method(f, (math.pi / 6, 0), 0, 2, 20)

plot_array(t_list, [y[0] for y in y_approx_list], "Approximation by Adams Fourth-Order Predictor-Corrector Algorithm: 15")
```

The code for Problem 15

Problem 16. Consider the differential equation

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

a. Show that

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

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

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.

a. We have

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + \frac{h^2}{2} y''(t_i) + \frac{h^3}{6} y'''(t_i) \cdots$$

and

$$y(t_{i+2}) = y(t_i) + 2hy'(t_i) + \frac{4h^2}{2} y''(t_i) + \frac{8h^3}{6} y'''(t_i) + \cdots .$$

Hence,

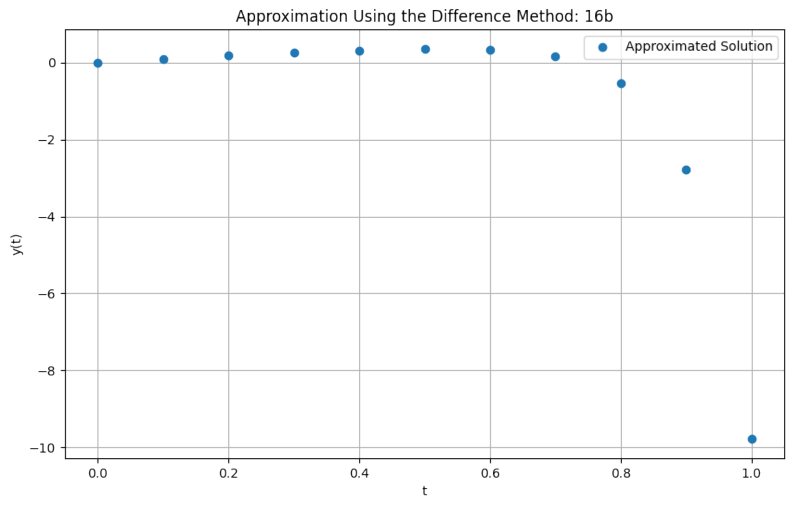
$$4y(t_{i+1}) + y(t_{i+2}) = 4y(t_i) + 2hy'(t_i) + \frac{2h^2}{3} y'''(t_i) + \cdots ,$$

which implies

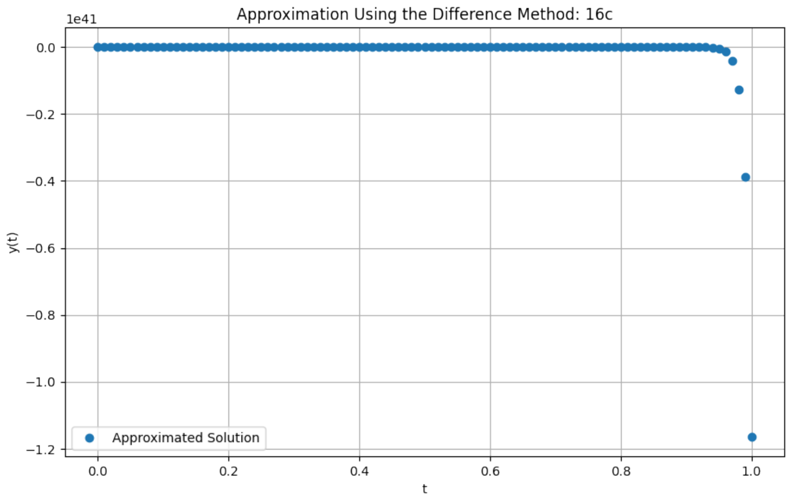
$$y'(t_i) = \frac{-3y(t_y) + 4y(t_{i+1}) - y(t_{i+2})}{2h} + \frac{h^2}{3} y'''(\xi_i)$$

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

b. By Python, the solution is approximated as follows:



c. By Python, the solution is approximated as follows:



d. The characteristic polynomial is

$$P(\lambda) = \lambda^2 - 4\lambda + 3,$$

which has roots $\lambda = 1, 3$. Since one of them is with magnitude greater than one, the method does not satisfy the root condition. Thus, it is not stable and not convergent. From (a), we have

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

for some $\xi_i \in (t_i, t_{i+2})$. Hence, the truncation error is

$$\tau_i(h) = \frac{h^2}{3} y'''(\xi_1) = \frac{h^2}{3} f''(t, y(t)).$$

If the function $f''(t, y(t))$ is bounded on $[a, b]$, then

$$\lim_{h \to 0} \max_{1 \leq i \leq N} |\tau_i(h)| = 0,$$

which implies that the method is consistent; otherwise, the method is not consistent. $\square$

Problem 17. Given the multistep method

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

with starting values ω_0, ω_1, and ω_2:

a. Find the local truncation error.

b. Comment on consistency, stability, and convergence.

Solution.

a. I consider ω_i as $y(t_i)$. I use the notation $y(t_i)$ instead of ω_i as it is more clear with derivatives. We have

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(t_i) + \frac{h^3}{6}y'''(t_i) + \frac{h^4}{24}y^{(4)}(t_i) + \cdots,$$
$$y(t_{i-1}) = y(t_i) - hy'(t_i) + \frac{h^2}{2}y''(t_i) - \frac{h^3}{6}y'''(t_i) + \frac{h^4}{24}y^{(4)}(t_i) + \cdots,$$

and

$$y(t_{i-2}) = y(t_i) - 2hy'(t_i) + \frac{4h^2}{2}y''(t_i) - \frac{8h^3}{6}y'''(t_i) + \frac{16h^4}{24}y^{(4)}(t_i) + \cdots.$$

Thus,

$$\begin{aligned}
3f(t_w, \omega_i) &= \frac{1}{h} \cdot \left(\omega_{i+1} + \frac{3}{2}\omega_i - 3\omega_{i-1} + \frac{1}{2}\omega_{i-2}\right) \\
&= \frac{1}{h} \cdot \left(y(t_{i+1}) + \frac{3}{2}y(t_i) - 3y(t_{i-1}) + \frac{1}{2}y(t_{i-1})\right) \\
&= \frac{1}{h} \cdot \Bigg[\left(1 + \frac{3}{2} - 3 + \frac{1}{2}\right) \cdot y(t_i) \\
&\qquad + (1 + 3 - 1) \cdot hy'(t_i) \\
&\qquad + \left(\frac{1}{2} - \frac{3}{2} + 1\right) \cdot h^2 y''(t_i) \\
&\qquad + \left(\frac{1}{6} + \frac{1}{2} - \frac{4}{6}\right) \cdot h^3 y'''(t_i) \\
&\qquad + \left(\frac{1}{24} - \frac{1}{8} + \frac{1}{3}\right) \cdot h^4 y^{(4)}(t_i) + \cdots\Bigg] \\
&= 3y'(t_i) + \frac{h^3}{4}y^{(4)}(t_i) + \mathcal{O}(h^4).
\end{aligned}$$

Hence, there is some $\xi_i \in (t_{i-2}, t_i)$ such that

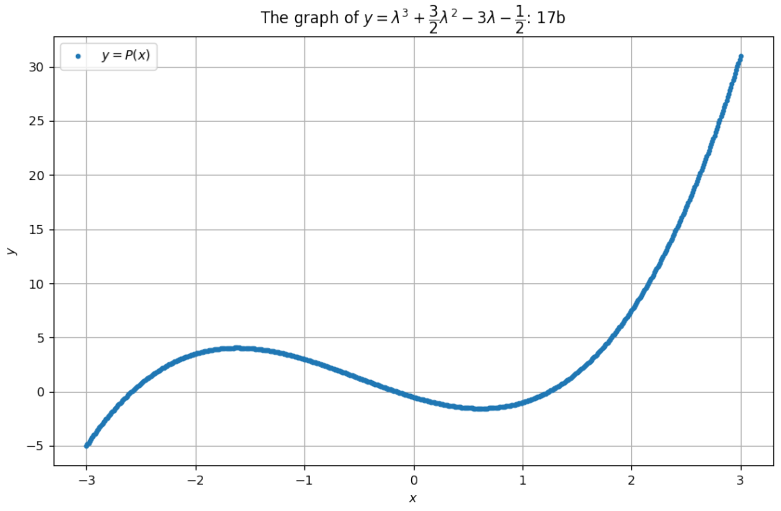
$$3f(t_w, \omega_i) = 3y'(t_i) + \frac{h^3}{4}y^{(4)}(\xi_i).$$

Therefore, the local truncation error is

$$\tau_i(h) = \frac{h^3}{12}y^4(\xi_i)$$

for some $\xi_i \in (t_{i-2}, t_i)$.

b. The characteristic polynomial is $P(\lambda) = \lambda^3 + \frac{3}{2}\lambda^2 - 3\lambda - \frac{1}{2}$, which has a root with magnitude greater than one (this root is in $(-3, -2)$). Hence, it is not stable and not convergent.



By (a), the local truncation error is

$$\tau_i(h) = \frac{h^3}{12}y^4(\xi_i).$$

If y^4 is bounded on the domain, then

$$\lim_{h\to 0} \max_{1\le i\le N} |\tau_i(h)| = \lim_{h\to 0} \frac{h^3}{12} \cdot \max_{1\le i\le N} |y^{(4)}(\xi_i)|$$
$$= 0.$$

Hence, the method is consistent. $\square$

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. Since the characteristic polynomial is $P(\lambda) = \lambda - 1$, which has a root $\lambda = 1$, the method is strongly stable, and thus is consistent. I consider ω_i as $y(t_i)$. I use the notation $y(t_i)$ instead of ω_i as it is more clear with derivatives. We have

$$y'(t_{i+1}) = y'(t_i) + hy''(t_i) + \frac{h^2}{2}y'''(t_i) + \cdots.$$

Hence,

$$\begin{aligned} y(t_{i+1}) &= y(t_i) + \frac{h}{2}\left(y'(t_i) + y'(t_i) + hy''(t_i) + \frac{h^2}{2}y'''(t_i) + \cdots\right) \\ &= y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(t_i) + \frac{h^3}{4}y'''(t_i) + \cdots. \end{aligned}$$

Hence, there is some $\xi \in (t_i, t_{i+1})$ such that

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(\xi_i).$$

Therefore, the local truncation error is

$$\tau_i(h) = \frac{h^2}{2}y''(\xi_i)$$

for some $\xi_i \in (t_i, t_{i+1})$. If y'' is bounded on the domain, then the method is consistent. $\square$

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

$$\begin{aligned}
k_1 &= h\, f(t_i, \omega_i), \\
k_2 &= h\, f(t_i + h/2, \omega_i + k_1/2), \\
k_3 &= h\, f(t_i + h/2, \omega_i + k_2/2), \\
k_4 &= h\, f(t_i + h, \omega_i + k_3), \\
\omega_{i+1} &= \omega_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)
\end{aligned}$$

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,

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

□