3/6/2024

Project 3 Report

MATH 452

Cameron Robinson

Table of Contents

Table of Figures

Project 3

Part 1

The objective of part one was to solve the two ODEs shown in Figure 1 using the trapezoid predictor-corrector method. The code for this method is shown in Figure 22 in the Appendix. Figure 1 also shows the solutions to each of the ODEs. Euler's method was used as the predictor and the trapezoid method was the corrector. The implementation of each method is illustrated in Figure 2. In general, Euler's method calculates an estimate for the ODE's solution at the next timestep, which is then input into the trapezoid method to get a more accurate estimate. This estimate is then iterated through the Trapezoid method multiple times to create an even more accurate estimate. Because of this ability to make better estimations through multiple iterations, predictor-corrector methods are supposed to be fairly accurate with relatively high step-sizes.

$$y_1' = (1-t)y, \quad y(0) = 3 \; on \; [0,3]$$

$$y_2' = ty^2, \; \; y(0) = \frac{2}{5} \; on \; [0,2]$$

$$y_1 = 3e^{t-\frac{1}{2}t^2}$$

$$y_2 = \frac{-1}{\frac{1}{2}t^2 - \frac{5}{2}}$$

Figure 1: ODEs and Solutions for Part 1 and 2

$$Euler - Trapezoid \; Predictor - Corrector$$

$$Predict \; (Euler's \; Method):$$

$$y(t+h) = y(t) + h * y'(t, y(t))$$

$$Correct \; (Trapezoid \; Method):$$

$$y(t+h) = y(t) + \frac{h}{2}\Big(y'\big(t+h, y(t+h)\big) + \; y'\big(t, y(t)\big)\Big)$$

Figure 2: Trapezoid P-C Method

Figure 3 contains two subplots. The first subplot shows solutions estimated with the trapezoid method for y1 at three different step sizes. For each estimate, the step size decreases by half. Since the method is accurate to $O(h^2)$, each estimate should be about 4x more accurate than the next highest step size. The change in accuracy is most evident in Figure 4, which shows the same solutions zoomed in. As shown,

the estimates with lower step sizes are much more accurate than the ones with higher step sizes, though not necessarily 4x more accurate. The trapezoid method seems to perform the worst when the solution is curved.

The second subplot shows graphs for the actual solution and a solution estimated with the built-in odeint function. Overall, odeint is more accurate than the solutions estimated with the trapezoid method, but the estimate with h = 0.0384 appears to be fairly close since neither are visible without zooming-in.

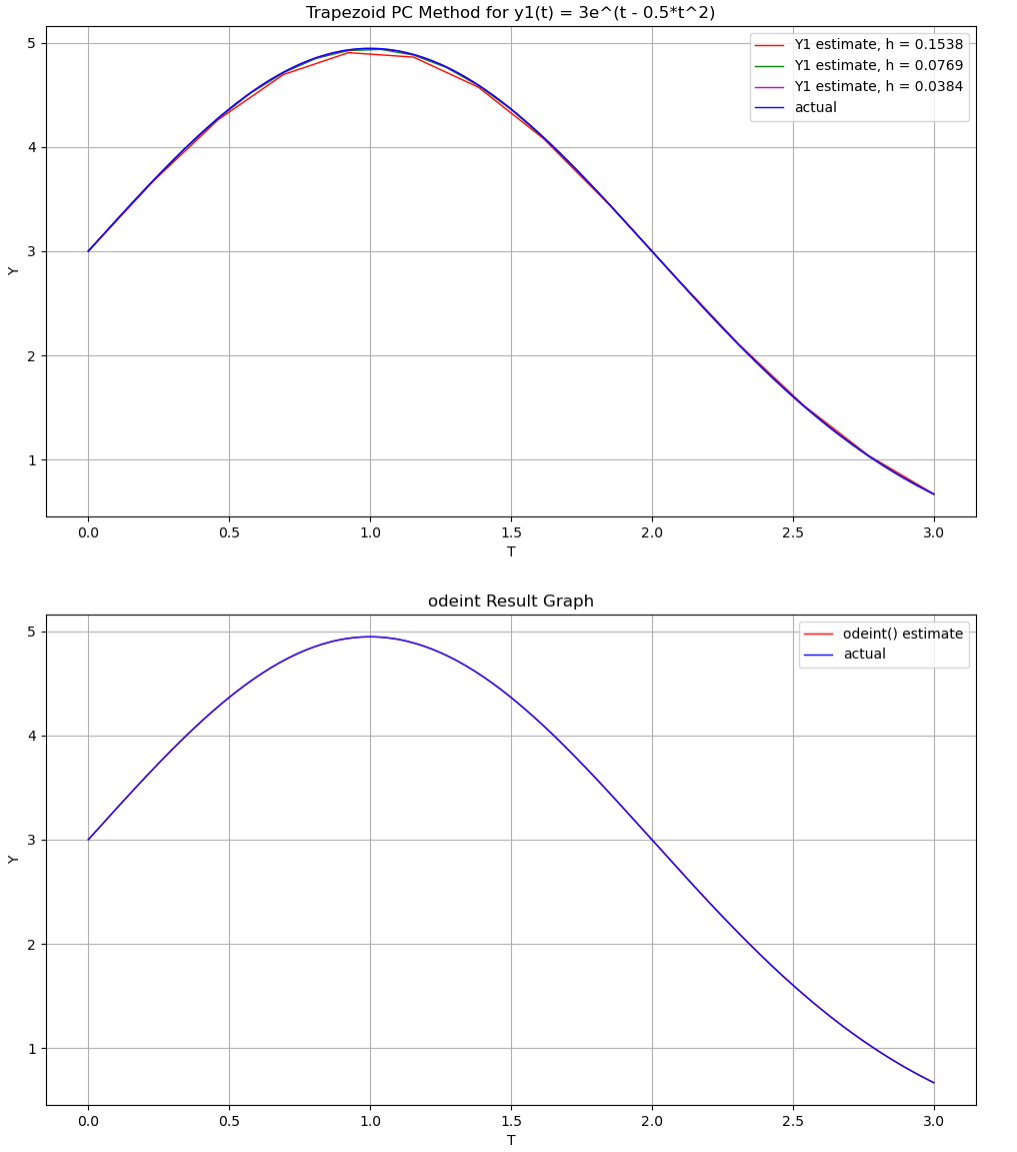


Figure 3: Part 1 Solution Graphs for y1

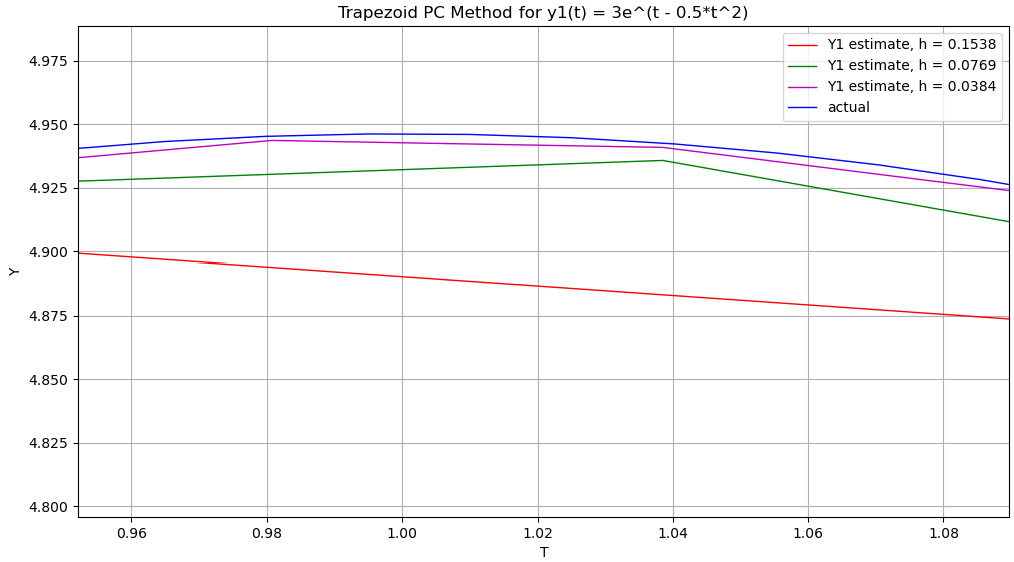


Figure 4: Part 1 Solutions to y1 - Zoomed in

Figure 5 shows the estimated and actual solutions to y2. In this case, the method appears to perform worse than for y1, especially as the slope increases at the right endpoint. This makes sense because the larger slope causes the estimates to overshoot, especially with larger step sizes. Unlike in the estimates for y1, the improvements in the estimates between the smaller step sizes are much more apparent. This is most apparent at the right endpoint where the solution with h = 0.0769 appears to be much more than 4x more accurate than the estimate with h = 0.1538. As before, the second subplot contains the solution estimated by odeint and the actual solution to y2. At the left endpoint, all three estimates from the trapezoid method appear to be about as accurate as odeint, however, at the right endpoint, odeint is much more accurate than even the solution with h = 0.0384. The code to create the graphs in Figure 3 and Figure 5 is shown in Figure 23 in the Appendix.

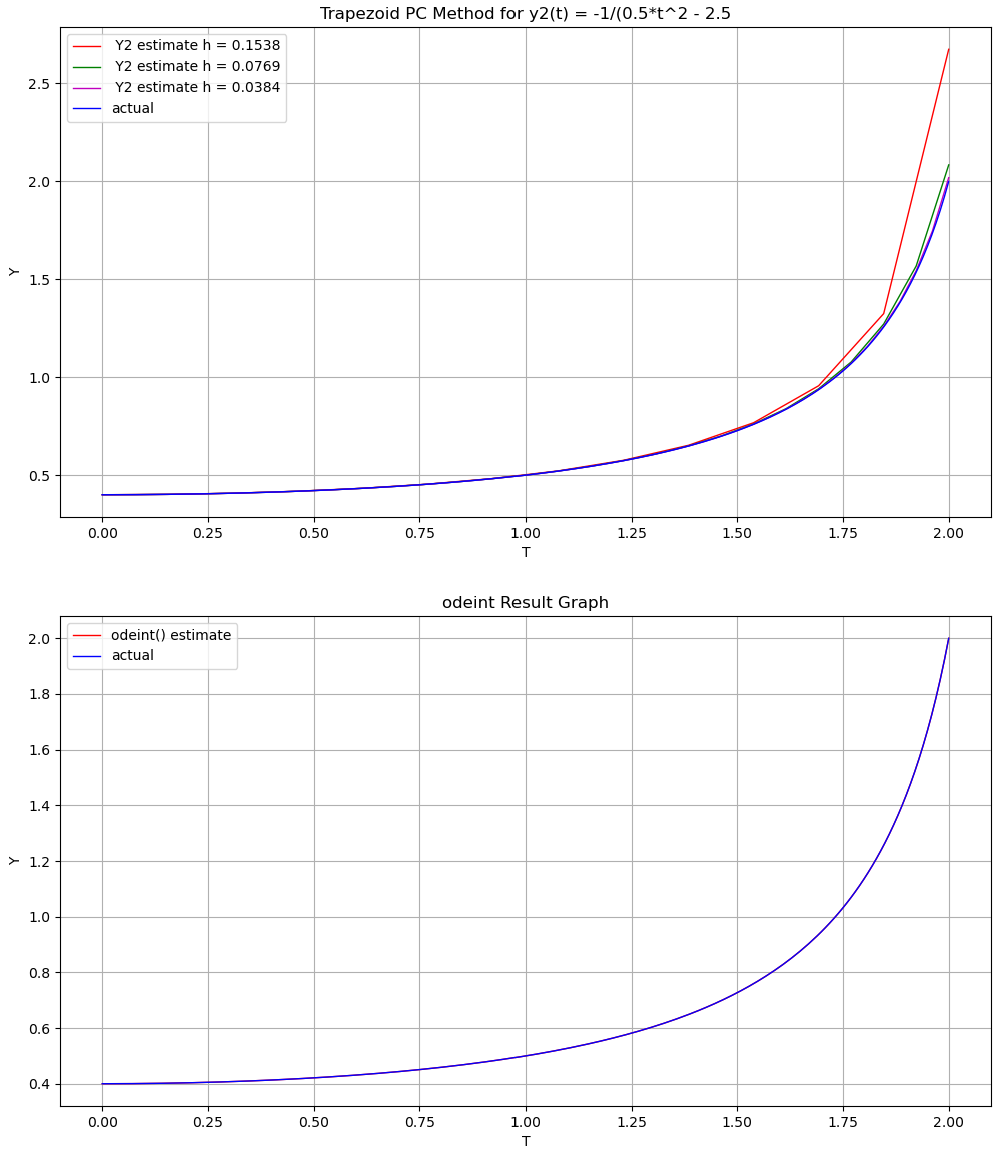


Figure 5: Part 1 Solution Graphs for y2

Part 2

The goal of part two was to solve the same ODEs in part one but using Adams-Bashforth of order four (AB4) as the predictor and Adams-Moulton of order four (AM4) as the corrector. Since AB4 requires four points to make a prediction for the next timestep, an order-four Runge-Kutta method (RK4) is used to make the first three estimations. Figure 6 shows the implementation of RK4, and Figure 7 shows the implementation of AB4 and AM4. The code for this method is shown in Figure 24.

$$RK4$$

$$k1 = f\big(t, y(t)\big)$$

$$k2 = f(t + \tfrac{1}{2}h, y(t) + \tfrac{1}{2}h\ k1)$$

$$k3 = f\left(t + \tfrac{1}{2}h, y(t) + \tfrac{1}{2}h\ k2\right)$$

$$k4 = f(t + h, y(t) + h\ k3)$$

$$y(t + h) = y(t) + \tfrac{h}{6}(k1 + 2\ k2 + 2\ k3 + k4)$$

Figure 6: RK4 Method Implementation

$$AB4 - AM4\ Predictor - Corrector$$

$$Predict\ (AB4):$$

$$y(t + h) = y(t) + \tfrac{h}{24}(55y'(t, y(t) - \ 59y'(t - h, y(t - h) + \ 37y'(t - 2h, y(t - 2h) - \ 9y'(t - 3h, y(t - 3h))$$

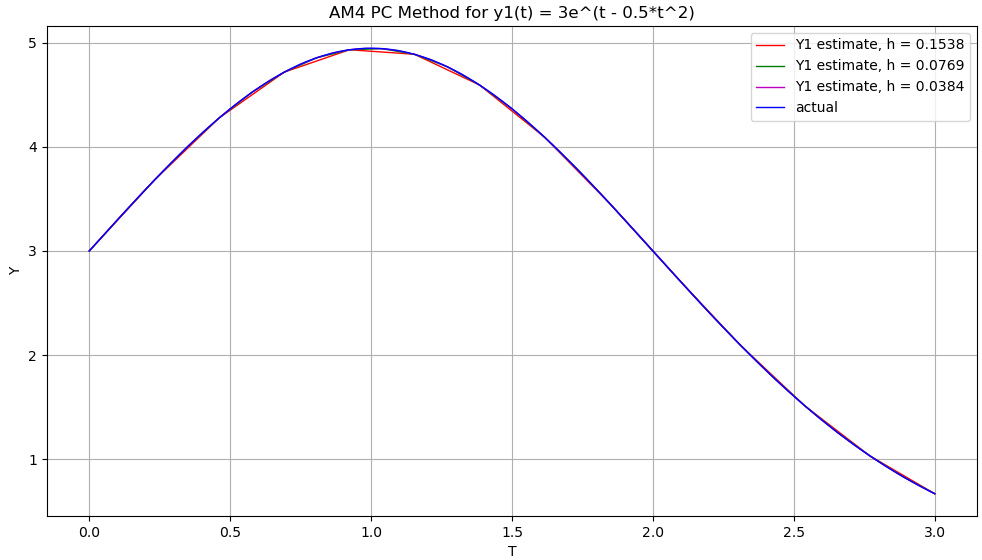
$$Correct\ (AM4):$$

$$y(t + h) = y(t) + \tfrac{h}{24}(9y'(t + h, y(t + h) + \ 19y'(t, y(t) - 5y'(t - h, y(t - h) + y'(t - 2h, y(t - 2h))$$

Figure 7: AB4-AM4 P-C Method Implementation

Figure 8 contains two subplots, one with the AB4-AM4 estimated solutions and the other with the odeint solution. Both subplots also have graphs of the exact solution. The estimated solutions of y1, at least the h = 0.1538 one, is slightly more accurate than the trapezoid predictor-corrector method. This is apparent at the top of the curve (about t = 1) where the AB4-AM4 method has a smaller gap between the actual and estimated solutions. This is expected since AB4 and AM4 both have $O(h^5)$ accuracy, while the trapezoid method only had $O(h^2)$ accuracy. Like the trapezoid method, this method does not perform very well for the curved sections of the solution, especially around t = 1.

Figure 9 shows the solutions to y1 zoomed in. As the step size decreases by a factor of two, the error should decrease by a factor of 32 (2^5), but his does not appear to be the case. The decrease in error seems more like a third of the previous estimate. Without zooming-in, the AB4-AM4 solution with h = 0.0384 appears to be about as accurate as odeint, but zooming-in indicates that odeint is still much more accurate since it takes a much smaller scale than the one shown in Figure 9 for the solution to be visible.



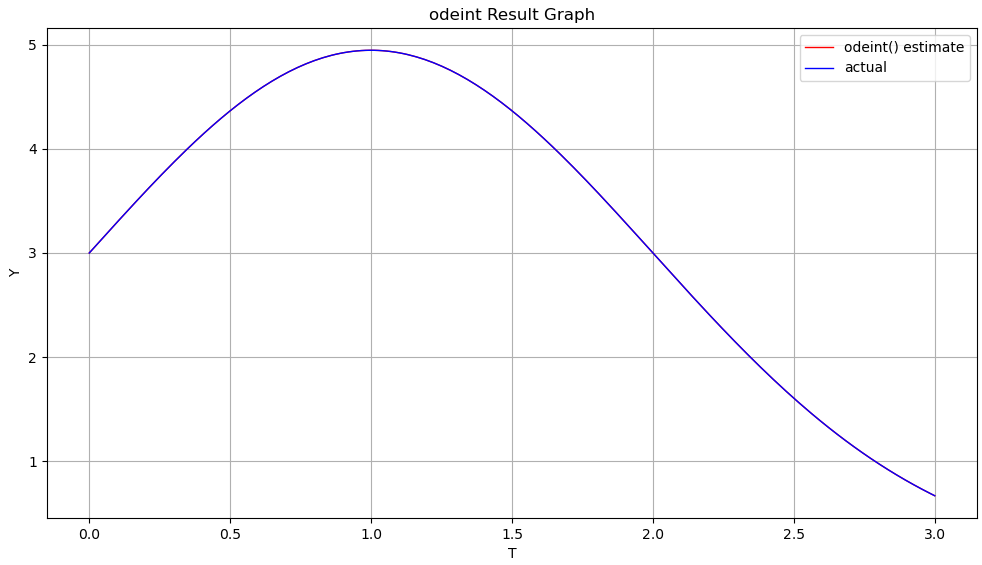


Figure 8: Part 2 Solution Graphs for y1

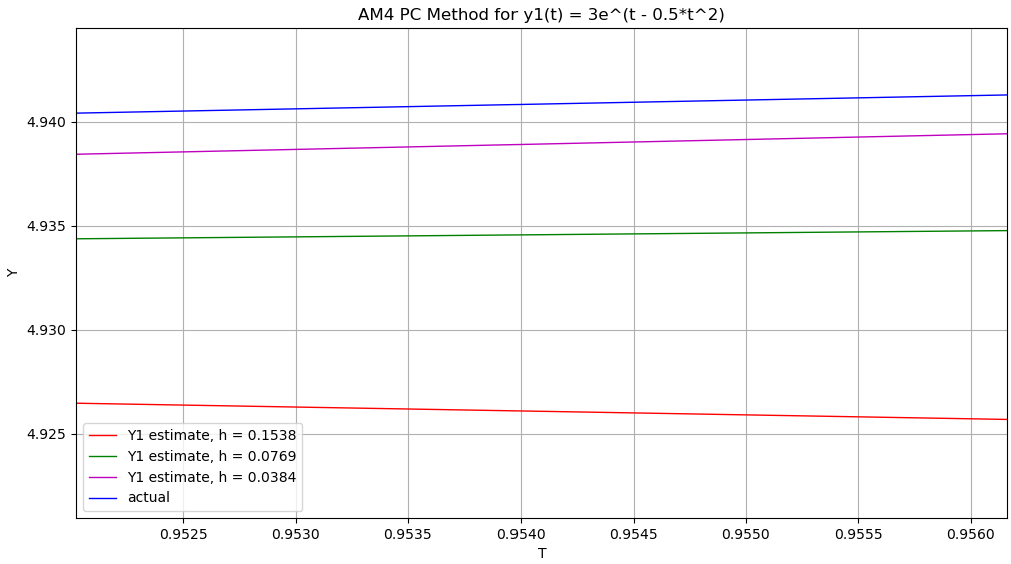


Figure 9: Part 2 Solution Graphs for y1 - Zoomed in

Figure 10 contains the AB4-AM4 predictor-corrector estimates for y2 at three different step-sizes, each decreasing by one-half. For this function, AB4-AM4 performed much better than the trapezoid method, especially at the right endpoint. In fact, the AB4-AM4 estimate with h = 0.1538 appears to be just slightly less accurate than the trapezoid method's estimate with h = 0.0769. The method does become more accurate with smaller step-sizes; however, it certainly does not appear to be 32x more accurate. As with the trapezoid method, this method seems to perform worse when the slope becomes steeper towards the right endpoint.

The code to create the graphs in Figure 8 and Figure 10 is shown in Figure 25 in the Appendix.

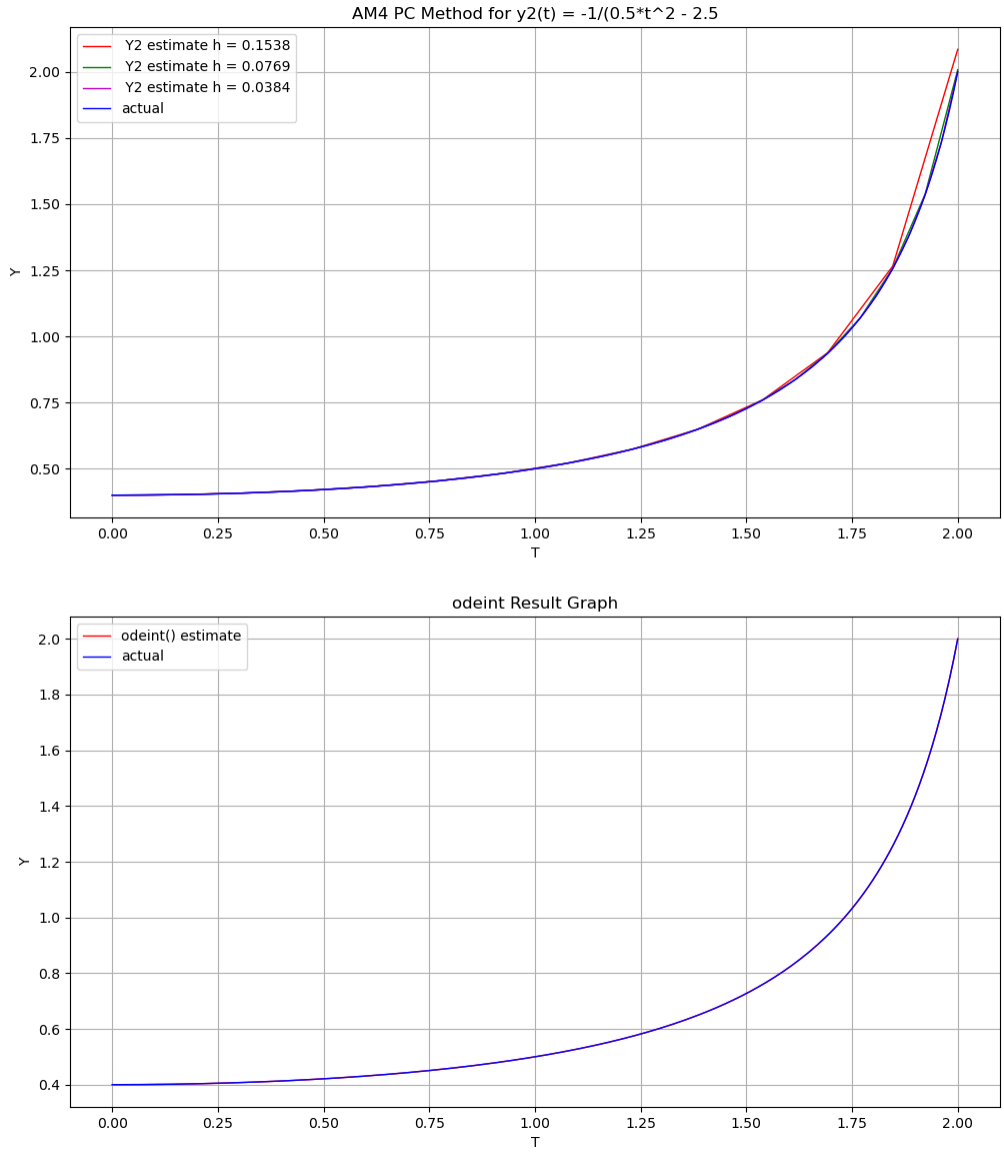


Figure 10: Part 2 Solution Graphs for y2

Part 3

The goal of part three was to implement any ODE-solving methods to solve the ODEs shown in Figure 11. Two methods were chosen for this part: RK4 and a forward/backward Euler's predictor-corrector method. The implementation of RK4 is the same as in part two, while the implementations of the forward and backward Euler's methods are shown in Figure 12. The Euler's predictor-corrector method was chosen just to see how well it would perform, while RK4 was chosen to have something to compare the Euler's predictor-corrector estimates to. This is not that fair of a comparison since forward and backward Euler both have $O(h^2)$ error while RK4 has $O(h^5)$, but at the same time, the predictor-corrector

method can be iterated at the same time step to make most of the estimates more accurate. The code for RK4 and the Euler's predictor-corrector method is shown in Figure 26 in the Appendix.

$$y_1' = \cos(t) - \sin(t) - y, \quad y(0) = 2 \; on \; [0,10]$$

$$y_2' = 2 + (y - 2t + 3)^{\frac{1}{2}}, \quad y(0) = 1 \; on \; [0,1.5]$$

$$y_1 = \cos(t) + e^{-t}$$

$$y_2 = 1 + 4t + \frac{1}{4}t^2$$

Figure 11: ODEs and Solutions for Part 3

$$Forward - Backward \; Euler \; Predictor - Corrector$$

$$Predict \; (Forward \; Euler's):$$

$$y(t+h) = y(t) + h * y'(t, y(t))$$

$$Correct \; (Backward \; Euler's):$$

$$y(t+h) = y(t) + h * y'(t+h, y(t+h))$$

Figure 12: Forward-Backward Euler's P-C Method

Figure 13 contains two subplots: the first with estimates of y1 by both RK4 and forward/backward Euler's method at three different step sizes; and the second with the odeint estimate for y1. Both subplots also contain the exact solutions to y1. From the graph it is immediately clear that RK4 performed much better than the Euler's predictor-corrector method, especially around the local minima and maximum. The Euler's estimate with h = 0.1923 is not even close to being as accurate as the RK4 estimate with h = 0.7692. The difference in accuracy is even clearer in Figure 14, which shows the same graph but zoomed in. With both methods, the estimates become much more accurate with smaller step sizes. As the step sizes decrease by half, the accuracy of the Euler's predictor-corrector method estimates become more than 2x more accurate, while the RK4 estimates seem to become even more accurate. This seems reasonable considering the Euler's methods' accuracy should increase by about 4x while RK4's accuracy should increase by about 32x when the step size decreases by half. Although RK4 is accurate, it is still not as accurate as odeint, however, it is not possible to see this difference without zooming in on the graphs much further.

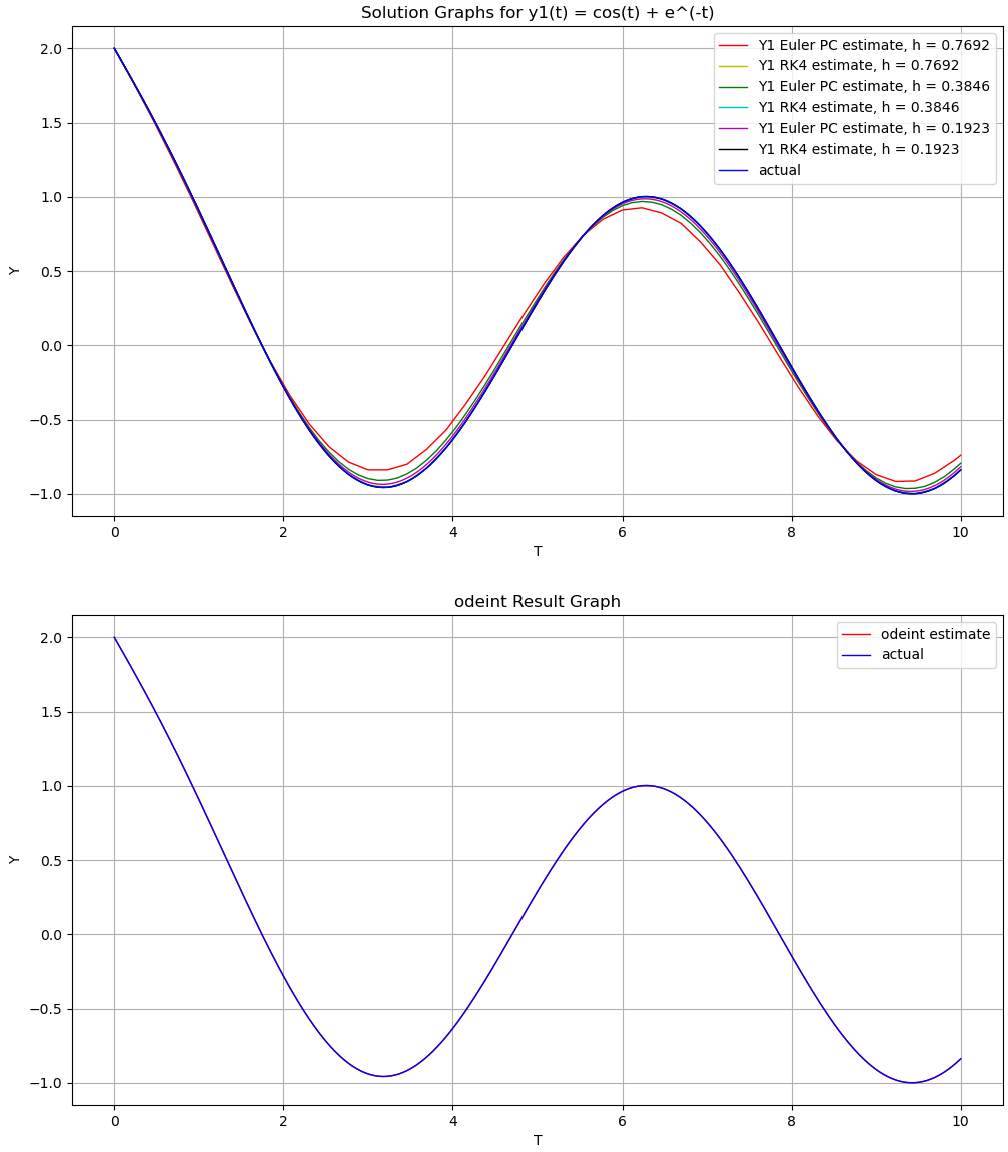


Figure 13: Part 3 Solution Graphs for y1

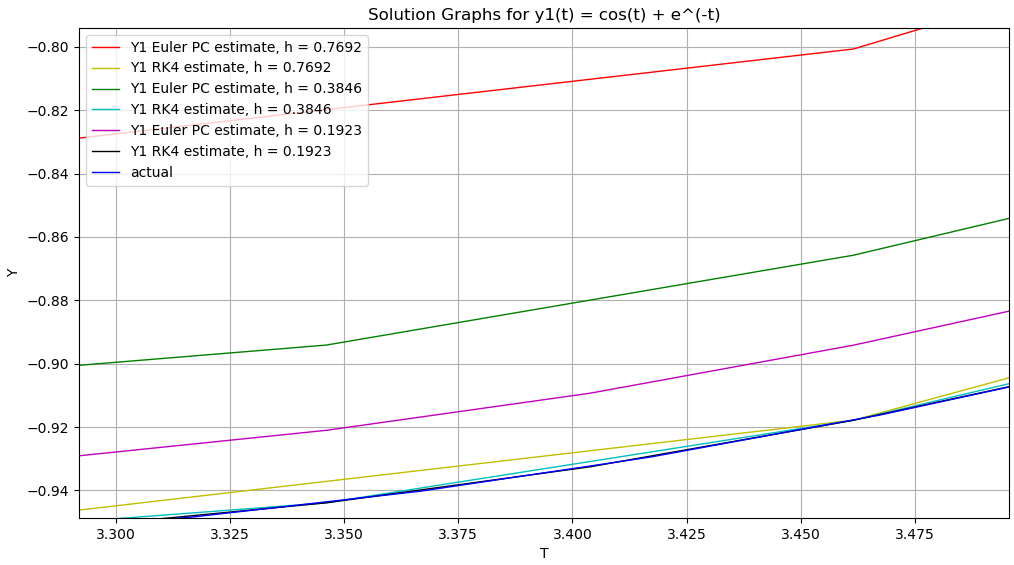
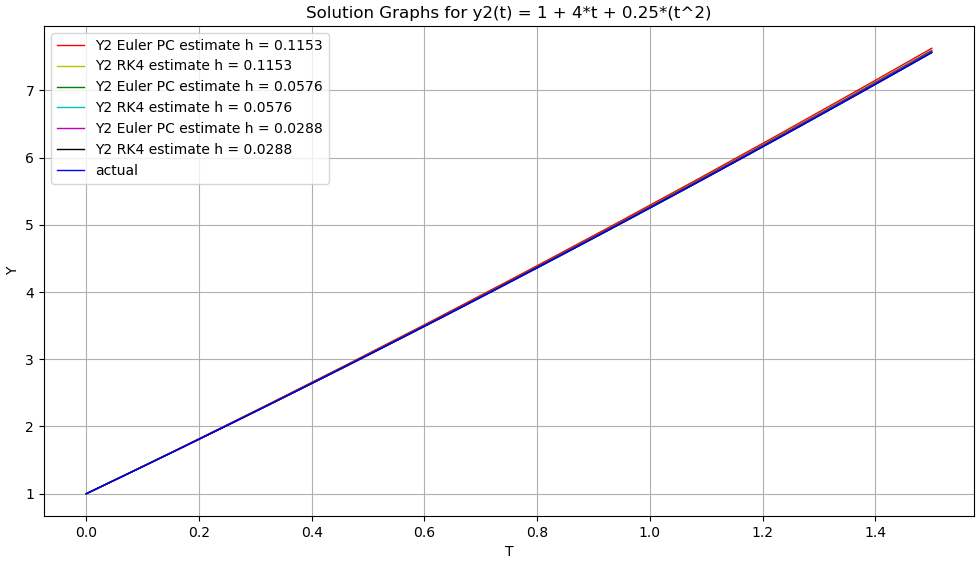


Figure 14: Part 3 Solution Graphs for y1 - Zoomed in

Figure 15 contains the subplots for the solutions to y2. The results are basically the same as the graph for y1. This is once again more evident in Figure 16, which shows a zoomed in view of the solutions. The least accurate RK4 method still outperformed the most accurate Euler's predictor-corrector method estimate. Both methods do a good job of estimating the solution, though that is not surprising considering it is basically a linear line on the interval. As before, both methods improve with smaller step sizes, and RK4's solutions seem to converge much faster than the Euler's methods' solutions. The code to create the graphs in Figure 13 and Figure 15 is shown in Figure 27 in the Appendix.



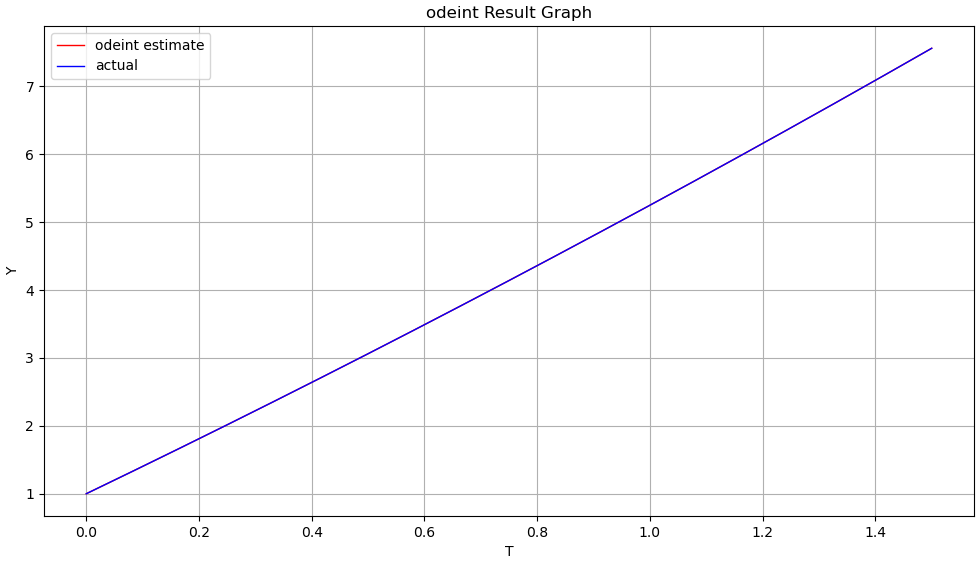


Figure 15: Part 3 Solution Graphs for y2

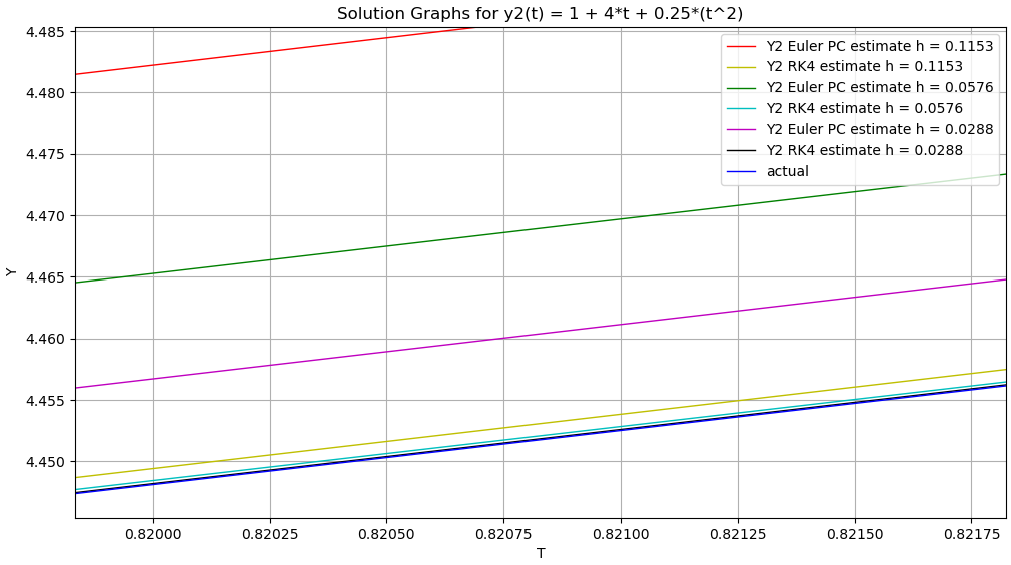


Figure 16: Part 3 Solution Graphs for y2 - Zoomed in

Figure 17 shows the total number of iterations that RK4 and Euler's predictor-corrector needed to solve y1 and y2 with certain step sizes. The number of iterations being the number of times the method looped. For the Euler predictor-corrector, this also includes iterated corrections at the same time step. The first three values for each method are the iterations for y1, while the next three are for y2. What is interesting is that aside from the estimate for y2 with h = 0.0384, where the number of iterations is the same, RK4 takes fewer iterations than the Euler's predictor-corrector method, while still being much more accurate. The number of iterations for the Euler's predictor-corrector method could probably be improved by adding better conditions for terminating the corrections. Currently, the corrections stop when the amount corrected is no longer within tolerance, or after a maximum number of iterations.

```
EulerPC (h = 0.0769): 90
RK4 (h = 0.0769): 44
EulerPC (h = 0.1153): 12
RK4 (h = 0.1153): 10
EulerPC (h = 0.0769): 139
RK4 (h = 0.0769): 87
EulerPC (h = 0.0384): 21
RK4 (h = 0.0384): 20
EulerPC (h = 0.0192): 236
RK4 (h = 0.0192): 174
EulerPC (h = 0.0384): 39
RK4 (h = 0.0384): 39
```

Figure 17: Iterations for RK4 and Euler's Predictor-Corrector

Part 4

The goal of part four was to solve the system of ODEs shown in Figure 18 using both explicit and implicit Euler's methods. In this case, explicit Euler's method was used as a predictor, while implicit Euler's method was used to find a function, g(y(t+h)), by solving implicit Euler's for zero. To make the predicted

value more accurate, the secant method was used to find the roots of g(y), which gives a better estimate for y(t+h). The implementations of g(y) and the secant method are shown in Figure 19. For the secant method, $f(x_n) = g(y(t+h))$, where y(t+h) is the current estimate of the ODE being solved (y1 or y2). Since the ODEs are a system, it was necessary to solve both problems at the same time. Overall, this method should be accurate to about $O(h^2)$. The code for this method is shown in Figure 28.

$$y_1' = 3y_1 - 37y_2, \quad y_1(0) = 16 \; on \; [0,2]$$

$$y_2' = 5y_1 - 39y_2, \quad y(0) = -16 \; on \; [0,2]$$

$$y_1 = 37e^{-2t} - 21e^{-34t}$$

$$y_2 = 5e^{-2t} - 21e^{-34t}$$

Figure 18: System of ODEs and Solutions for Part 4

$$Secant\ Method$$

$$x_n = \frac{x_{n-2}f(x_{n-1}) - x_{n-1}f(x_{n-2})}{f(x_{n-1}) - f(x_{n-2})}$$

$$f(x_n) = g\big(y_x(t+h)\big) = y_x(t+h) - y_x(t) - hy_x'(y_1(t+h), y_2(t+h))$$

Figure 19: Secant Method Implementation

Figure 20 contains three subplots; the first subplot has plots for the estimated solutions of y1; the second subplot has plots for the estimated solutions of y2; and the third subplot has solutions for both y1 and y2 estimated by odeint. All three subplots also contain plots of the exact solutions to y1 or y2.

The most significant issue with the Euler's-secant predictor-corrector method was its instability. Depending on the step-size, the entire method could break and either result in large spikes and oscillations or it could shoot off to infinity. This can partially be seen in the y1 and y2 estimates with h = 1/30, where the solutions spike, but then quickly converge back to the actual solution. This instability is likely due to the ODEs being non-stiff. Non-stiff ODEs are ones where the solutions contain large gradients, whereas a stiff ODE has a solution with relatively small gradients [1]. Implicit methods tend to perform better on stiff systems since larger step sizes can be used without causing the estimation to become unstable. On the other hand, non-stiff systems often require small step sizes, but this negates the benefits of implicit methods which are best used with larger step sizes and then iterated to make the estimations more accurate. For both y1 and y2, there are large slopes from t = 0 to t = 0.25, which is also where the estimated solutions are least accurate. This difference in accuracy is more apparent in Figure 21, which shows the same solutions zoomed in. Even for the smallest step-size, both solutions are far less accurate at the curve than anywhere else.

Aside from the issue with stability, the method performed well and became much more accurate as the values of h decreased. At the curve, the difference between the estimate for h = 1/60 is far more than 4x more accurate than h = 1/30. Other than that specific point, however, the method appears to become about twice as accurate for step sizes that are half as large.

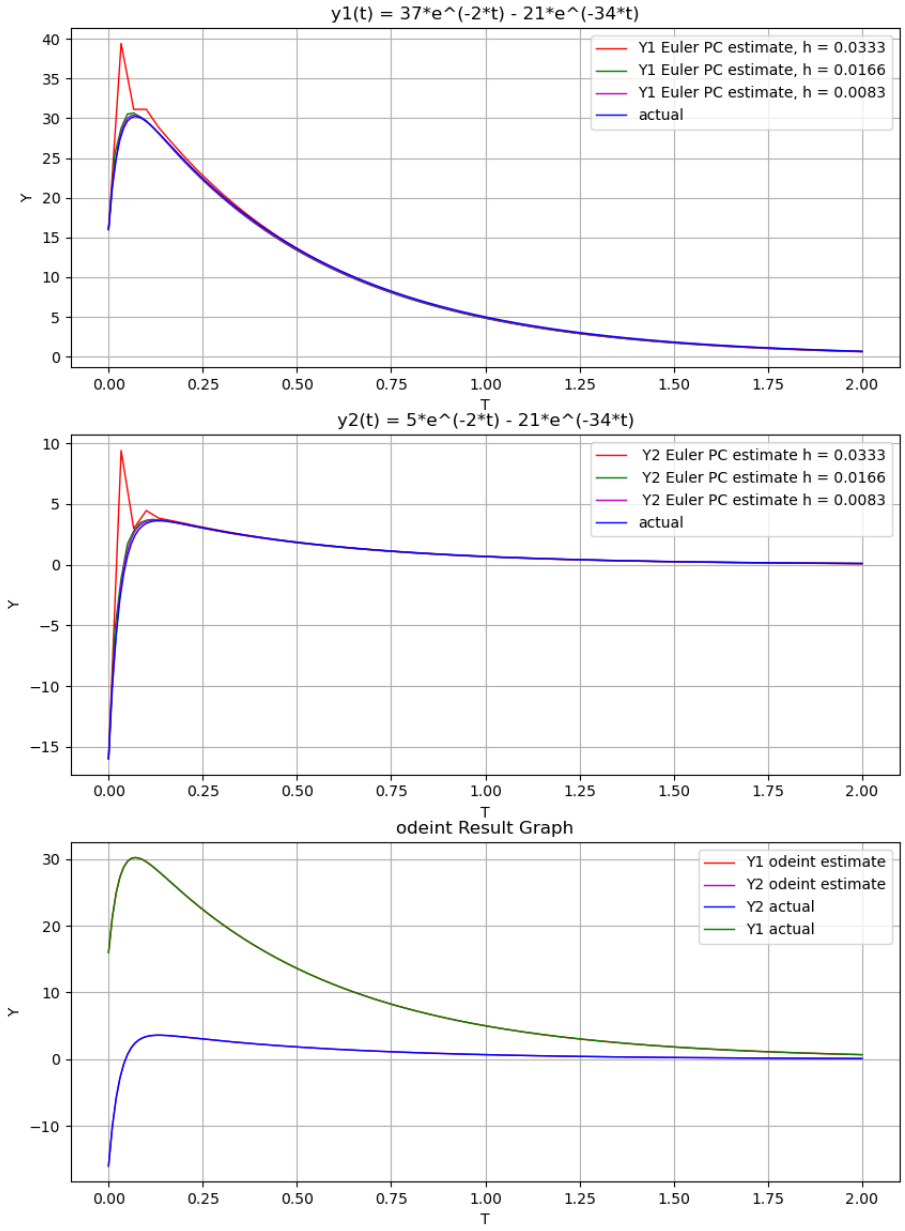


Figure 20: Part 4 Solution Graphs for y1 and y2

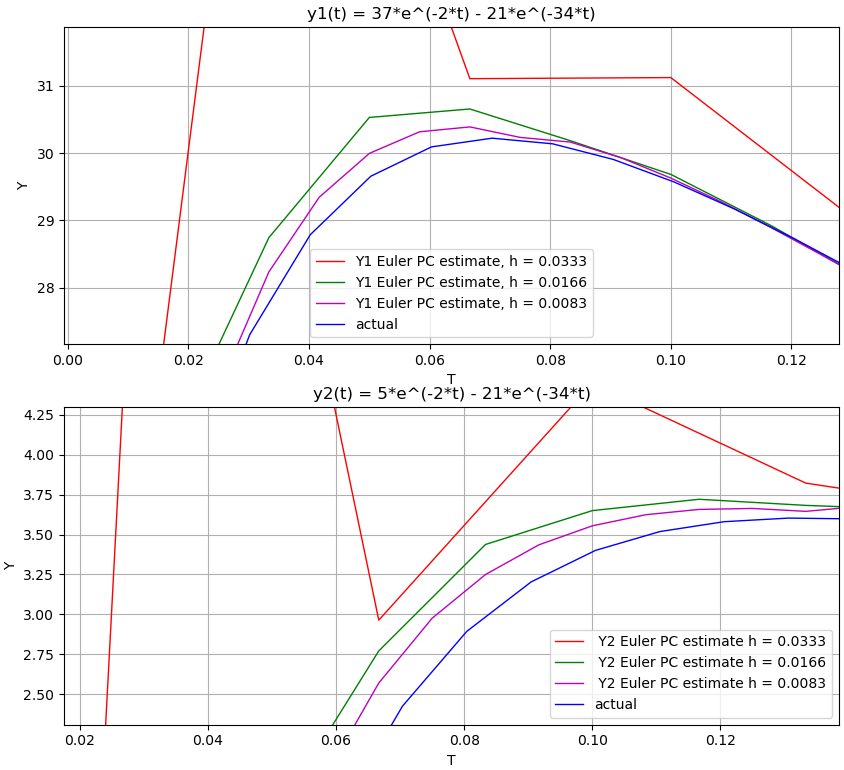


Figure 21: Part 4 Solution Graphs for y1 and y2 - Zoomed in

Appendix

```python
def f1(t,y):
    return (1 - t) * y

def f2(t,y):
    return t*(y**2)

def TrapezoidPredictorCorrector(func, bounds, y0, h, tolerance):
    tolerance_flag = False
    t_points = [bounds[0]]
    y_points = [y0]
    totalIterations = 0
    prevError = -1
    while t_points[-1] < (bounds[1] - 0.0000000000001): #Account for any rounding error
        numIters = 0
        tolerance_flag = False
        if (t_points[-1] + h > bounds[1]):#Clamp t in case h does not divide (b - a)
            h = abs(bounds[1] - t_points[-1])
        predict = y_points[-1] + h * func(t_points[-1], y_points[-1])
        y_points.append(predict)
        t_points.append(t_points[-1] + h)
        while not tolerance_flag:   #Correct until tolerance flag is set to True (by if-statement below)
            totalIterations += 1
            numIters = numIters + 1
            correct = y_points[-2] + (h/2) * (func(t_points[-1], y_points[-1]) + func(t_points[-2], y_points[-2]))
            #Stop correcting if the increase was <= tolerance, if the maximum number of iterations is reached, or if the change in error is zero
            if(not(abs(correct - y_points[-1]) > tolerance) or numIters > 1000 or abs(correct - y_points[-1]) == prevError):
                tolerance_flag = True
            prevError = abs(correct - y_points[-1])
            y_points[-1] = correct
    print("Trapezoid PC (h = " + str(h)[:6] + "): " + str(totalIterations))
    return t_points, y_points
```

Figure 22: Trapezoid PC Method Code

```
#-------------------- EQN 1 ------------------------
t1_linspace = np.linspace(0,3,200)
steps = 13
colors = ['r', 'g', 'm', 'y', 'c']
fig1 = plt.figure(1, figsize=(12,14))
fig2 = plt.figure(2, figsize=(12,14))
#Graph Functions for Y1 and Y2 with different h values
for i in range(3):
    fig1 = plt.figure(1, figsize=(12,14))
    t1, y1 = TrapezoidPredictorCorrector(f1, [0,3], 3, 3/steps, 0.0001) #Solve Y1
    plt.subplot(2,1,1)
    plt.plot(t1, y1, color = colors[i], linewidth=1, label = 'Y1 estimate, h = ' + str(2/steps)[:6])
    t2, y2 = TrapezoidPredictorCorrector(f2, [0,2], 0.4, 2/steps, 0.0001) #Solve Y2
    fig2 = plt.figure(2, figsize=(12,14))
    plt.subplot(2,1,1)
    plt.plot(t2, y2, color = colors[i], linewidth=1, label = ' Y2 estimate h = ' + str(3/steps)[:6])
    steps *= 2

#Graph Acutal Solution to Y1
fig1 = plt.figure(1, figsize=(12,14))
plt.subplot(2,1,1)
plt.grid()
plt.plot(t1_linspace, 3*e**(t1_linspace - 0.5*t1_linspace**2), color = 'b', label = 'actual', linewidth=1)
plt.title('Trapezoid PC Method for y1(t) = 3e^(t - 0.5*t^2)')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint for Y1
sol = odeint(f1, 3, t1_linspace, tfirst=True)
plt.subplot(2,1,2)
plt.grid()
plt.plot(t1_linspace, sol, color = 'r', linewidth=1, label = 'odeint() estimate')
plt.plot(t1_linspace, 3*e**(t1_linspace - 0.5*t1_linspace**2), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#-------------------- EQN 2 ------------------------
#Graph Acutal Solution to Y2
t2_linspace = np.linspace(0,2,200)
fig2 = plt.figure(2, figsize=(12,14))
plt.subplot(2,1,1)
plt.grid()
plt.plot(t2_linspace, -1/(0.5*t2_linspace**2 - 2.5), color = 'b', label = 'actual', linewidth=1)
plt.title('Trapezoid PC Method for y2(t) = -1/(0.5*t^2 - 2.5')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint for Y2
sol = odeint(f2, 0.4, t2_linspace, tfirst=True)
plt.subplot(2,1,2)
plt.grid()
plt.plot(t2_linspace, sol, color = 'r', linewidth=1, label = 'odeint() estimate')
plt.plot(t2_linspace, -1/(0.5*t2_linspace**2 - 2.5), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 23: Graphing Code for Part 1

```python
def f1(t,y):
    return (1 - t) * y

def f2(t,y):
    return t*(y**2)

def AdamsMoultonPredictorCorrector(func, bounds, y0, h, tolerance):
    tolerance_flag = False
    t_points = [bounds[0]]
    y_points = [y0]
    totalIterations = 0
    prevError = -1
    count = 0
    while t_points[-1] < (bounds[1] - 0.0000000000001): #Account for any rounding error
        numIters = 0
        tolerance_flag = False
        if (t_points[-1] + h > bounds[1]):#Clamp t in case h does not divide (b - a)
            h = abs(bounds[1] - t_points[-1])
        #Use RK4 for the first three predictions then use AB4 - AM4 predictor corrector for remaining points
        if(count < 3):                                          #Compute RK4 for first 4 points
            k1 = func(t_points[-1], y_points[-1])
            k2 = func(t_points[-1] + 0.5*(h), y_points[-1] + 0.5*(h)*k1)#Use h/3 for the first three points since RK4 is explicit
            k3 = func(t_points[-1] + 0.5*(h), y_points[-1] + 0.5*(h)*k2)
            k4 = func(t_points[-1] + (h), y_points[-1] + (h)*k3)
            y_points.append(y_points[-1] + (h)*(k1 + 2*k2 + 2*k3 + k4)/6)
            t_points.append((t_points[-1] + h))
        else: #Use AB4 for the rest of the points
            y_points.append(y_points[-1] + (h/24)*(55*func(t_points[-1], y_points[-1]) - 59*func(t_points[-2], y_points[-2]) + 37*func(t_points[-3], y_points[-3]) - 9*func(t_points[-4], y_points[-4])))
            t_points.append(t_points[-1] + h)
            while not tolerance_flag:   #Correct until tolerance flag is set to True (by if-statement below)
                totalIterations += 1
                numIters = numIters + 1
                correct = y_points[-2] + (h/24) * (9*func(t_points[-1], y_points[-1]) + 19*func(t_points[-2], y_points[-2]) - 5*func(t_points[-3], y_points[-3]) + func(t_points[-4], y_points[-4]))
                #Stop correcting if the increase was <= tolerance, if the maximum number of iterations is reached, or if the change in error is zero
                if(not(abs(correct - y_points[-1]) > tolerance) or numIters > 1000 or abs(correct - y_points[-1]) == prevError):
                    tolerance_flag = True
                prevError = abs(correct - y_points[-1])
                y_points[-1] = correct
        count += 1
    print("AM4 PC (h = " + str(h)[:6] + "): " + str(totalIterations))
    return t_points, y_points
```

Figure 24: AB4-AM4 PC Method Code

```
#-------------------- EQN 1 ------------------------

t1_linspace = np.linspace(0,3,200)
steps = 13
colors = ['r', 'g', 'm', 'y', 'c']
fig1 = plt.figure(1, figsize=(12,14))
fig2 = plt.figure(2, figsize=(12,14))

#Graph Functions for Y1 and Y2 with three h values
for i in range(3):
    fig1 = plt.figure(1, figsize=(12,14))
    t1, y1 =  AdamsMoultonPredictorCorrector(f1, [0,3], 3, 3/steps, 0.0001) #Solve Y1
    plt.subplot(2,1,1)
    plt.plot(t1, y1, color = colors[i], linewidth=1, label = 'Y1 estimate, h = ' + str(3/steps)[:6]) #Graph Y1
    t2, y2 =  AdamsMoultonPredictorCorrector(f2, [0,2], 0.4, 2/steps, 0.0001) #Solve Y2
    fig2 = plt.figure(2, figsize=(12,14))
    plt.subplot(2,1,1)
    plt.plot(t2, y2, color = colors[i], linewidth=1, label = ' Y2 estimate h = ' + str(2/steps)[:6]) #Graph Y2
    steps *= 2


fig1 = plt.figure(1, figsize=(12,14))
plt.subplot(2,1,1)
plt.grid()
plt.plot(t1_linspace, 3*e**(t1_linspace - 0.5*t1_linspace**2), color = 'b', label = 'actual', linewidth=1) #Graph Exact Y1
plt.title('AM4 PC Method for y1(t) = 3e^(t - 0.5*t^2)')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f1, 3, t1_linspace, tfirst=True)
plt.subplot(2,1,2)
plt.grid()
plt.plot(t1_linspace, sol, color = 'r', linewidth=1, label = 'odeint() estimate')
plt.plot(t1_linspace, 3*e**(t1_linspace - 0.5*t1_linspace**2), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#-------------------- EQN 2 ------------------------
t2_linspace = np.linspace(0,2,200)
fig2 = plt.figure(2, figsize=(12,14))
plt.subplot(2,1,1)
plt.grid()
plt.plot(t2_linspace, -1/(0.5*t2_linspace**2 - 2.5), color = 'b', label = 'actual', linewidth=1) #Graph Exact Y2
plt.title('AM4 PC Method for y2(t) = -1/(0.5*t^2 - 2.5')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f2, 0.4, t2_linspace, tfirst=True)
plt.subplot(2,1,2)
plt.grid()
plt.plot(t2_linspace, sol, color = 'r', linewidth=1, label = 'odeint() estimate')
plt.plot(t2_linspace, -1/(0.5*t2_linspace**2 - 2.5), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 25: Graphing Code for Part 2

```python
def f1(t,y):# Sol: np.cos(t) + e**(-t)
    return np.cos(t) - np.sin(t) - y

def f2(t,y):#Sol: 1 + 4*t + 0.25*(t**2)
    return 2 + (y - 2*t + 3)**(0.5)

def RK4(f, bounds, y0, h):
    totalIterations = 0
    if bounds[0] > bounds[1]:
        temp = bounds[0]
        bounds[0] = bounds[1]
        bounds[1] = temp
    y_points = [y0]                                         #Initialize Y values list
    t_points = [bounds[0]]                                  #Initialize T values list
    while t_points[-1] < (bounds[1] - 0.0000000000001):     #Account for any rounding error
        if (t_points[-1] + h > bounds[1]):                  #Clamp t in case h does not divide (b - a)
            h = abs(bounds[1] - t_points[-1])
        k1 = f(t_points[-1], y_points[-1])
        k2 = f(t_points[-1] + 0.5*h, y_points[-1] + 0.5*h*k1)
        k3 = f(t_points[-1] + 0.5*h, y_points[-1] + 0.5*h*k2)
        k4 = f(t_points[-1] + h, y_points[-1] + h*k3)
        y_points.append(y_points[-1] + h*(k1 + 2*k2 + 2*k3 + k4)/6)       #Gets most recent y-value and adds the derivative at most recent t-value to it
        t_points.append(t_points[-1] + h)                   #Gets most recent t-value and increments it by h
        totalIterations += 1
    print("RK4 (h = " + str(h)[:6] + "): " + str(totalIterations))
    return t_points, y_points

def EulerPC(f, bounds, y0, h, tolerance):
    y_points = [y0]
    t_points = [bounds[0]]
    numIters = 0
    totalIterations = 0
    prevError = -1
    while t_points[-1] < (bounds[1] - 0.0000000000001):     #Account for any rounding error
        tolerance_flag = False
        if (t_points[-1] + h > bounds[1]):                  #Clamp t in case h does not divide (b - a)
            h = abs(bounds[1] - t_points[-1])
        #Predict: (Fwd Euler)
        y_points.append(y_points[-1] + h*f(t_points[-1], y_points[-1]))
        t_points.append(t_points[-1] + h)
        #Correct:
        while not tolerance_flag:   #Correct until tolerance flag is set to True (by if-statement below)
            totalIterations += 1
            numIters += 1
            #Implicit Euler, no root finding
            correct = y_points[-2] + h*f(t_points[-1], y_points[-1])
            #Stop correcting if the increase was <= tolerance, if the maximum number of iterations is reached, or if the change in error is >= 0
            if(not(abs(correct - y_points[-1]) > tolerance) or numIters > 1000 or abs(correct - y_points[-1]) >= prevError):
                tolerance_flag = True
            prevError = abs(correct - y_points[-1])
            y_points[-1] = correct

    print("EulerPC (h = " + str(h)[:6] + "): " + str(totalIterations))
    return t_points, y_points
```

Figure 26: RK4 and Euler's PC Method Code

```
#-------------------- EQN 1 ------------------------

t1_linspace = np.linspace(0,10,200)
steps = 13
colors1 = ['r', 'g', 'm']
colors2 = ['y', 'c', 'k']
fig1 = plt.figure(1, figsize=(12,14))
fig2 = plt.figure(2, figsize=(12,14))
#Graph Functions for Y1 and Y2 with three h values
for i in range(3):
    fig1 = plt.figure(1, figsize=(12,14))
    #Solve Y1 Euler PC
    t1, y1 =  EulerPC(f1, [0,10], 2, 3/steps, 0.0001)
    plt.subplot(2,1,1)
    #Graph Y1 Euler PC
    plt.plot(t1, y1, color = colors1[i], linewidth=1, label = 'Y1 Euler PC estimate, h = ' + str(10/steps)[:6])
    #Solve Y1 RK4
    t1, y1 =  RK4(f1, [0,10], 2, 3/steps)
    #Graph Y1 RK4
    plt.plot(t1, y1, color = colors2[i], linewidth=1, label = 'Y1 RK4 estimate, h = ' + str(10/steps)[:6])
    #Solve Y2 Euler PC
    t2, y2 =  EulerPC(f2, [0,1.5], 1, 2/steps, 0.0001)
    fig2 = plt.figure(2, figsize=(12,14))
    plt.subplot(2,1,1)
    #Graph Y2 Euler PC
    plt.plot(t2, y2, color = colors1[i], linewidth=1, label = 'Y2 Euler PC estimate h = ' + str(1.5/steps)[:6])
    #Solve Y2 RK4
    t2, y2 =  RK4(f2, [0,1.5], 1, 2/steps)
    #Graph Y2 RK4
    plt.plot(t2, y2, color = colors2[i], linewidth=1, label = 'Y2 RK4 estimate h = ' + str(1.5/steps)[:6])
    steps *= 2


fig1 = plt.figure(1, figsize=(12,14))
plt.subplot(2,1,1)
plt.grid()
plt.plot(t1_linspace, np.cos(t1_linspace) + e**(-t1_linspace), color = 'b', label = 'actual', linewidth=1) #Graph Exact Y1
plt.title('Solution Graphs for y1(t) = cos(t) + e^(-t)')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f1, 2, t1_linspace, tfirst=True)
plt.subplot(2,1,2)
plt.grid()
plt.plot(t1_linspace, sol, color = 'r', linewidth=1, label = 'odeint estimate')
plt.plot(t1_linspace, np.cos(t1_linspace) + e**(-t1_linspace), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#-------------------- EQN 2 ------------------------
t2_linspace = np.linspace(0,1.5,200)
fig2 = plt.figure(2, figsize=(12,14))
plt.subplot(2,1,1)
plt.grid()
plt.plot(t2_linspace, 1 + 4*t2_linspace + 0.25*(t2_linspace**2), color = 'b', label = 'actual', linewidth=1) #Graph Exact Y2
plt.title('Solution Graphs for y2(t) = 1 + 4*t + 0.25*(t^2)')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f2, 1, t2_linspace, tfirst=True)
plt.subplot(2,1,2)
plt.grid()
plt.plot(t2_linspace, sol, color = 'r', linewidth=1, label = 'odeint estimate')
plt.plot(t2_linspace, 1 + 4*t2_linspace + 0.25*(t2_linspace**2), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 27: Graphing Code for Part 3

```python
def y1_prime(y1, y2):
    return 3*y1 - 37*y2

def y2_prime(y1, y2):
    return 5*y1 - 39*y2

def y_prime_sys(y, t = 0):
    y1_prime = 3*y[0] - 37*y[1]
    y2_prime = 5*y[0] - 39*y[1]
    return [y1_prime, y2_prime]

def EulerPC(f1, f2, bounds, y10, y20, h, tolerance):
    y1_points = [y10]
    y2_points = [y20]
    y1 = 0
    y2 = 0
    t_points = [bounds[0]]
    numIters = 0
    totalIterations = 0
    prevError1 = -1
    prevError2 = -1
    while t_points[-1] < (bounds[1] - 0.0000000000001):                 #Account for any rounding error
        tolerance_flag = False
        if (t_points[-1] + h > bounds[1]):                              #Clamp t in case h does not divide (b - a)
            h = abs(bounds[1] - t_points[-1])
        #Predict: (Fwd Euler)
        y1 = (y1_points[-1] + h*f1(y1_points[-1], y2_points[-1]))
        y2 = (y2_points[-1] + h*f2(y1_points[-1], y2_points[-1]))
        y1_points.append(y1)
        y2_points.append(y2)

        t_points.append(t_points[-1] + h)

        #Get initial prediction with implicit Euler
        y1 = y1_points[-2] + h*f1(y1_points[-1], y2_points[-1])
        y2 = y2_points[-2] + h*f2(y1_points[-1], y2_points[-1])

        while not tolerance_flag:   #Correct until tolerance flag is set to True (by if-statement below)
            totalIterations += 1
            numIters += 1
            #Improve implicit Euler prediction with Secant method on g(y) = y(t+h) - y(t) - h*f(y1(t+h),y2(t+h))
            correct1 = (y1_points[-1] * (y1 - y1_points[-1] - h * f1(y1, y2)) - y1 * (y1_points[-1] - y1_points[-2] - h * f1(y1, y2))) / \
                       ((y1 - y1_points[-1] - h * f1(y1, y2)) - (y1_points[-1] - y1_points[-2] - h * f1(y1, y2)))
            correct2 = (y2_points[-1] * (y2 - y2_points[-1] - h * f2(y1, y2)) - y2 * (y2_points[-1] - y2_points[-2] - h * f2(y1, y2))) / \
                       ((y2 - y2_points[-1] - h * f2(y1, y2)) - (y2_points[-1] - y2_points[-2] - h * f2(y1, y2)))

            #Stop correcting if the increase was <= tolerance, if the maximum number of iterations is reached, or if the change in error is >= 0
            if(((not(abs(correct1 - y1_points[-1]) > tolerance) or abs(correct1 - y1_points[-1]) >= prevError1) or \
                (not(abs(correct2 - y2_points[-1]) > tolerance) or abs(correct2 - y2_points[-1]) >= prevError2)) or numIters > 50 ):
                tolerance_flag = True

            prevError1 = abs(correct1 - y1_points[-1])
            prevError2 = abs(correct2 - y2_points[-1])
            y1_points[-1] = correct1
            y2_points[-1] = correct2

    print("EulerPC (h = " + str(h)[:6] + "): " + str(totalIterations))
    return t_points, y1_points, y2_points
```

Figure 28: Euler PC Method Code for System of Two ODEs

```
colors = ['r', 'g', 'm', 'y', 'c']
steps = 60
t = np.linspace(0,2,200)
fig1 = plt.figure(1, figsize=(10,14))
y1 = 0
y2 = 0

for i in range(3):
    t1, y1, y2 = EulerPC(y1_prime, y2_prime, [0,2], 16, -16, 2/steps, 0.0001) #Solve with Euler PC
    plt.subplot(3,1,1)
    plt.plot(t1, y1, color = colors[i], linewidth=1, label = 'Y1 Euler PC estimate, h = ' + str(2/steps)[:6]) #Plot y1
    plt.subplot(3,1,2)
    plt.plot(t1, y2, color = colors[i], linewidth=1, label = ' Y2 Euler PC estimate h = ' + str(2/steps)[:6]) #Plot y2
    steps *= 2

plt.subplot(3,1,1)
plt.plot(t, 37*e**(-2*t) - 21*e**(-34*t), color = 'b', label = 'actual', linewidth=1) #Plot exact y1
plt.grid()
plt.title('y1(t) = 37*e^(-2*t) - 21*e^(-34*t)')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

plt.subplot(3,1,2)
plt.grid()
plt.plot(t, 5*e**(-2*t) - 21*e**(-34*t), color = 'b', label = 'actual', linewidth=1) #Plot exact y2
plt.title('y2(t) = 5*e^(-2*t) - 21*e^(-34*t)')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Plot odeint solution
sol = odeint(y_prime_sys, [16, -16], t)
plt.subplot(3,1,3)
plt.grid()
plt.plot(t, sol[:,0], color = 'r', linewidth=1, label = 'Y1 odeint estimate')
plt.plot(t, sol[:,1], color = 'm', linewidth=1, label = 'Y2 odeint estimate')
plt.plot(t, 5*e**(-2*t) - 21*e**(-34*t), color = 'b', label = 'Y2 actual', linewidth=1)
plt.plot(t, 37*e**(-2*t) - 21*e**(-34*t), color = 'g', label = 'Y1 actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 29: Graphing Code for Part 4

References

[1] C. Moler, “Stiff Differential Equations,” MathWorks, https://www.mathworks.com/company/technical-articles/stiff-differential-equations.html (accessed Mar. 6, 2024).