2/19/2024

Project 2 Report

MATH 452

Cameron Robinson

Table of Contents

Table of Figures

Project 2

Part 1

The objective of part 1 was to implement a 3rd order Taylor Series method for solving ODEs and to use that method to calculate and graph the solutions to the initial-value ODEs shown in Figure 1.

$$1)\ y' = y + 2te^{2t}, \quad y(0) = 3 \text{ on } [0,1]$$

$$2)\ y' = 2t - 3y + 1, \quad y(1) = 5 \text{ on } [1,2]$$

Figure 1: ODEs for Part 1

The 3rd order Taylor Series method is implemented with the equation:

$$y(t+h) = y(t) + hf(t,y) + \frac{h^2}{2}f'(t,y) + \frac{h^3}{6}f''(t,y) + O(h^4), where\ y' = f(t,y)$$

The method estimates the function y(t) at the next step by summing the y(t) and the first three derivatives of the function y(t). Since the method was implemented with the first three derivatives of y(t), the error of the method is O(h^4). The code for the implementation is shown in Figure 24. Both of the functions in Figure 1 were solved by hand to get the exact solution. The numerical approximations were graphed with step sizes of h = 0.5 and 0.1 along with the graph of the exact solution. Additionally, the built-in ODE solver function, odeint(), was used to calculate the solutions to the ODEs and the result was also graphed with the exact solution. The code to create each of the graphs is shown in Figure 25 – 26.

Figure 2 shows the graphs for the solution to the first ODE. The exact solution is:

$$y(t) = 2te^{2t} - 2e^{2t} + 5e^t$$

The graph for h = 0.5 is not very accurate, which is expected since the estimate is made with three points. The h = 0.1 graph is much more accurate and is nearly indistinguishable from the exact answer, though it appears to become less accurate towards the right endpoint since parts of the red line are visible. Since the error of the method is O(h^4), the h = 0.1 graph should be about 5^4 times more accurate. The estimate calculated using odeint() appears to be slightly more accurate than the h = 0.1 graph, but not by much.

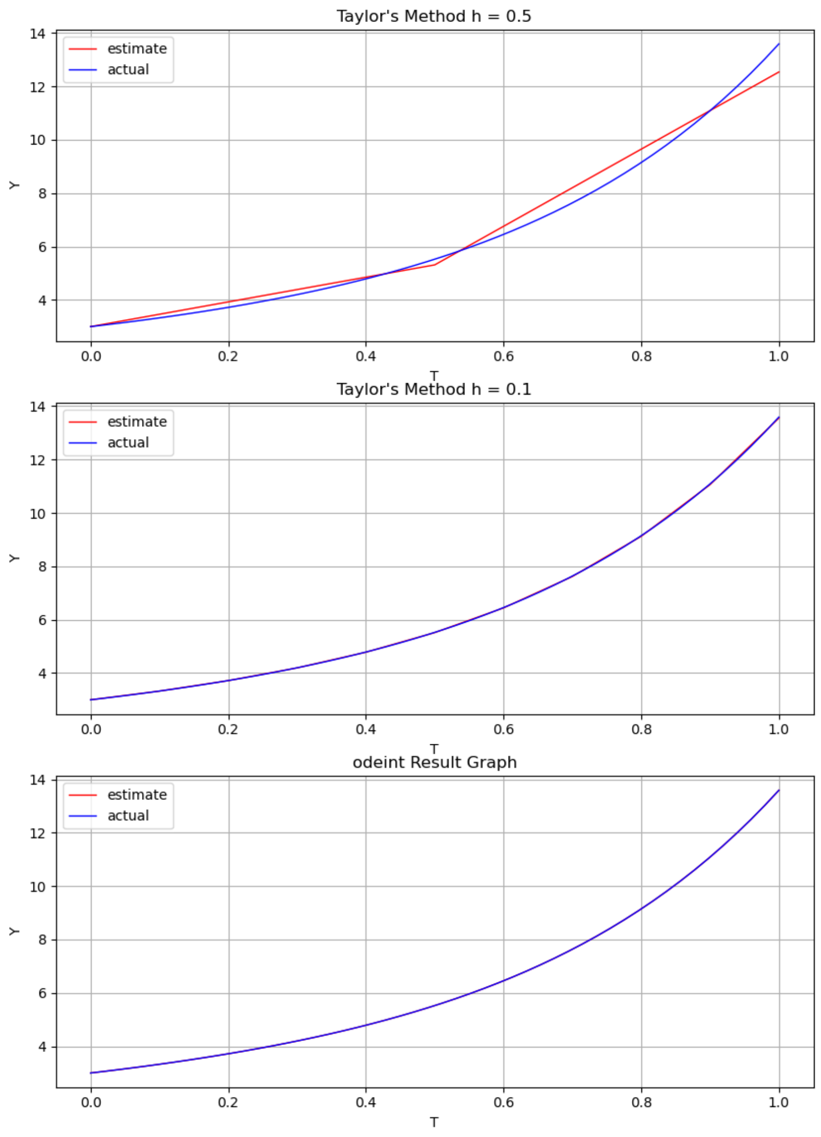


Figure 2: Graphs for Solutions to y' = 2te^(2t) - 2e^(2t) + 5e^t

Figure 3 shows the graphs for the solutions to the second ODE. The exact solution, calculated by hand is:

$$y(t) = \frac{1}{9} + \frac{2}{3}t + \frac{38}{9}e^{(3-3t)}$$

Like the first equation the solutions were calculated and graphed for h = 0.5 and 0.1. The third subplot graphs the solution calculated by the odeint() function. The blue line of each plot is the actual solution to the ODE. As with the first equation, h = 0.5 estimate has a lot of error, while h = 0.1 estimate is very accurate. One thing that is interesting is that the most error for the h = 0.1 graph is at the left endpoint rather than the right endpoint as it was for the first ODE. This seems to indicate that the Taylor Series method does not perform as well predicting steep slopes. Once again, the solution from odeint() appears to be more accurate than the Taylor Series estimations.

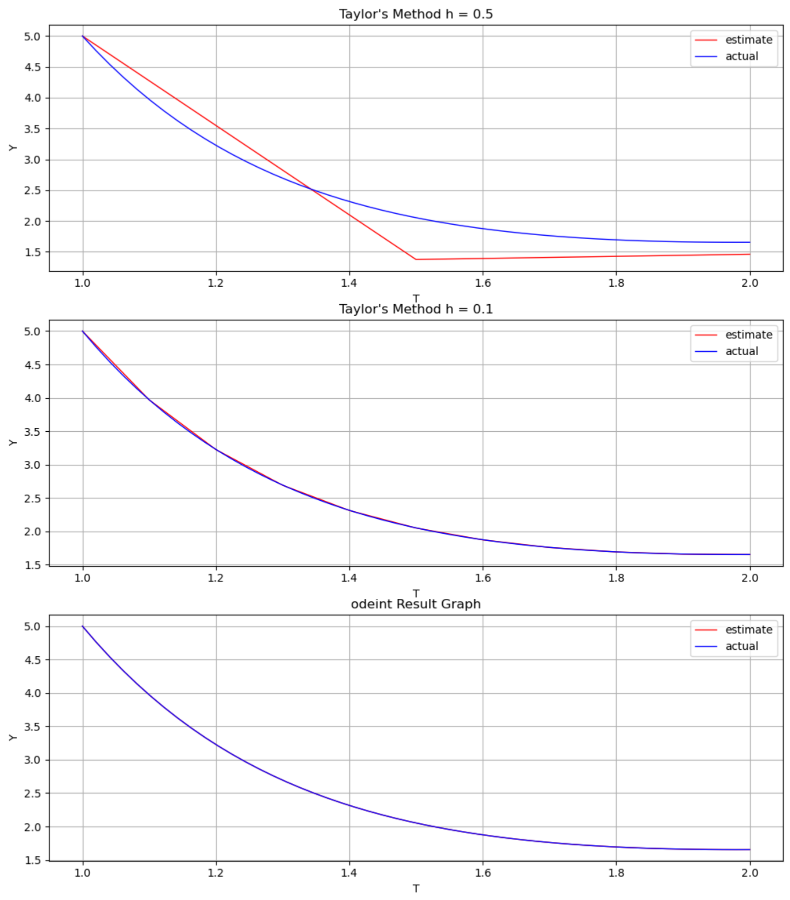


Figure 3: Graphs for Solutions to y' = y + 2te^(2t)

Part 2

The goal of part 2 was to implement at least one order-two Runge-Kutta method and use it to solve the ODEs shown in Figure 4. The first two ODEs are the same as the ones solved in part 1, the third one is a separable ODE with an exact solution of $y(t) = 3e^{(-e^t+1)}$.

$$1)\ y' = y + 2te^{2t}, \qquad y(0) = 3 \text{ on } [0,1]$$

$$2)\ y' = 2t - 3y + 1, \quad y(1) = 5 \text{ on } [1,2]$$

$$3)\ y' = -e^t y, \qquad y(0) = 3 \text{ on } [0,1]$$

Figure 4: ODEs for Part 2

In this project, two RK2 methods were implemented, the modified Euler's method and the midpoint method. The implementation equations for the methods are illustrated in Figure 5. Both methods have O(h^3) error, so they should perform worse on the first two ODEs than the Taylor's method from part 1. The code for each method is shown in Figure 27.

$RK2\ Modified\ Euler's\ Method$

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

$k2 = f(t + h, y(t) + h\, k1)$

$y(t + h) = y(t) + \frac{1}{2}h(k1 + k2)$

$RK2\ Midpoint\ Method$

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

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

$y(t + h) = y(t) + h\, k2$

Figure 5: RK2 Modified Euler and Midpoint Method Algorithms

The solutions to each of the three equations were calculated and graphed using both RK2 methods with h = 0.5 and 0.1, and the built-in odeint() function. In each case the exact solutions calculated by hand were graphed alongside the estimations. The code to create the graphs is shown in Figures 28 – 30.

Figure 6 contains the graphs for the first ODE. As expected, the RK2 estimates with h = 0.5 are not very accurate, however, it is interesting that the RK2 midpoint method is more accurate than the modified Euler's method at every point except for those around the pivot point in the middle and the right endpoint. On the h = 0.1 graph it is not clear which of the RK2 methods are more accurate, but like Taylor's method, they do not appear to be more accurate than the odeint() function, especially near the right endpoint.

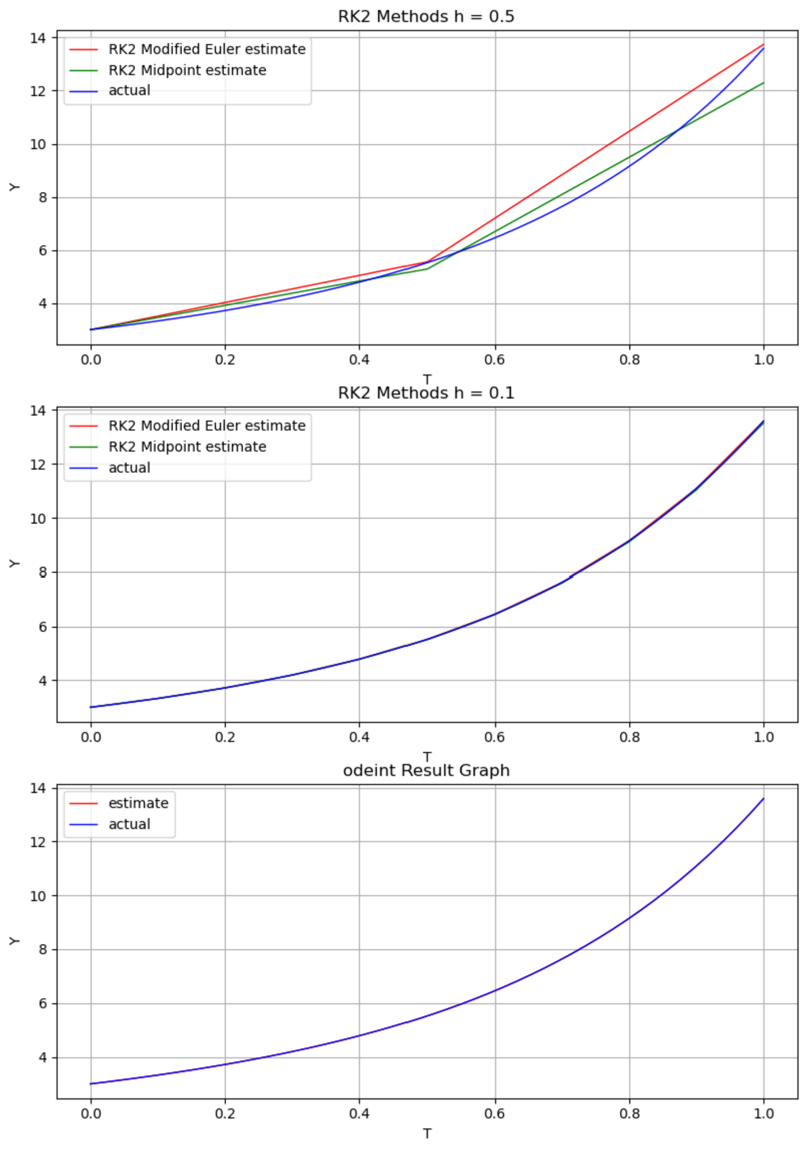


Figure 6: Part 2 Graphs for Solutions to ODE 1

Figure 7 shows the graphs for the second ODE. With a step size of 0.5, both RK2 methods performed very poorly on this ODE, though it is interesting that they end up with nearly the same estimations. The red line for the modified Euler's method is directly below the green line for the midpoint method; they were initially graphed separately to verify that this was the case. With h = 0.1, the RK2 methods performed much better, though they clearly have more error than odeint(), and it appears that they are less accurate than the Taylor's method graph from Figure 3, which is expected since the RK2 method have larger error.

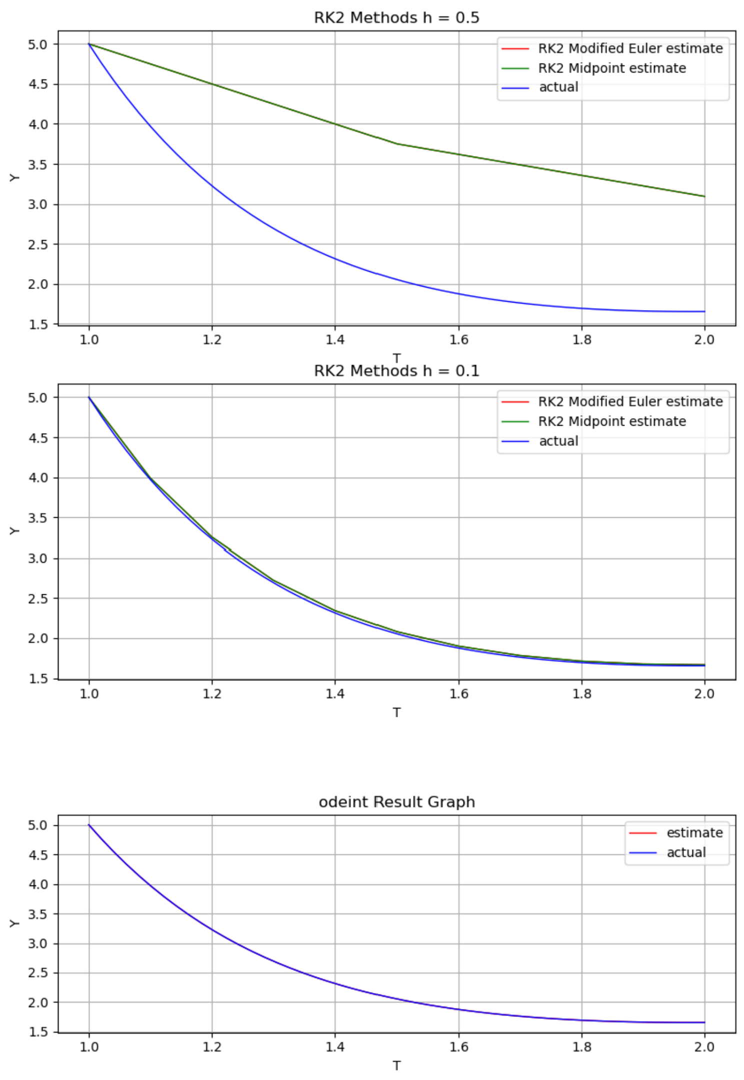


Figure 7: Part 2 Graphs for Solutions to ODE 2

Figure 8 shows the graphs of the estimated solutions for the third ODE. As before, the h = 0.5 graphs are not accurate, but once again indicate that the midpoint method tends to be more accurate than the modified Euler's method. This is true even at the pivot point and right endpoint unlike with the first ODE. With h = 0.1, both RK2 methods performed very well; so much that it is not clear if they are more or less accurate than the odeint() function.

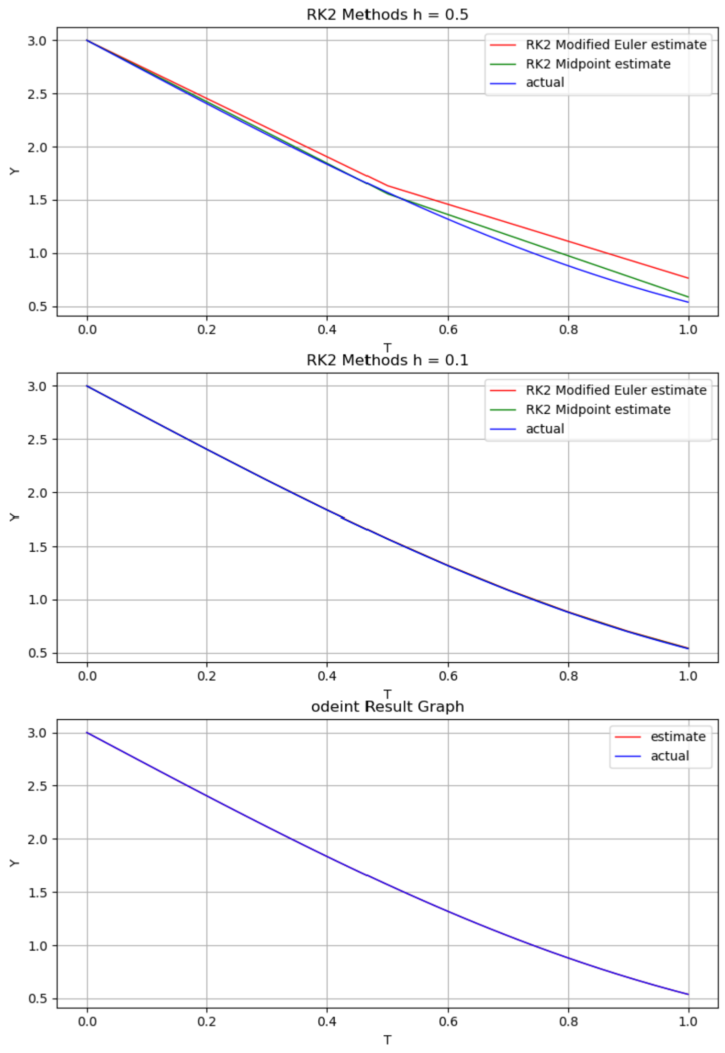


Figure 8: Part 2 Graphs for Solutions to ODE 3

Part 3

The objective of part 3 was to implement an order-four Runge-Kutta method (RK4) to numerically solve all four ODEs illustrated in Figure 9. The first three ODEs are the same ones from the previous parts. Like the third ODE, the fourth ODE is also separable; its solution is $y(t) = 2e^{-\sin(t)}$.

1) $y' = y + 2te^{2t}, \qquad y(0) = 3 \text{ on } [0,1]$

2) $y' = 2t - 3y + 1, \qquad y(1) = 5 \text{ on } [1,2]$

3) $y' = -e^{t}y, \qquad y(0) = 3 \text{ on } [0,1]$

4) $y' = -\cos(t)y, \qquad y(0) = 2 \text{ on } [0,\frac{\pi}{3}]$

Figure 9: ODEs for Part 3 and Part 4

The implementation of the RK4 method is shown in Figure 10. As implied by the name, the RK4 method has an error of O(h^5), which makes it more accurate than both the RK2 methods and Taylor's method. The code for the RK4 implementation is shown in Figure 31.

$$RK4$$

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

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

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

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

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

Figure 10: Runge-Kutta of Order 4 Implementation

Same as the other parts, RK4 and odeint() were used to solve all of the ODEs, and the estimated solutions were graphed alongside the exact solutions. The RK4 method was graphed with step sizes of 0.5 and 0.1. The code to create each of the graphs is in Figures 32 – 35.

Figure 11 shows the graphs for the first ODE. As always, the h = 0.5 graph is not very accurate between the calculated points, but that makes sense because two straight lines are not going to estimate a curved line very well. Based on the graphs, the Taylor's method estimate for h = 0.5 is more accurate than the RK4 method's estimate. This appears to be due to Taylor's method underestimating both the middle point and the right endpoint which made the estimation more accurate for the curved line in between those points. Similarly, RK4 with h = 0.5 seems to perform worse than the RK2 midpoint method and about as well as the RK2 modified Euler's method. For h = 0.1, it is difficult to tell if RK4 is much more accurate than any of the previous methods, but it still seems to be less accurate than the odeint() estimation.

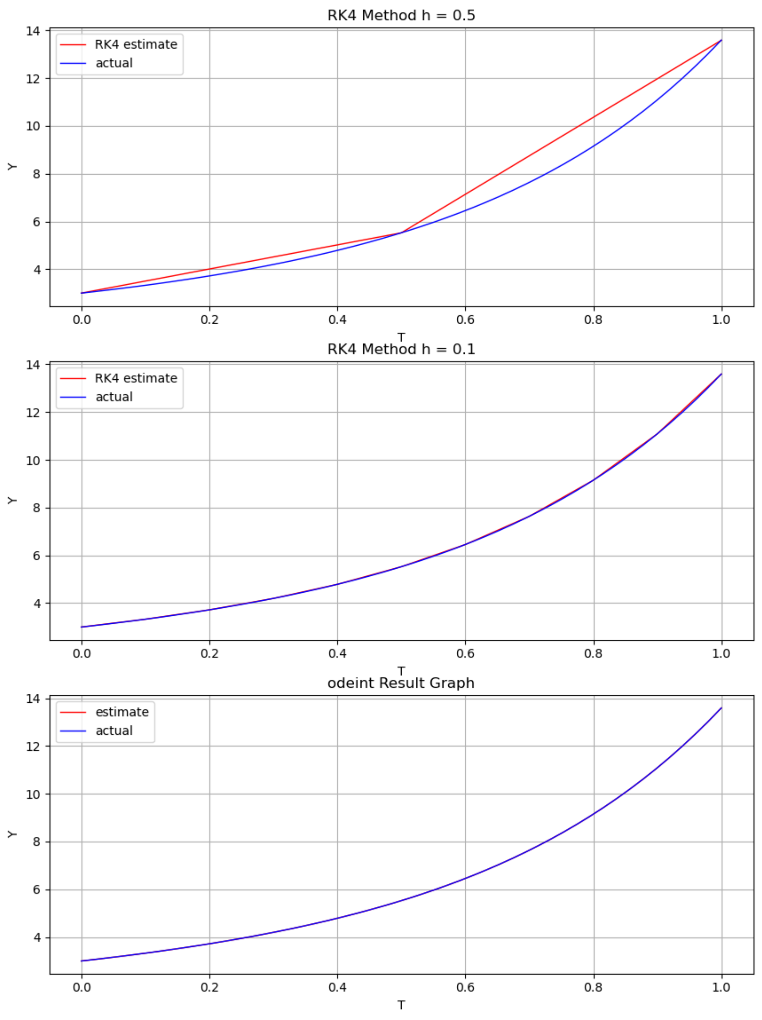


Figure 11: Part 3 Solution Graphs for ODE 1

Figure 12 shows the RK4 graphs for the second ODE. RK4 performed much better than the RK2 methods when h = 0.5, and it seems to perform slightly better than the RK2 methods with h = 0.1. With h = 0.1, RK4 appears to perform just as well as Taylor's method.

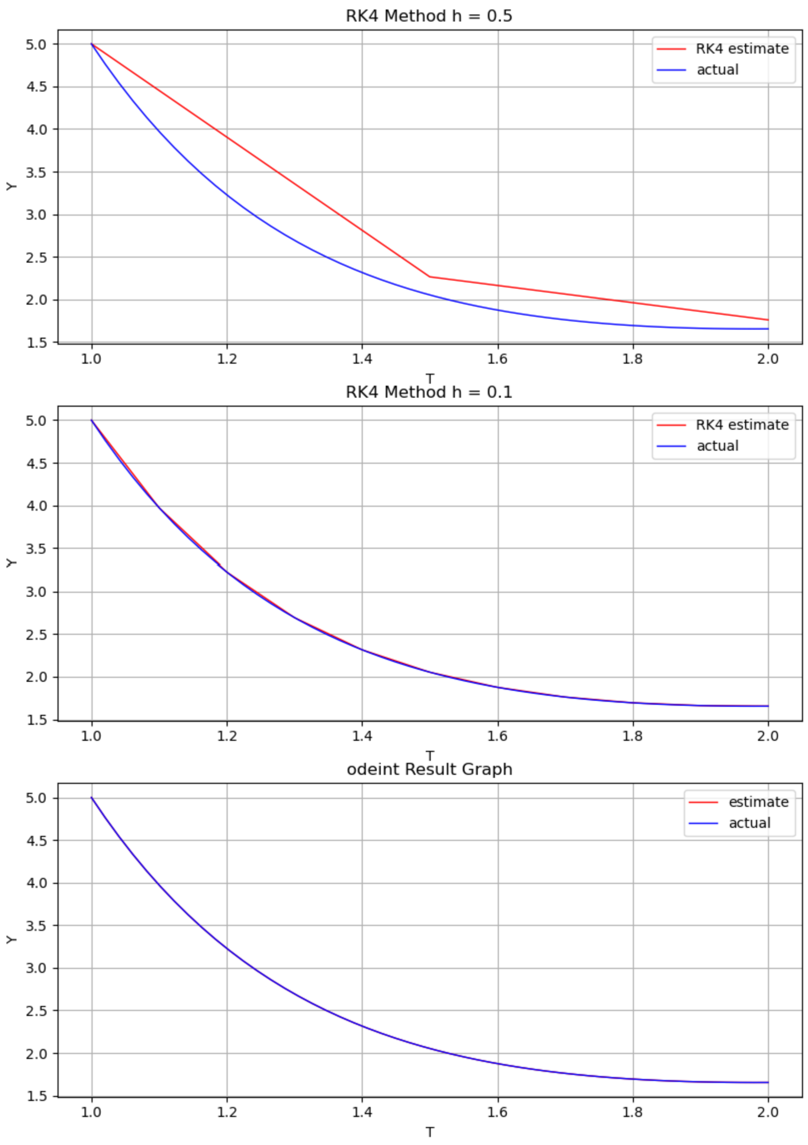


Figure 12: Part 3 Solution Graphs for ODE 2

Figure 13 contains the graphs for the third ODE. For h = 0.5, RK4 performed better than the RK2 modified Euler's method, but is nearly identical to the RK2 midpoint method, however, RK4 seems to perform better at the right endpoint. Like the RK2 methods, RK4 performs about as well as the odeint() function for h = 0.1.

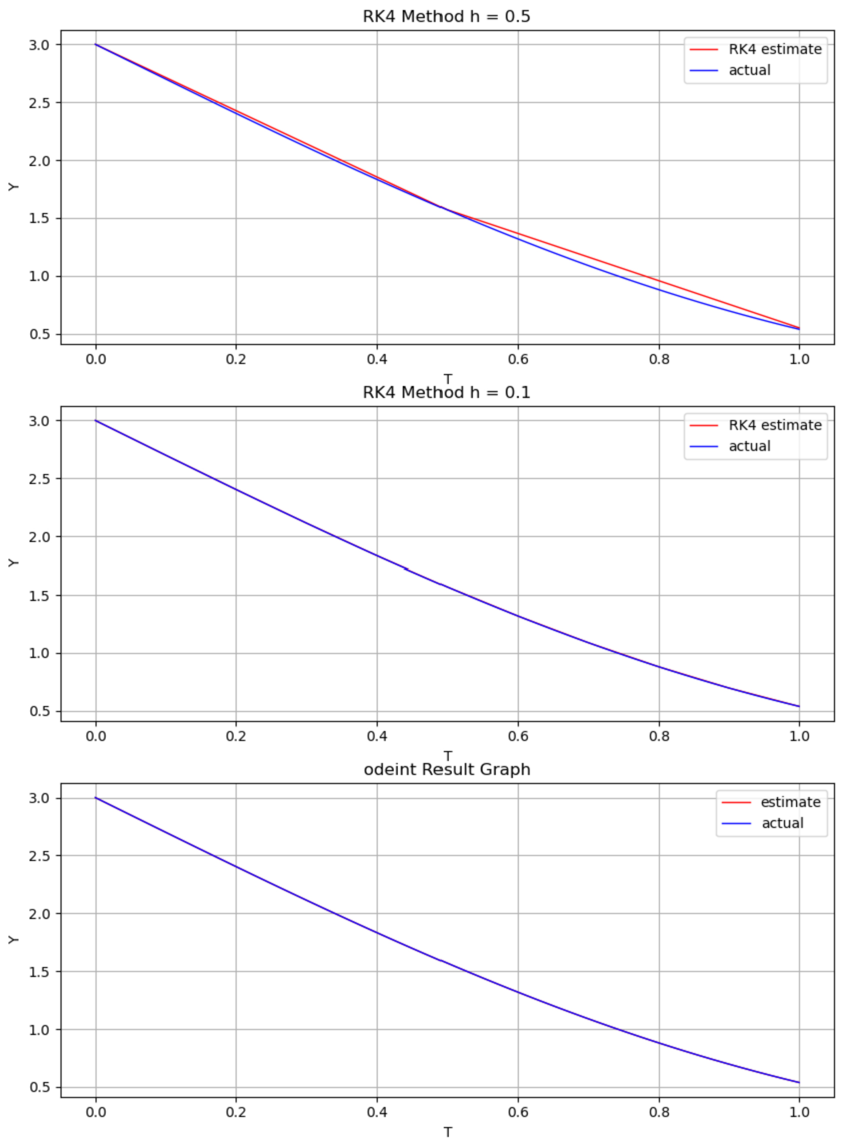


Figure 13: Part 3 Solution Graphs for ODE 3

Figure 14 contains the graphs for the fourth ODE. As with the other functions, with h = 0.5 RK4 is very accurate at the points where it calculated an estimate, the middle point and right endpoint. With h = 0.1, RK4 again performs about as well as odeint() and is indistinguishable from the actual solution.

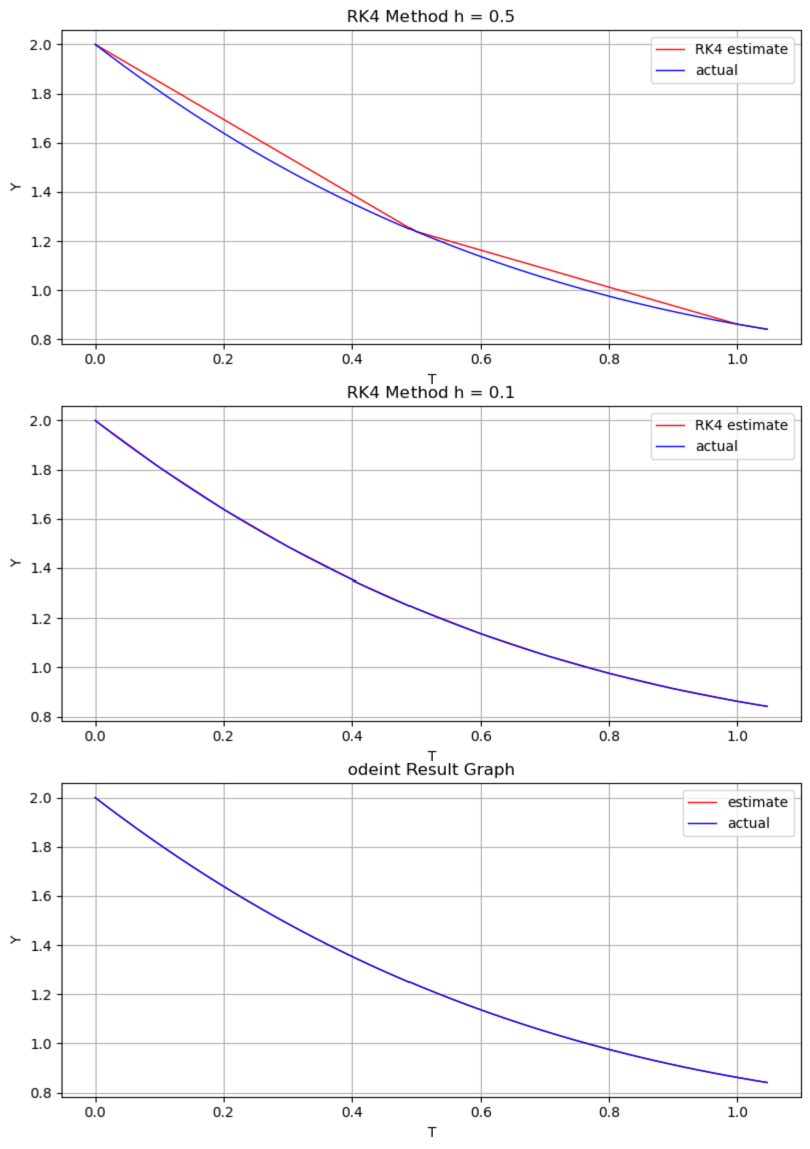


Figure 14: Part 3 Solution Graphs for ODE 4

Part 4

The goal of part 4 was the exact same as part 3, only that the method implemented to numerically solve the ODEs in Figure 9 was order four Adams-Bashforth (AB4). The equation for the AB4 method is shown in Figure 15; the method is accurate to O(h^5), which means it has about the same accuracy as RK4. This method is derived from Taylor series and Newton's backward divided difference.

$AB4$

$$y(t+h) = y(t) + (\tfrac{h}{24})(55f(t,y(t)) - 59f(t-h,y(t-h)) + 37f(t-2h,y(t-2h)) - 9f(t-3h,y(t-3h)))$$

Figure 15: AB4 Equation

Perhaps the largest issue with AB4 is that it requires at least four points of the solution to calculate the point at the next time-step. This means that another method is required to calculate the first three points of the solution before AB4 can be used. For this project, RK4 was used to calculate the first three estimates since it has the same error as AB4 but does not need future or past points to calculate the next estimate. As before, AB4 was used to solve the ODEs, and the estimated solutions were graphed alongside the actual solution. The code for the AB4 implementation is shown in Figure 36. Because AB4 can only be used on the fifth data point (fourth estimated point), the solutions were calculated with step sizes of 0.2 and 0.1, as apposed to the other methods which were graphed with h = 0.5 and 0.1. With h = 0.5, only three or four points would be graphed for any of the ODEs, so the graphs would just show RK4 estimates.

The code to create each of the graphs is shown in Figures 37 – 40. Overall, there is not anything new about any of the graphs. AB4 was fairly accurate for h = 0.2, and appears about as accurate as RK4 for h = 0.1. AB4 appears to be most accurate when the slope of the function it is estimating is small.

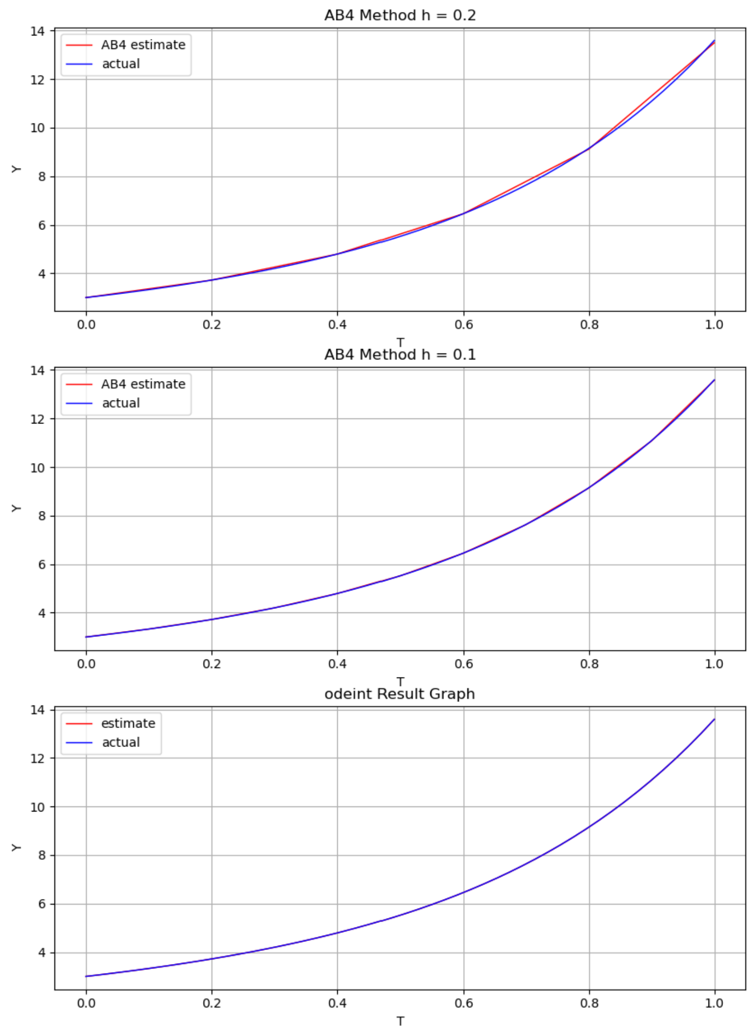


Figure 16: AB4 Solutions to ODE 1

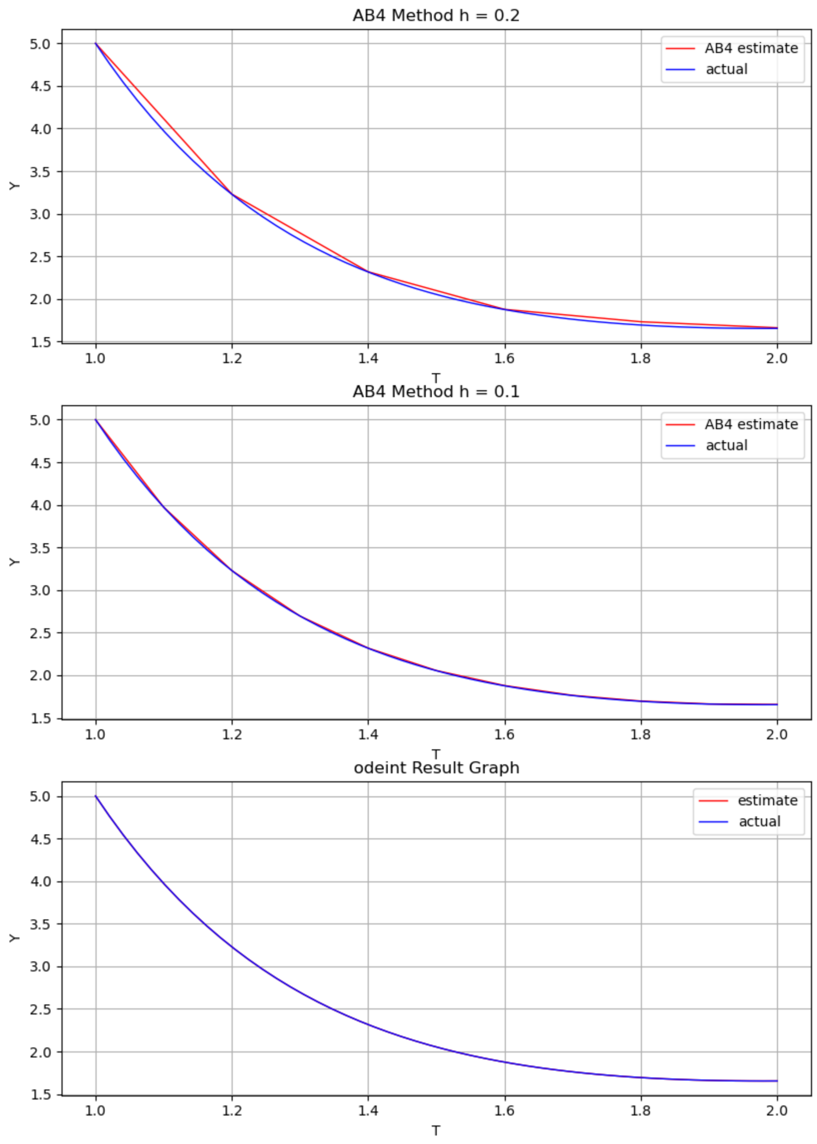


Figure 17: AB4 Solutions to ODE 2

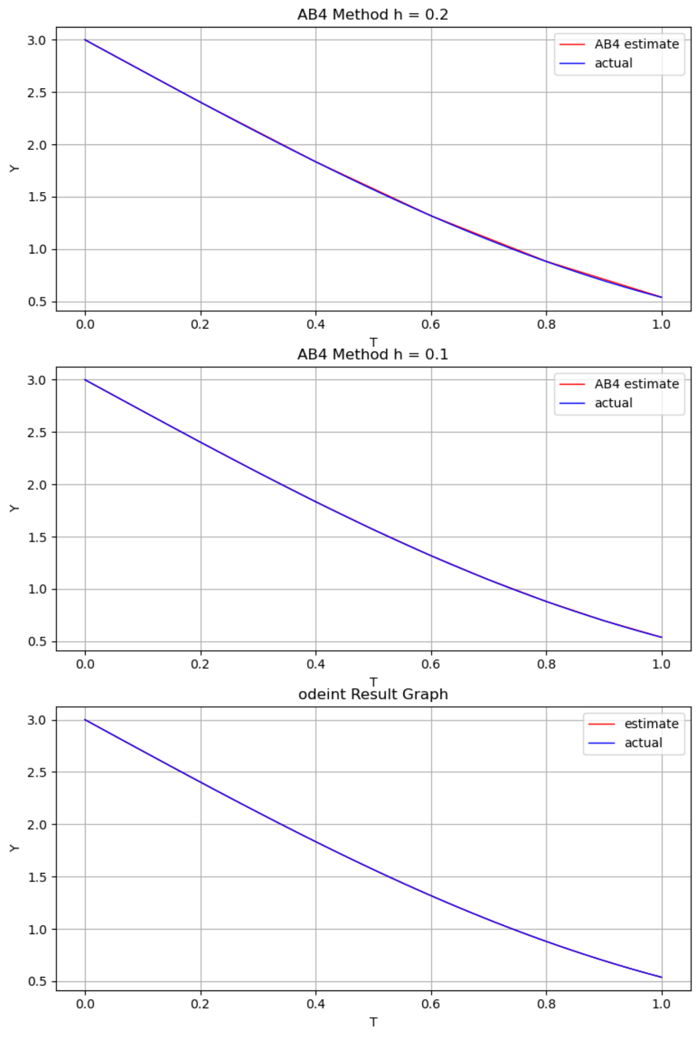


Figure 18: AB4 Solutions to ODE 3

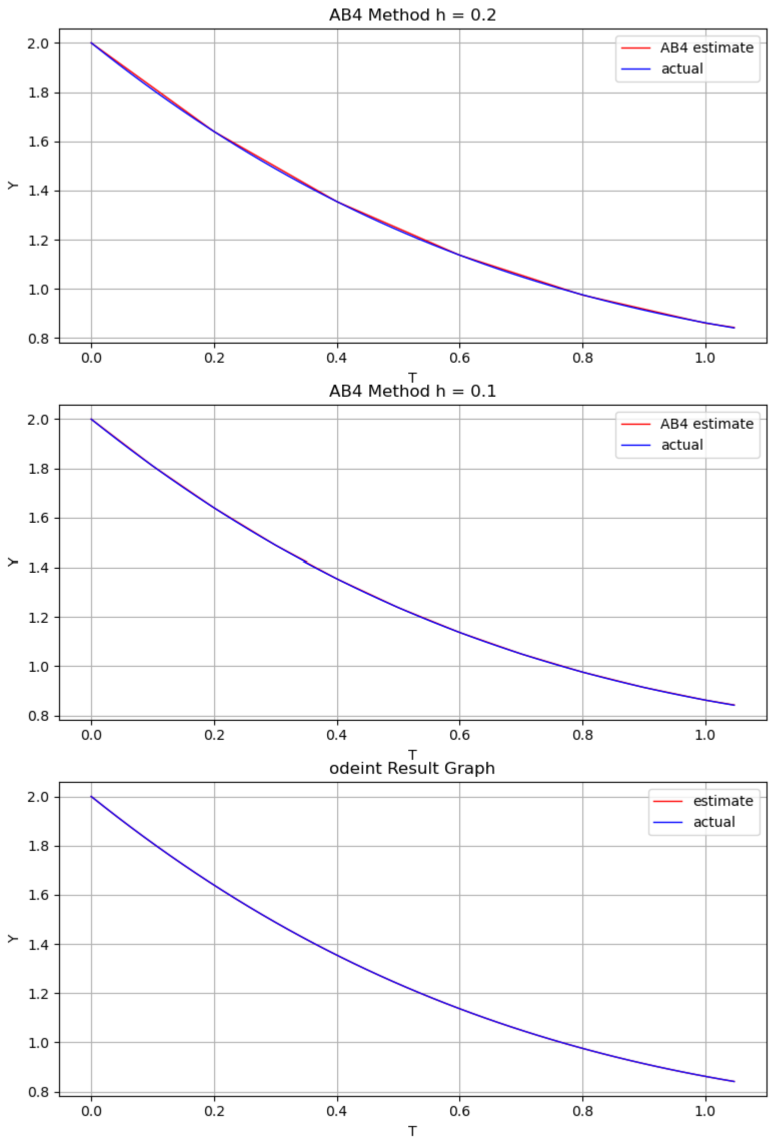


Figure 19: AB4 Solutions to ODE 4

Figures 20 – 23 contain zoomed-in images of graphs with solutions to the four ODEs estimated by both RK4 and AB4. The first three ODE graphs also show the RK2 estimates for the midpoint method and modified Euler's method. The solutions calculated by the methods were so similar, that on the graphs of the entire solutions, it was not possible to determine which method was most accurate. Perhaps the most surprising result shown in the graphs for the first and third ODEs is that the RK2 midpoint method outperformed both AB4 and RK4. Additionally, the RK2 modified Euler's method performed better than RK4 for the first ODE but performed the worst for the second and third. Another interesting thing to note is that for first and fourth ODEs, AB4 was more accurate than RK4, however, for the second and third ODEs, RK4 was more accurate than AB4. I am not sure why this is the case; all of the graphs resemble sections of exponential functions, and AB4 does better on the solution to the fourth ODE which is decreasing, and the solution to the first ODE, which is increasing.

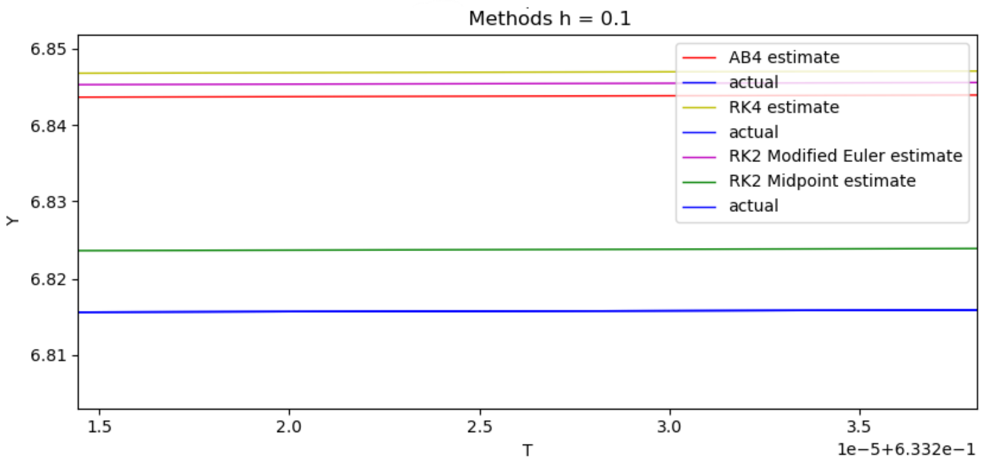


Figure 20: Comparison of AB4, RK4, and RK2 for ODE 1

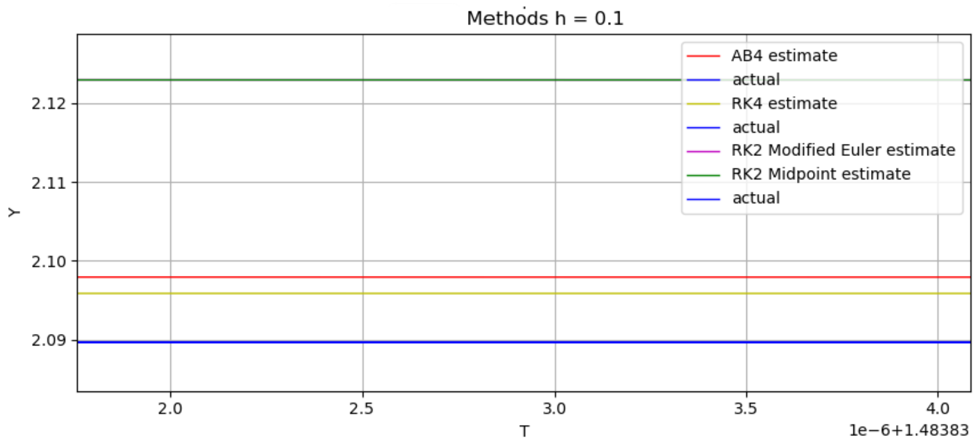


Figure 21: Comparison of AB4, RK4, and RK2 for ODE 2

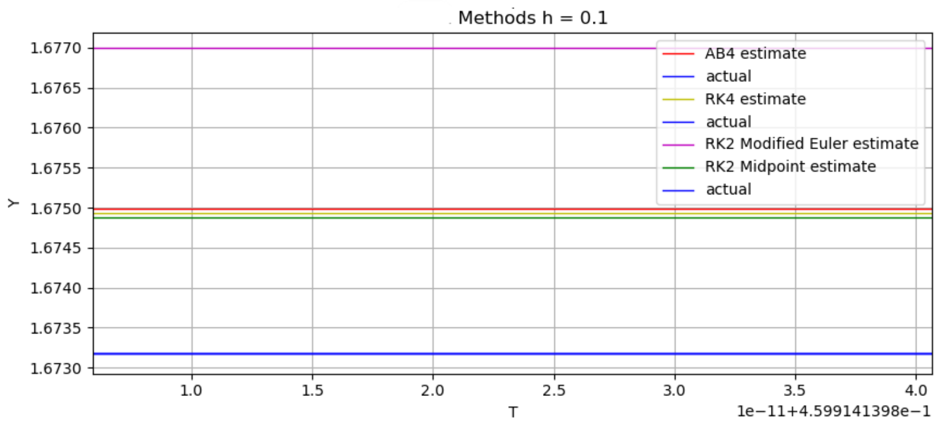


Figure 22: Comparison of AB4, RK4, and RK2 for ODE 3

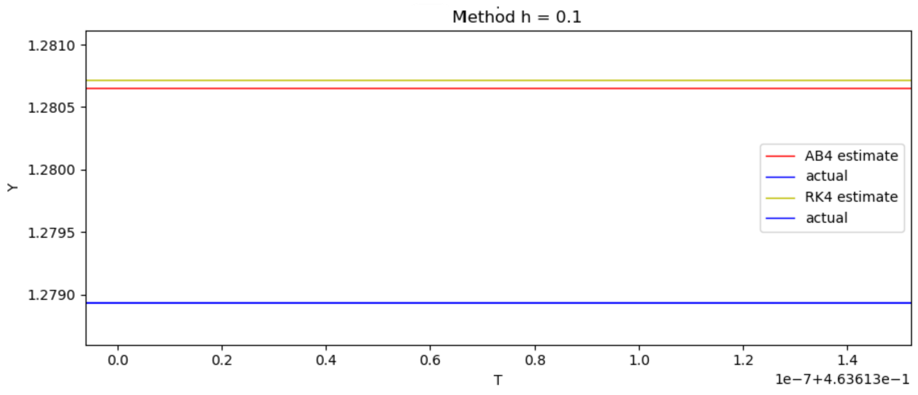
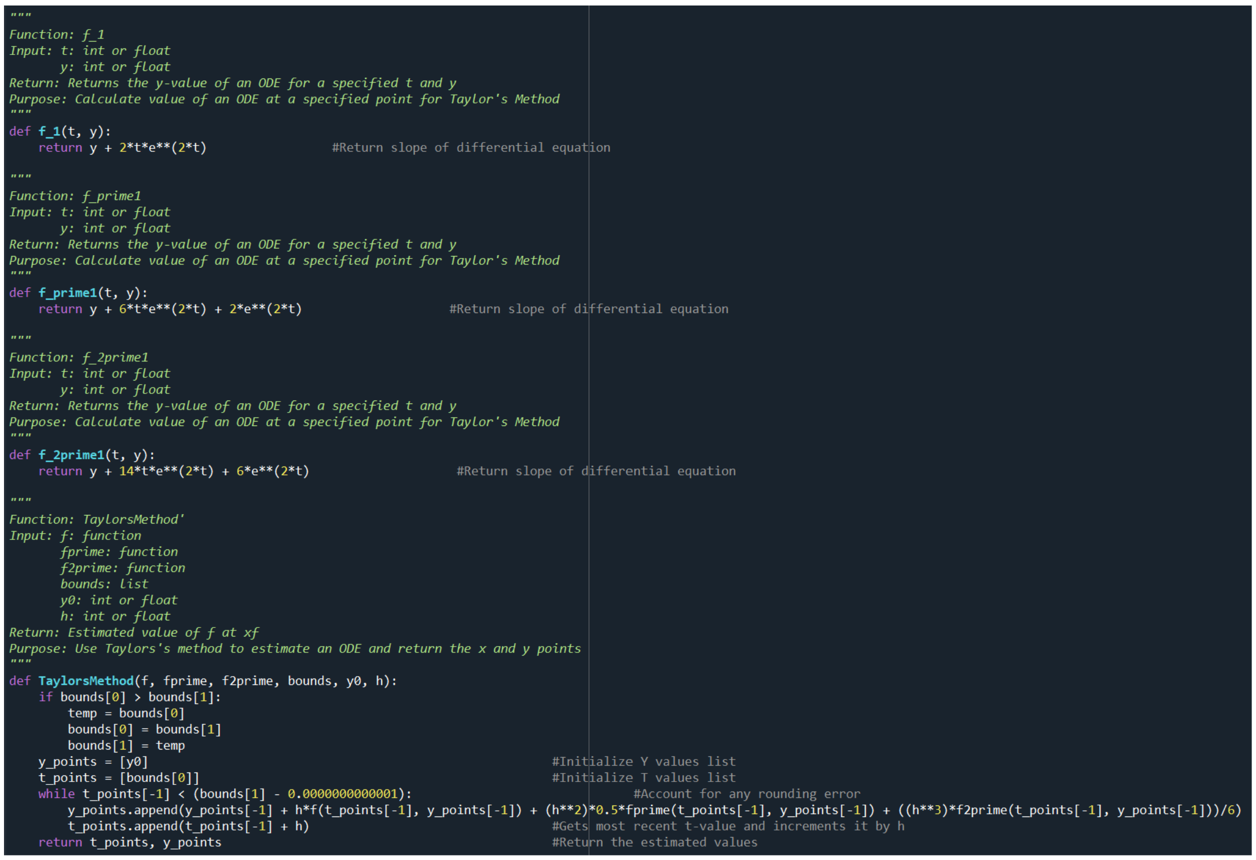


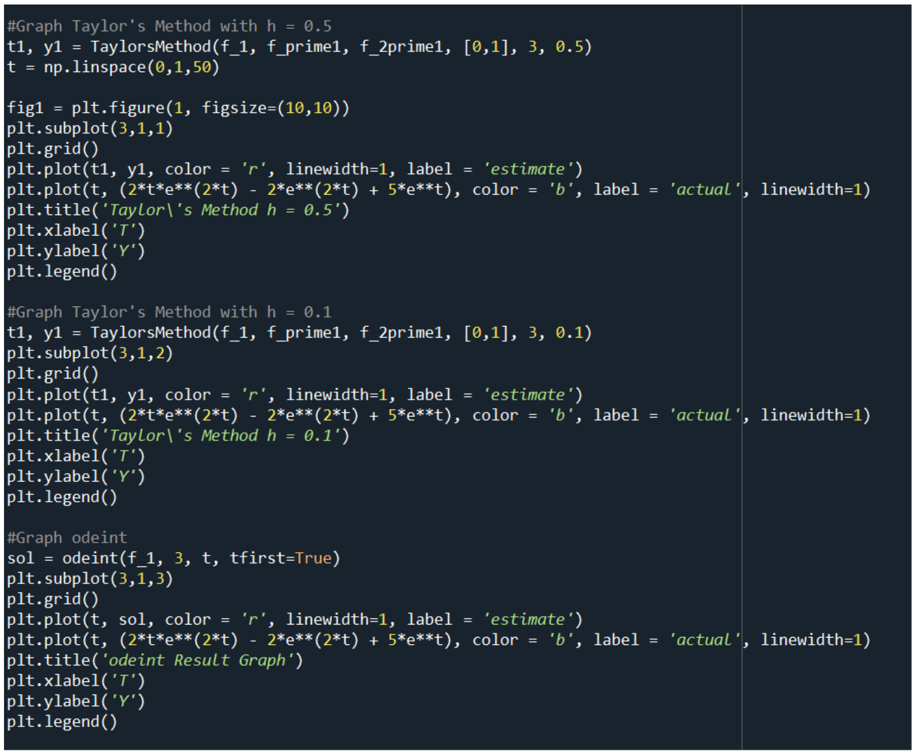
Figure 23: Comparison of AB4 and RK4 for ODE 4

Appendix A



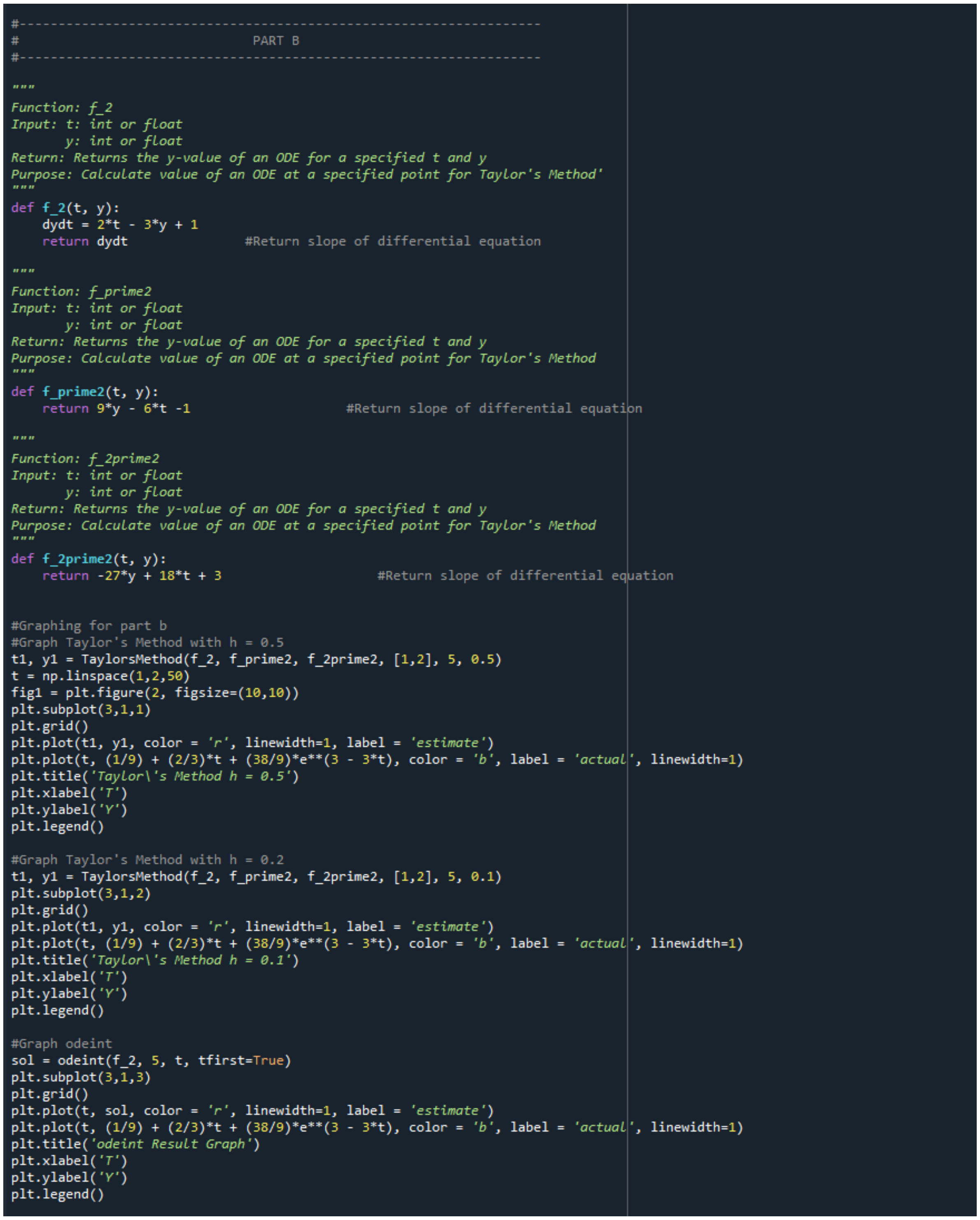
```
"""
Function: f_1
Input: t: int or float
       y: int or float
Return: Returns the y-value of an ODE for a specified t and y
Purpose: Calculate value of an ODE at a specified point for Taylor's Method
"""
def f_1(t, y):
    return y + 2*t*e**(2*t)                 #Return slope of differential equation

"""
Function: f_prime1
Input: t: int or float
       y: int or float
Return: Returns the y-value of an ODE for a specified t and y
Purpose: Calculate value of an ODE at a specified point for Taylor's Method
"""
def f_prime1(t, y):
    return y + 6*t*e**(2*t) + 2*e**(2*t)                #Return slope of differential equation

"""
Function: f_2prime1
Input: t: int or float
       y: int or float
Return: Returns the y-value of an ODE for a specified t and y
Purpose: Calculate value of an ODE at a specified point for Taylor's Method
"""
def f_2prime1(t, y):
    return y + 14*t*e**(2*t) + 6*e**(2*t)               #Return slope of differential equation

"""
Function: TaylorsMethod'
Input: f: function
       fprime: function
       f2prime: function
       bounds: list
       y0: int or float
       h: int or float
Return: Estimated value of f at xf
Purpose: Use Taylors's method to estimate an ODE and return the x and y points
"""
def TaylorsMethod(f, fprime, f2prime, bounds, y0, h):
    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
        y_points.append(y_points[-1] + h*f(t_points[-1], y_points[-1]) + (h**2)*0.5*fprime(t_points[-1], y_points[-1]) + ((h**3)*f2prime(t_points[-1], y_points[-1]))/6)
        t_points.append(t_points[-1] + h)               #Gets most recent t-value and increments it by h
    return t_points, y_points                           #Return the estimated values
```

Figure 24: Taylor's Method Code



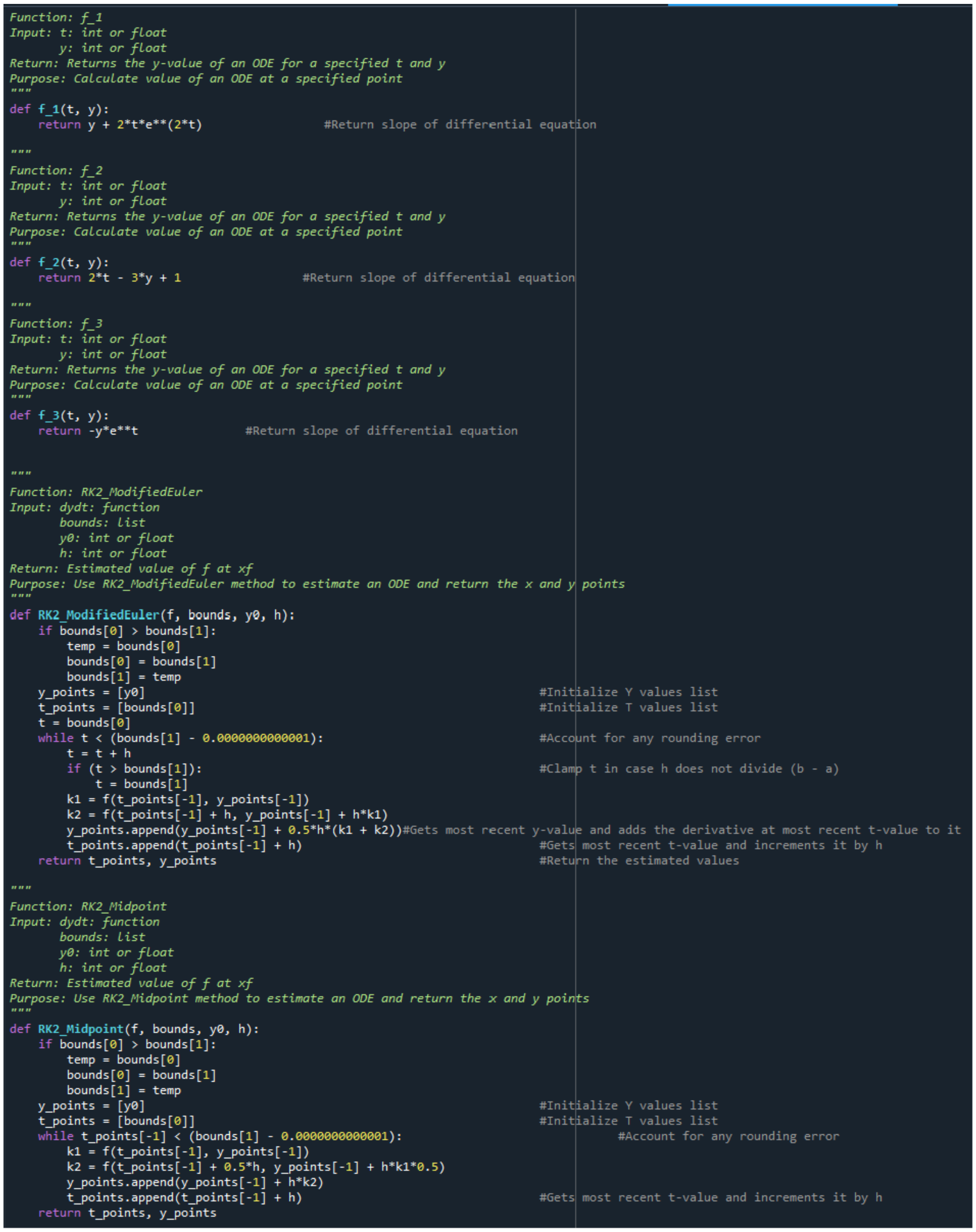
```
#Graph Taylor's Method with h = 0.5
t1, y1 = TaylorsMethod(f_1, f_prime1, f_2prime1, [0,1], 3, 0.5)
t = np.linspace(0,1,50)

fig1 = plt.figure(1, figsize=(10,10))
plt.subplot(3,1,1)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, (2*t*e**(2*t) - 2*e**(2*t) + 5*e**t), color = 'b', label = 'actual', linewidth=1)
plt.title('Taylor\'s Method h = 0.5')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph Taylor's Method with h = 0.1
t1, y1 = TaylorsMethod(f_1, f_prime1, f_2prime1, [0,1], 3, 0.1)
plt.subplot(3,1,2)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, (2*t*e**(2*t) - 2*e**(2*t) + 5*e**t), color = 'b', label = 'actual', linewidth=1)
plt.title('Taylor\'s Method h = 0.1')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f_1, 3, t, tfirst=True)
plt.subplot(3,1,3)
plt.grid()
plt.plot(t, sol, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, (2*t*e**(2*t) - 2*e**(2*t) + 5*e**t), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 25: Part 1 Graph Code for ODE 1



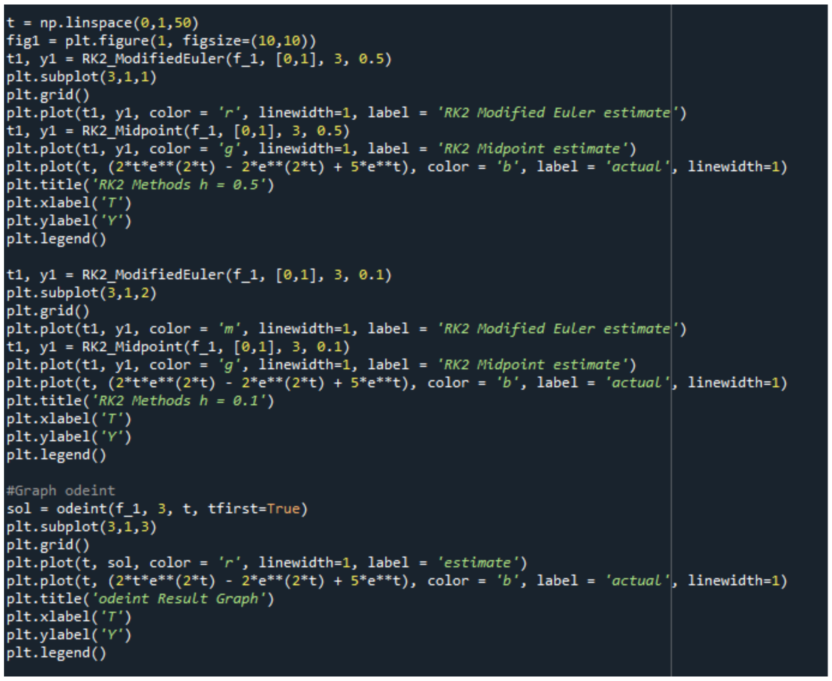
```
#-------------------------------------------------------------------
#                              PART B
#-------------------------------------------------------------------

"""
Function: f_2
Input: t: int or float
       y: int or float
Return: Returns the y-value of an ODE for a specified t and y
Purpose: Calculate value of an ODE at a specified point for Taylor's Method'
"""
def f_2(t, y):
    dydt = 2*t - 3*y + 1
    return dydt               #Return slope of differential equation

"""
Function: f_prime2
Input: t: int or float
       y: int or float
Return: Returns the y-value of an ODE for a specified t and y
Purpose: Calculate value of an ODE at a specified point for Taylor's Method
"""
def f_prime2(t, y):
    return 9*y - 6*t -1                   #Return slope of differential equation

"""
Function: f_2prime2
Input: t: int or float
       y: int or float
Return: Returns the y-value of an ODE for a specified t and y
Purpose: Calculate value of an ODE at a specified point for Taylor's Method
"""
def f_2prime2(t, y):
    return -27*y + 18*t + 3                   #Return slope of differential equation


#Graphing for part b
#Graph Taylor's Method with h = 0.5
t1, y1 = TaylorsMethod(f_2, f_prime2, f_2prime2, [1,2], 5, 0.5)
t = np.linspace(1,2,50)
fig1 = plt.figure(2, figsize=(10,10))
plt.subplot(3,1,1)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, (1/9) + (2/3)*t + (38/9)*e**(3 - 3*t), color = 'b', label = 'actual', linewidth=1)
plt.title('Taylor\'s Method h = 0.5')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph Taylor's Method with h = 0.2
t1, y1 = TaylorsMethod(f_2, f_prime2, f_2prime2, [1,2], 5, 0.1)
plt.subplot(3,1,2)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, (1/9) + (2/3)*t + (38/9)*e**(3 - 3*t), color = 'b', label = 'actual', linewidth=1)
plt.title('Taylor\'s Method h = 0.1')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f_2, 5, t, tfirst=True)
plt.subplot(3,1,3)
plt.grid()
plt.plot(t, sol, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, (1/9) + (2/3)*t + (38/9)*e**(3 - 3*t), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 26: Part 1 Graph Code for ODE 2



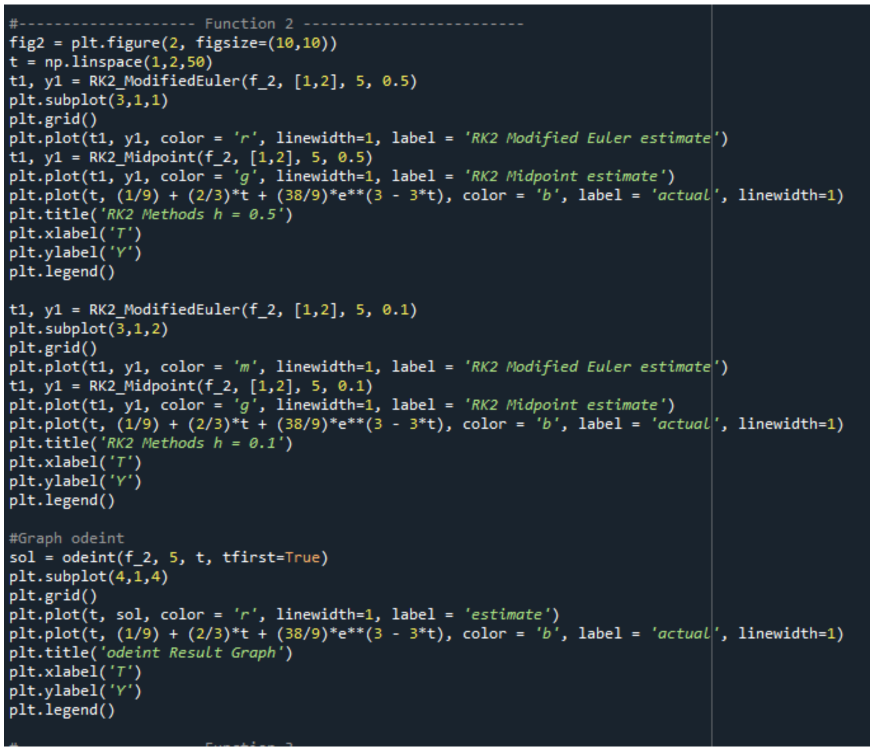
```python
Function: f_1
Input: t: int or float
       y: int or float
Return: Returns the y-value of an ODE for a specified t and y
Purpose: Calculate value of an ODE at a specified point
"""
def f_1(t, y):
    return y + 2*t*e**(2*t)                 #Return slope of differential equation

"""
Function: f_2
Input: t: int or float
       y: int or float
Return: Returns the y-value of an ODE for a specified t and y
Purpose: Calculate value of an ODE at a specified point
"""
def f_2(t, y):
    return 2*t - 3*y + 1                 #Return slope of differential equation

"""
Function: f_3
Input: t: int or float
       y: int or float
Return: Returns the y-value of an ODE for a specified t and y
Purpose: Calculate value of an ODE at a specified point
"""
def f_3(t, y):
    return -y*e**t               #Return slope of differential equation


"""
Function: RK2_ModifiedEuler
Input: dydt: function
       bounds: list
       y0: int or float
       h: int or float
Return: Estimated value of f at xf
Purpose: Use RK2_ModifiedEuler method to estimate an ODE and return the x and y points
"""
def RK2_ModifiedEuler(f, bounds, y0, h):
    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
    t = bounds[0]
    while t < (bounds[1] - 0.0000000000001):                     #Account for any rounding error
        t = t + h
        if (t > bounds[1]):                                      #Clamp t in case h does not divide (b - a)
            t = bounds[1]
        k1 = f(t_points[-1], y_points[-1])
        k2 = f(t_points[-1] + h, y_points[-1] + h*k1)
        y_points.append(y_points[-1] + 0.5*h*(k1 + k2))#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
    return t_points, y_points                                    #Return the estimated values

"""
Function: RK2_Midpoint
Input: dydt: function
       bounds: list
       y0: int or float
       h: int or float
Return: Estimated value of f at xf
Purpose: Use RK2_Midpoint method to estimate an ODE and return the x and y points
"""
def RK2_Midpoint(f, bounds, y0, h):
    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
        k1 = f(t_points[-1], y_points[-1])
        k2 = f(t_points[-1] + 0.5*h, y_points[-1] + h*k1*0.5)
        y_points.append(y_points[-1] + h*k2)
        t_points.append(t_points[-1] + h)                        #Gets most recent t-value and increments it by h
    return t_points, y_points
```

Figure 27: Code for RK2 Methods



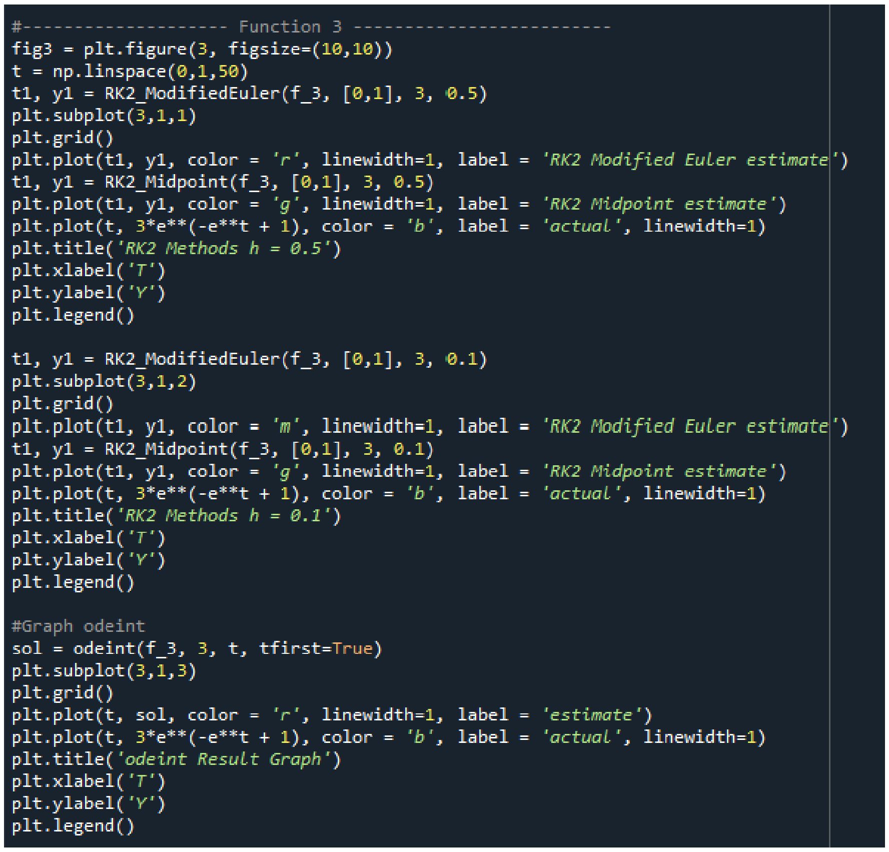
```python
t = np.linspace(0,1,50)
fig1 = plt.figure(1, figsize=(10,10))
t1, y1 = RK2_ModifiedEuler(f_1, [0,1], 3, 0.5)
plt.subplot(3,1,1)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'RK2 Modified Euler estimate')
t1, y1 = RK2_Midpoint(f_1, [0,1], 3, 0.5)
plt.plot(t1, y1, color = 'g', linewidth=1, label = 'RK2 Midpoint estimate')
plt.plot(t, (2*t*e**(2*t) - 2*e**(2*t) + 5*e**t), color = 'b', label = 'actual', linewidth=1)
plt.title('RK2 Methods h = 0.5')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

t1, y1 = RK2_ModifiedEuler(f_1, [0,1], 3, 0.1)
plt.subplot(3,1,2)
plt.grid()
plt.plot(t1, y1, color = 'm', linewidth=1, label = 'RK2 Modified Euler estimate')
t1, y1 = RK2_Midpoint(f_1, [0,1], 3, 0.1)
plt.plot(t1, y1, color = 'g', linewidth=1, label = 'RK2 Midpoint estimate')
plt.plot(t, (2*t*e**(2*t) - 2*e**(2*t) + 5*e**t), color = 'b', label = 'actual', linewidth=1)
plt.title('RK2 Methods h = 0.1')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f_1, 3, t, tfirst=True)
plt.subplot(3,1,3)
plt.grid()
plt.plot(t, sol, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, (2*t*e**(2*t) - 2*e**(2*t) + 5*e**t), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 28: Part 2 Graph Code for ODE 1



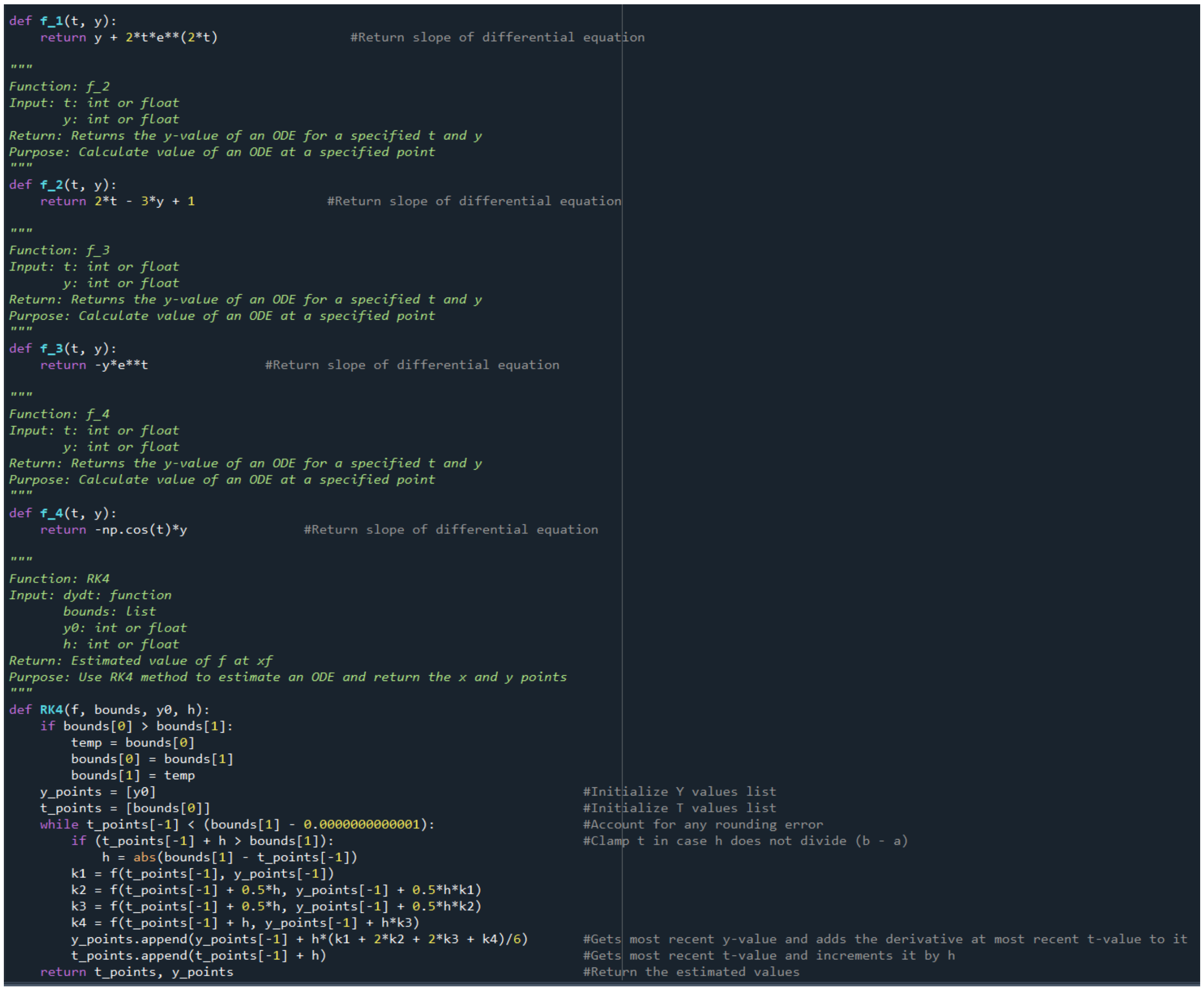
```python
#------------------- Function 2 ------------------------
fig2 = plt.figure(2, figsize=(10,10))
t = np.linspace(1,2,50)
t1, y1 = RK2_ModifiedEuler(f_2, [1,2], 5, 0.5)
plt.subplot(3,1,1)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'RK2 Modified Euler estimate')
t1, y1 = RK2_Midpoint(f_2, [1,2], 5, 0.5)
plt.plot(t1, y1, color = 'g', linewidth=1, label = 'RK2 Midpoint estimate')
plt.plot(t, (1/9) + (2/3)*t + (38/9)*e**(3 - 3*t), color = 'b', label = 'actual', linewidth=1)
plt.title('RK2 Methods h = 0.5')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

t1, y1 = RK2_ModifiedEuler(f_2, [1,2], 5, 0.1)
plt.subplot(3,1,2)
plt.grid()
plt.plot(t1, y1, color = 'm', linewidth=1, label = 'RK2 Modified Euler estimate')
t1, y1 = RK2_Midpoint(f_2, [1,2], 5, 0.1)
plt.plot(t1, y1, color = 'g', linewidth=1, label = 'RK2 Midpoint estimate')
plt.plot(t, (1/9) + (2/3)*t + (38/9)*e**(3 - 3*t), color = 'b', label = 'actual', linewidth=1)
plt.title('RK2 Methods h = 0.1')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f_2, 5, t, tfirst=True)
plt.subplot(4,1,4)
plt.grid()
plt.plot(t, sol, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, (1/9) + (2/3)*t + (38/9)*e**(3 - 3*t), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 29: Part 2 Graph Code for ODE 2



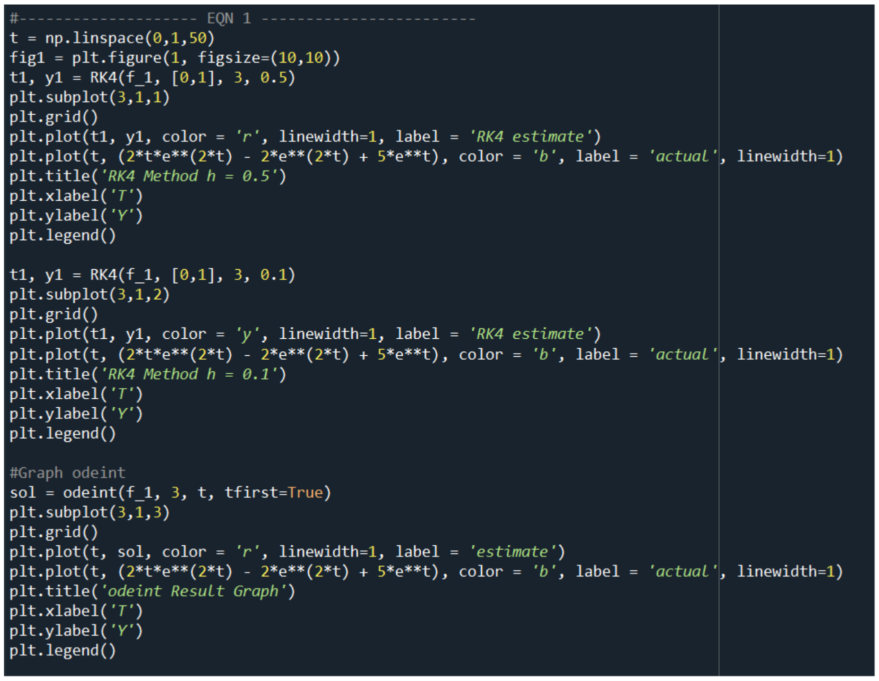
```
#-------------------- Function 3 -------------------------
fig3 = plt.figure(3, figsize=(10,10))
t = np.linspace(0,1,50)
t1, y1 = RK2_ModifiedEuler(f_3, [0,1], 3, 0.5)
plt.subplot(3,1,1)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'RK2 Modified Euler estimate')
t1, y1 = RK2_Midpoint(f_3, [0,1], 3, 0.5)
plt.plot(t1, y1, color = 'g', linewidth=1, label = 'RK2 Midpoint estimate')
plt.plot(t, 3*e**(-e**t + 1), color = 'b', label = 'actual', linewidth=1)
plt.title('RK2 Methods h = 0.5')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

t1, y1 = RK2_ModifiedEuler(f_3, [0,1], 3, 0.1)
plt.subplot(3,1,2)
plt.grid()
plt.plot(t1, y1, color = 'm', linewidth=1, label = 'RK2 Modified Euler estimate')
t1, y1 = RK2_Midpoint(f_3, [0,1], 3, 0.1)
plt.plot(t1, y1, color = 'g', linewidth=1, label = 'RK2 Midpoint estimate')
plt.plot(t, 3*e**(-e**t + 1), color = 'b', label = 'actual', linewidth=1)
plt.title('RK2 Methods h = 0.1')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f_3, 3, t, tfirst=True)
plt.subplot(3,1,3)
plt.grid()
plt.plot(t, sol, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, 3*e**(-e**t + 1), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 30: Part 2 Graph Code for ODE 3



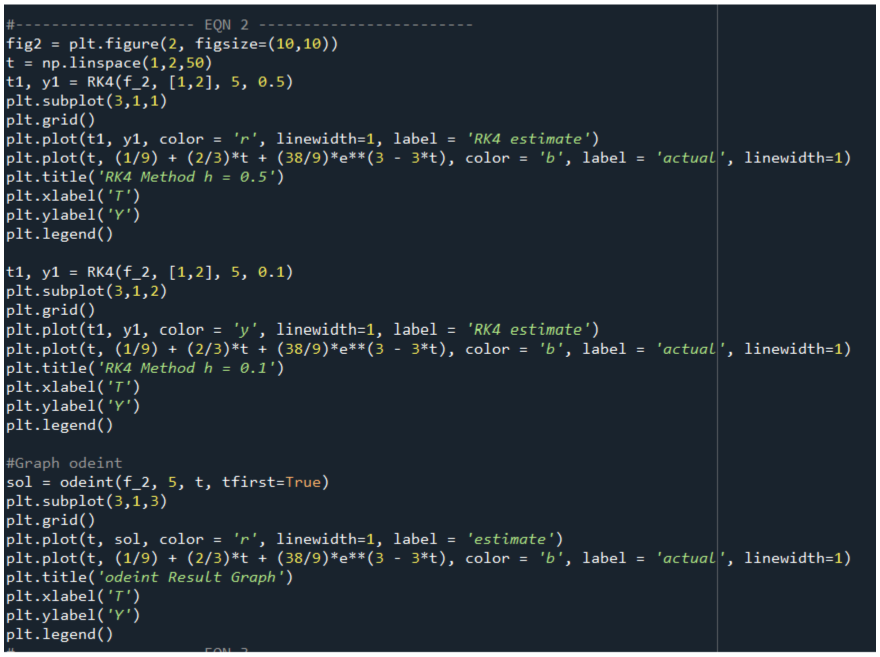
```python
def f_1(t, y):
    return y + 2*t*e**(2*t)                 #Return slope of differential equation

"""
Function: f_2
Input: t: int or float
       y: int or float
Return: Returns the y-value of an ODE for a specified t and y
Purpose: Calculate value of an ODE at a specified point
"""
def f_2(t, y):
    return 2*t - 3*y + 1                 #Return slope of differential equation

"""
Function: f_3
Input: t: int or float
       y: int or float
Return: Returns the y-value of an ODE for a specified t and y
Purpose: Calculate value of an ODE at a specified point
"""
def f_3(t, y):
    return -y*e**t               #Return slope of differential equation

"""
Function: f_4
Input: t: int or float
       y: int or float
Return: Returns the y-value of an ODE for a specified t and y
Purpose: Calculate value of an ODE at a specified point
"""
def f_4(t, y):
    return -np.cos(t)*y               #Return slope of differential equation

"""
Function: RK4
Input: dydt: function
       bounds: list
       y0: int or float
       h: int or float
Return: Estimated value of f at xf
Purpose: Use RK4 method to estimate an ODE and return the x and y points
"""
def RK4(f, bounds, y0, h):
    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
    return t_points, y_points                                            #Return the estimated values
```

Figure 31: Code for RK4 Method



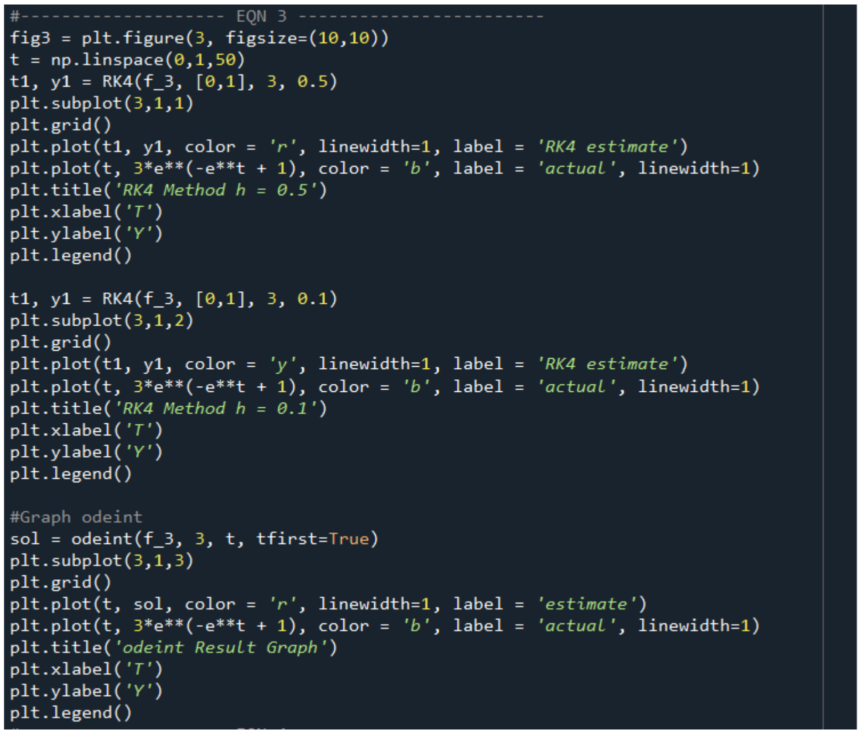
```python
#-------------------- EQN 1 ------------------------
t = np.linspace(0,1,50)
fig1 = plt.figure(1, figsize=(10,10))
t1, y1 = RK4(f_1, [0,1], 3, 0.5)
plt.subplot(3,1,1)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'RK4 estimate')
plt.plot(t, (2*t*e**(2*t) - 2*e**(2*t) + 5*e**t), color = 'b', label = 'actual', linewidth=1)
plt.title('RK4 Method h = 0.5')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

t1, y1 = RK4(f_1, [0,1], 3, 0.1)
plt.subplot(3,1,2)
plt.grid()
plt.plot(t1, y1, color = 'y', linewidth=1, label = 'RK4 estimate')
plt.plot(t, (2*t*e**(2*t) - 2*e**(2*t) + 5*e**t), color = 'b', label = 'actual', linewidth=1)
plt.title('RK4 Method h = 0.1')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f_1, 3, t, tfirst=True)
plt.subplot(3,1,3)
plt.grid()
plt.plot(t, sol, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, (2*t*e**(2*t) - 2*e**(2*t) + 5*e**t), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 32: Part 3 Graph Code for ODE 1



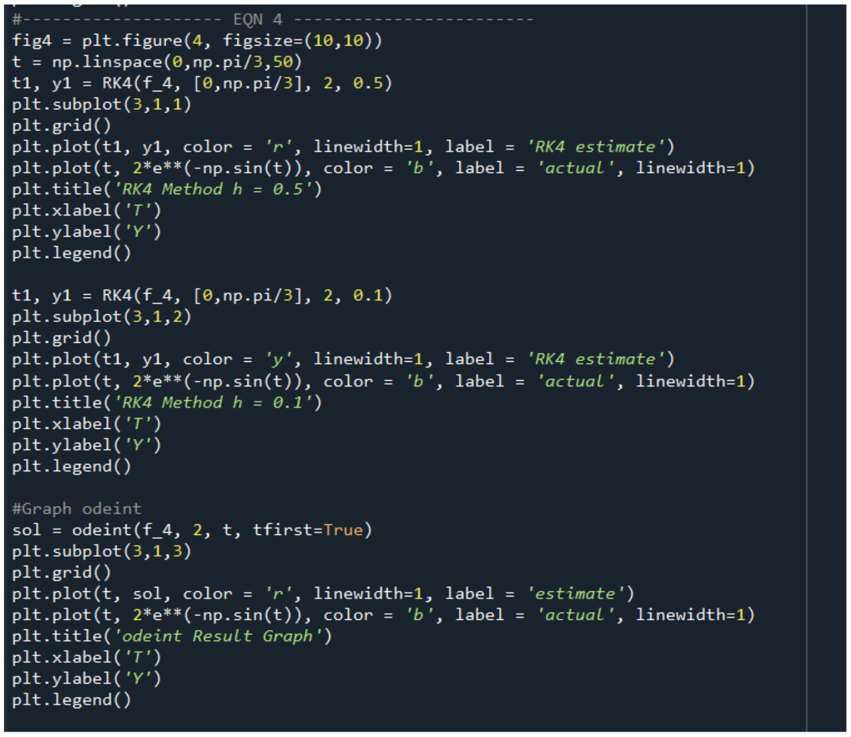
```python
#-------------------- EQN 2 ------------------------
fig2 = plt.figure(2, figsize=(10,10))
t = np.linspace(1,2,50)
t1, y1 = RK4(f_2, [1,2], 5, 0.5)
plt.subplot(3,1,1)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'RK4 estimate')
plt.plot(t, (1/9) + (2/3)*t + (38/9)*e**(3 - 3*t), color = 'b', label = 'actual', linewidth=1)
plt.title('RK4 Method h = 0.5')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

t1, y1 = RK4(f_2, [1,2], 5, 0.1)
plt.subplot(3,1,2)
plt.grid()
plt.plot(t1, y1, color = 'y', linewidth=1, label = 'RK4 estimate')
plt.plot(t, (1/9) + (2/3)*t + (38/9)*e**(3 - 3*t), color = 'b', label = 'actual', linewidth=1)
plt.title('RK4 Method h = 0.1')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f_2, 5, t, tfirst=True)
plt.subplot(3,1,3)
plt.grid()
plt.plot(t, sol, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, (1/9) + (2/3)*t + (38/9)*e**(3 - 3*t), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 33: Part 3 Graph Code for ODE 2



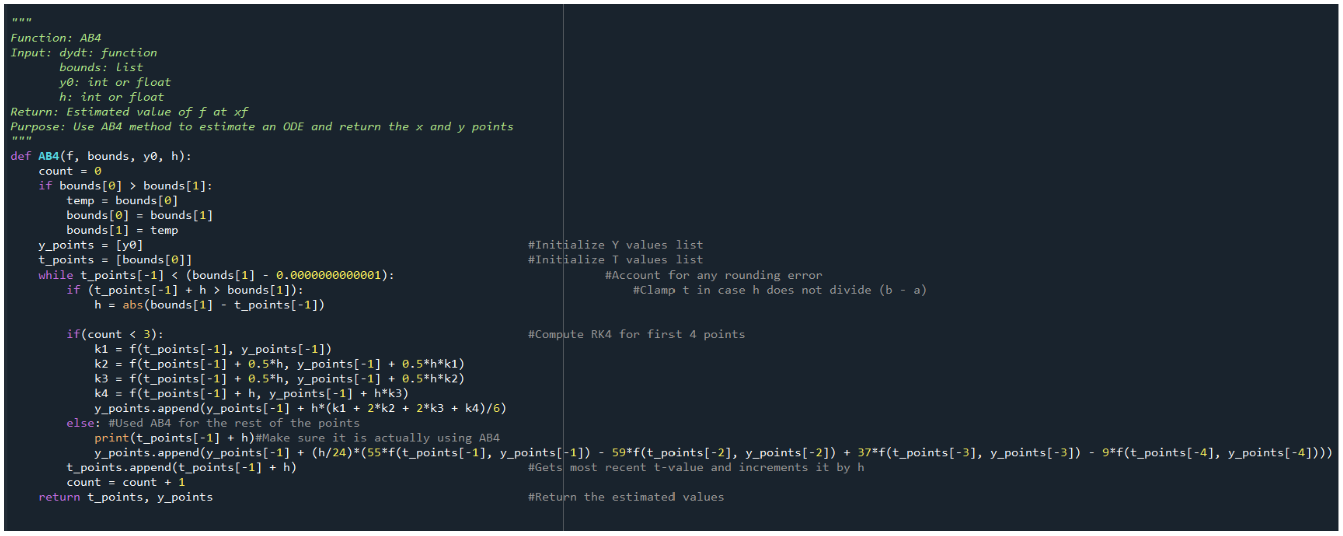
```python
#-------------------- EQN 3 -----------------------
fig3 = plt.figure(3, figsize=(10,10))
t = np.linspace(0,1,50)
t1, y1 = RK4(f_3, [0,1], 3, 0.5)
plt.subplot(3,1,1)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'RK4 estimate')
plt.plot(t, 3*e**(-e**t + 1), color = 'b', label = 'actual', linewidth=1)
plt.title('RK4 Method h = 0.5')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

t1, y1 = RK4(f_3, [0,1], 3, 0.1)
plt.subplot(3,1,2)
plt.grid()
plt.plot(t1, y1, color = 'y', linewidth=1, label = 'RK4 estimate')
plt.plot(t, 3*e**(-e**t + 1), color = 'b', label = 'actual', linewidth=1)
plt.title('RK4 Method h = 0.1')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f_3, 3, t, tfirst=True)
plt.subplot(3,1,3)
plt.grid()
plt.plot(t, sol, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, 3*e**(-e**t + 1), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 34: Part 3 Graph Code for ODE 3



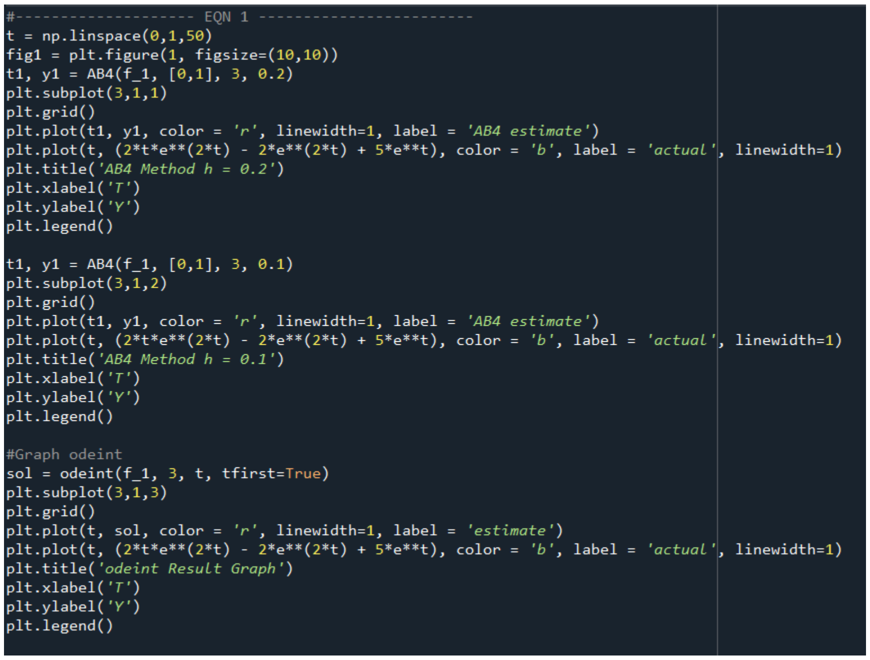
```python
#------------------- EQN 4 -----------------------
fig4 = plt.figure(4, figsize=(10,10))
t = np.linspace(0,np.pi/3,50)
t1, y1 = RK4(f_4, [0,np.pi/3], 2, 0.5)
plt.subplot(3,1,1)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'RK4 estimate')
plt.plot(t, 2*e**(-np.sin(t)), color = 'b', label = 'actual', linewidth=1)
plt.title('RK4 Method h = 0.5')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

t1, y1 = RK4(f_4, [0,np.pi/3], 2, 0.1)
plt.subplot(3,1,2)
plt.grid()
plt.plot(t1, y1, color = 'y', linewidth=1, label = 'RK4 estimate')
plt.plot(t, 2*e**(-np.sin(t)), color = 'b', label = 'actual', linewidth=1)
plt.title('RK4 Method h = 0.1')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f_4, 2, t, tfirst=True)
plt.subplot(3,1,3)
plt.grid()
plt.plot(t, sol, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, 2*e**(-np.sin(t)), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 35: Part 3 Graph Code for ODE 4



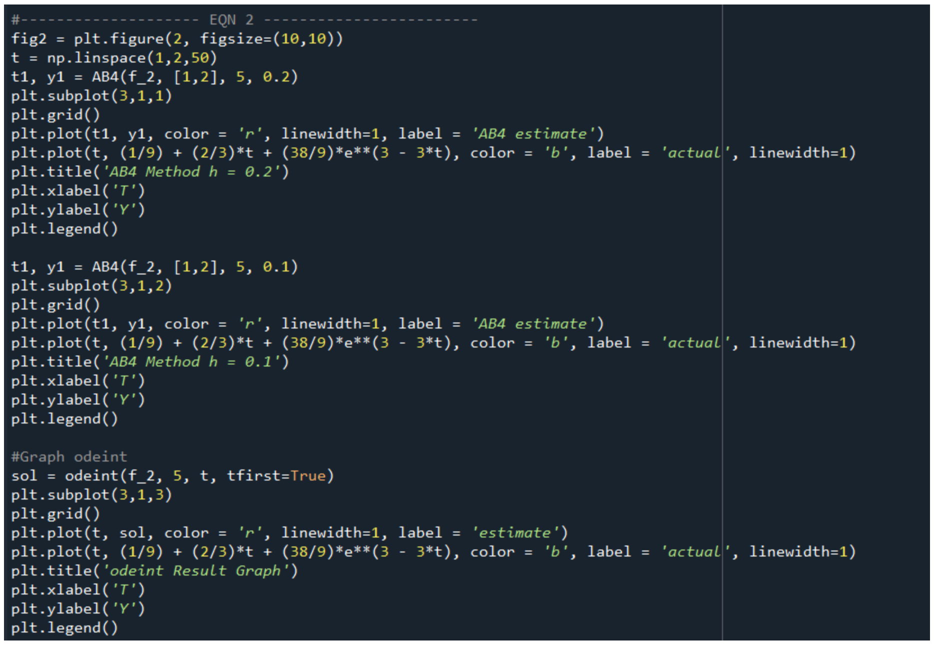
```python
"""
Function: AB4
Input: dydt: function
       bounds: list
       y0: int or float
       h: int or float
Return: Estimated value of f at xf
Purpose: Use AB4 method to estimate an ODE and return the x and y points
"""
def AB4(f, bounds, y0, h):
    count = 0
    if bounds[0] > bounds[1]:
        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])

        if(count < 3):                                                    #Compute RK4 for first 4 points
            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)
        else: #Used AB4 for the rest of the points
            print(t_points[-1] + h)#Make sure it is actually using AB4
            y_points.append(y_points[-1] + (h/24)*(55*f(t_points[-1], y_points[-1]) - 59*f(t_points[-2], y_points[-2]) + 37*f(t_points[-3], y_points[-3]) - 9*f(t_points[-4], y_points[-4])))
        t_points.append(t_points[-1] + h)                                 #Gets most recent t-value and increments it by h
        count = count + 1
    return t_points, y_points                                             #Return the estimated values
```

Figure 36: Code for AB4 Method



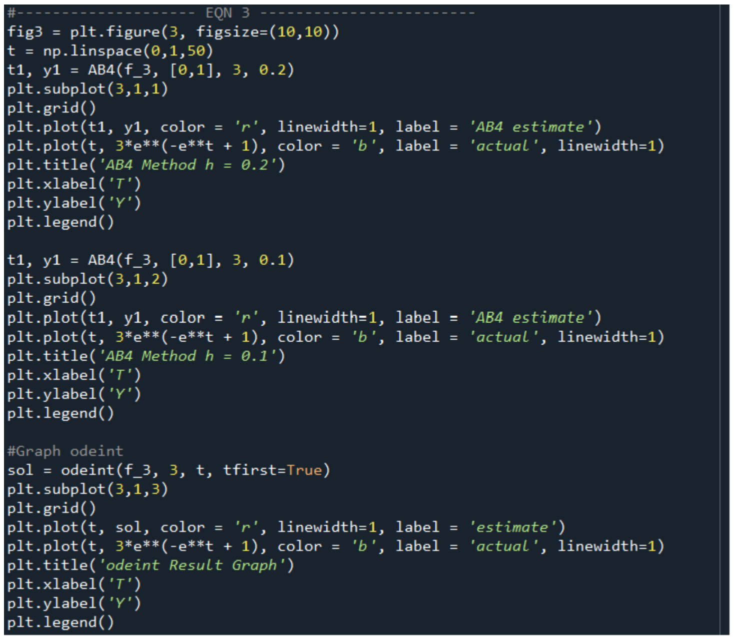
```python
#-------------------- EQN 1 ------------------------
t = np.linspace(0,1,50)
fig1 = plt.figure(1, figsize=(10,10))
t1, y1 = AB4(f_1, [0,1], 3, 0.2)
plt.subplot(3,1,1)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'AB4 estimate')
plt.plot(t, (2*t*e**(2*t) - 2*e**(2*t) + 5*e**t), color = 'b', label = 'actual', linewidth=1)
plt.title('AB4 Method h = 0.2')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

t1, y1 = AB4(f_1, [0,1], 3, 0.1)
plt.subplot(3,1,2)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'AB4 estimate')
plt.plot(t, (2*t*e**(2*t) - 2*e**(2*t) + 5*e**t), color = 'b', label = 'actual', linewidth=1)
plt.title('AB4 Method h = 0.1')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f_1, 3, t, tfirst=True)
plt.subplot(3,1,3)
plt.grid()
plt.plot(t, sol, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, (2*t*e**(2*t) - 2*e**(2*t) + 5*e**t), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 37: Part 4 Graph Code for ODE 1



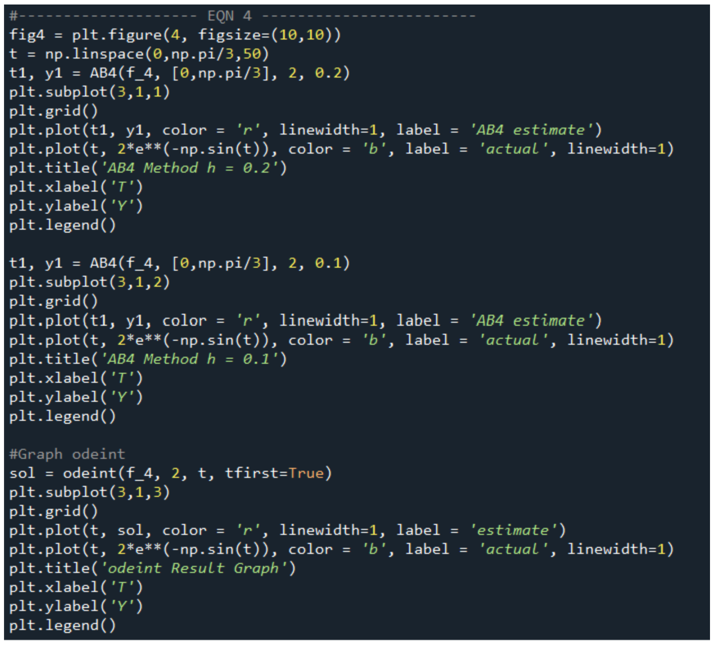
```
#-------------------- EQN 2 ------------------------
fig2 = plt.figure(2, figsize=(10,10))
t = np.linspace(1,2,50)
t1, y1 = AB4(f_2, [1,2], 5, 0.2)
plt.subplot(3,1,1)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'AB4 estimate')
plt.plot(t, (1/9) + (2/3)*t + (38/9)*e**(3 - 3*t), color = 'b', label = 'actual', linewidth=1)
plt.title('AB4 Method h = 0.2')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

t1, y1 = AB4(f_2, [1,2], 5, 0.1)
plt.subplot(3,1,2)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'AB4 estimate')
plt.plot(t, (1/9) + (2/3)*t + (38/9)*e**(3 - 3*t), color = 'b', label = 'actual', linewidth=1)
plt.title('AB4 Method h = 0.1')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f_2, 5, t, tfirst=True)
plt.subplot(3,1,3)
plt.grid()
plt.plot(t, sol, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, (1/9) + (2/3)*t + (38/9)*e**(3 - 3*t), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 38: Part 4 Graph Code for ODE 2



```
#-------------------- EQN 3 ------------------------
fig3 = plt.figure(3, figsize=(10,10))
t = np.linspace(0,1,50)
t1, y1 = AB4(f_3, [0,1], 3, 0.2)
plt.subplot(3,1,1)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'AB4 estimate')
plt.plot(t, 3*e**(-e**t + 1), color = 'b', label = 'actual', linewidth=1)
plt.title('AB4 Method h = 0.2')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

t1, y1 = AB4(f_3, [0,1], 3, 0.1)
plt.subplot(3,1,2)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'AB4 estimate')
plt.plot(t, 3*e**(-e**t + 1), color = 'b', label = 'actual', linewidth=1)
plt.title('AB4 Method h = 0.1')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f_3, 3, t, tfirst=True)
plt.subplot(3,1,3)
plt.grid()
plt.plot(t, sol, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, 3*e**(-e**t + 1), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 39: Part 4 Graph Code for ODE 3



```python
#-------------------- EQN 4 ------------------------
fig4 = plt.figure(4, figsize=(10,10))
t = np.linspace(0,np.pi/3,50)
t1, y1 = AB4(f_4, [0,np.pi/3], 2, 0.2)
plt.subplot(3,1,1)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'AB4 estimate')
plt.plot(t, 2*e**(-np.sin(t)), color = 'b', label = 'actual', linewidth=1)
plt.title('AB4 Method h = 0.2')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

t1, y1 = AB4(f_4, [0,np.pi/3], 2, 0.1)
plt.subplot(3,1,2)
plt.grid()
plt.plot(t1, y1, color = 'r', linewidth=1, label = 'AB4 estimate')
plt.plot(t, 2*e**(-np.sin(t)), color = 'b', label = 'actual', linewidth=1)
plt.title('AB4 Method h = 0.1')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(f_4, 2, t, tfirst=True)
plt.subplot(3,1,3)
plt.grid()
plt.plot(t, sol, color = 'r', linewidth=1, label = 'estimate')
plt.plot(t, 2*e**(-np.sin(t)), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 40: Part 4 Graph Code for ODE 4