3/21/2024

Project 3 Report

MATH 452

Cameron Robinson

Table of Contents

Table of Figures

Project 4

Part 1

The objective of part one was to numerically approximate the velocity of a hot-air balloon after three seconds by solving the ODE in Figure 1 with any method. The initial velocity of the balloon is zero ft/s, and gravity (g) is 32 ft/s^2. Figure 1 also contains the solution to the ODE.

$$v'(t) = g - 2.2v,\ v(0) = 0\ on\ [0, 3]$$

$$v(t) = \frac{g}{2.2}(1 - e^{-2.2t})$$

$$g = 32\frac{ft}{s^2}$$

Figure 1: ODE and Solution for Part 1

The methods chosen for this problem were the Euler predictor-corrector method (Euler PC), and the RK2 midpoint method. These methods were chosen because I compared Euler PC to RK4 in project 3, and RK4 performed a lot better than Euler PC, so I was curious to see how RK2 would compare. The implementation of Euler PC is shown in Figure 2. Euler PC uses forward Euler's method to predict, and backward Euler's method to correct; no root-finding methods are used. The implementation of RK2 is shown in Figure 3. Euler PC estimated the velocity of the balloon at three seconds to be 14.515 ft/s, while RK2 estimated 14.525 ft/s. Plugging t = 3s into the actual solution, the velocity turns out to be about 14.526 ft/s, so RK2 is slightly more accurate than Euler PC in this case. Code for RK2 and Euler PC is shown in Figure 20.

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

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

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

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

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

Figure 2: Euler PC Method Implementation

$$RK2\ Midpoint\ Method$$

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

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

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

Figure 3: RK2 Midpoint Method Implementation

Figure 4 contains one subplot with estimated solutions to the ODE made by both RK2 and Euler PC. Both methods are used to estimate solutions at three step-sizes, with each step-size decreasing by half of the previous one. The second subplot contains a solution estimated by odeint, a built-in ODE solver. Both subplots contain graphs of the actual solution as well. One interesting result of the predicted solutions is that RK2 with the largest step-size performed about as well as Euler PC with the smallest step-size. This is more apparent on the zoomed in graph in Figure 5, but even in Figure 4 the magenta, red, and green lines (Euler PC solutions) are all clearly visible and are pretty far away from the actual solution. At some points RK2 is more accurate than the most accurate Euler PC, and other times it is less accurate.

Both methods converge as expected. RK2 with an error of $O(h^3)$ converges much faster than Euler PC which has an error of $O(h^2)$. Both methods also become much more accurate as the step-sizes decrease. This is especially apparent on Euler PC from h = 0.2307 to h = 0.1153, where the estimation appears to become more than twice as accurate as the step-size decreases by half. Neither method comes close to being as accurate as odeint, but that is expected since neither method is particularly accurate to begin with. The code to create the graphs is shown in Figure 21.

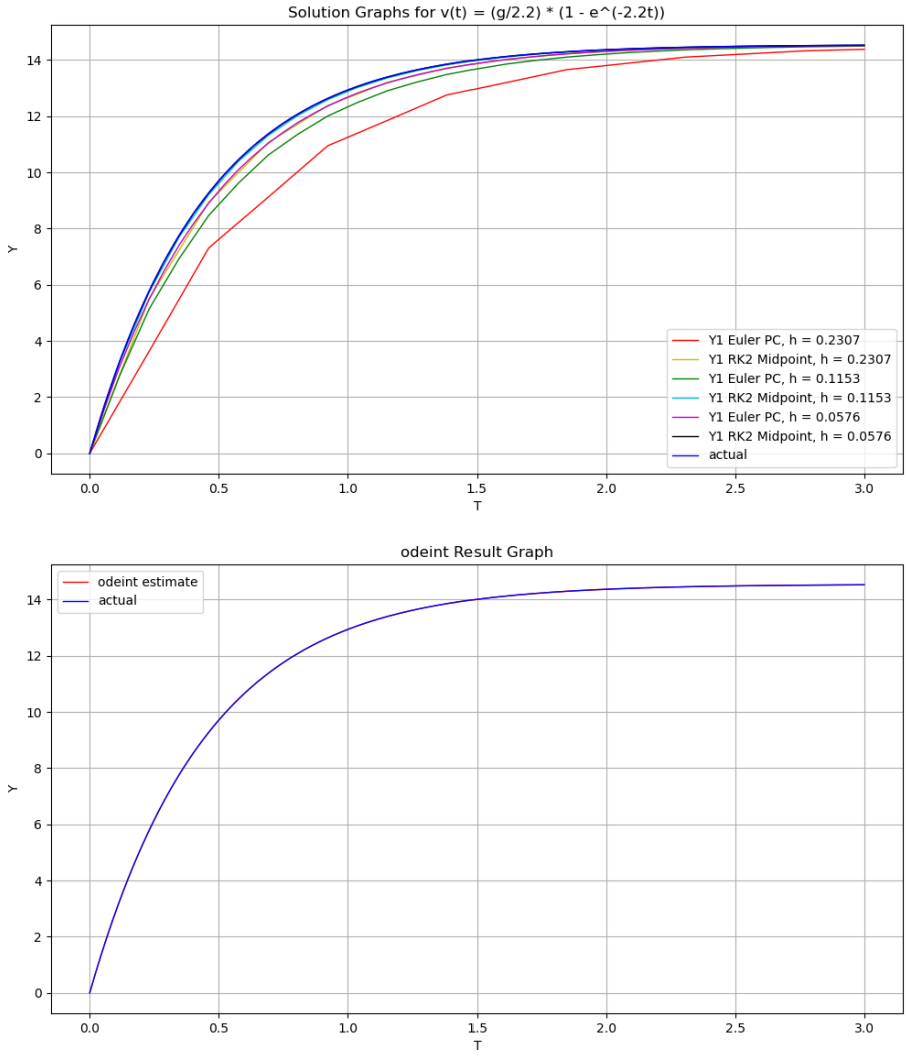

Figure 4: Solution Graphs for Part 1

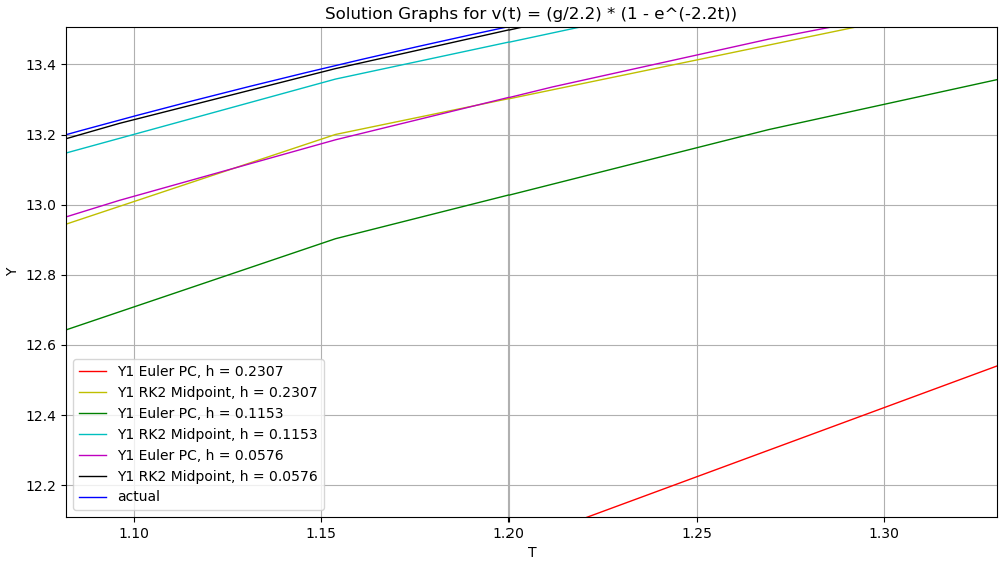

Figure 5: Graph of Solutions to Part 1 ODE - Zoomed in

Part 2

The objective of part two was to use any method to solve the ODE in Figure 6 to calculate the velocity of an object five seconds after it was dropped. The object has a mass of five slugs, gravity is 32 ft/s^2, and the initial velocity if 0 ft/s. The methods chosen to solve this problem were Euler PC and Euler's method. Both implementations of these methods are shown in Figure 2, Euler PC uses both forward and backward Euler's, while Euler's method is just forward Euler's. I decided to use these methods because I wanted to compare the accuracy of Euler's method and Euler PC. I would expect Euler PC to be better since it can be corrected, but the results from this project seem to indicate that either my correction code does not work, or Euler PC is just not a very accurate method. The velocity of the mass calculated by Euler's method was 35.772 ft/s and the velocity calculated by Euler PC was 35.760 ft/s. The actual velocity at t = 5s was calculated to be about 35.768 ft/s, so Euler's method is more accurate, but not by much. The code for the Euler PC and Euler's method is in Figure 22.

$$v'(t) = g - \frac{k}{m}v^2, v(0) = 0 \; on \; [0,5]$$

$$v(t) = sqrt\left(\frac{mg}{k}\right)\tanh\left(t * sqrt(\frac{kg}{m})\right)$$

$$k = \frac{1}{8}, g = 32\frac{ft}{s^2}, m = 5 \; slugs$$

Figure 6: ODE and Solution for Part2

Figure 7 contains two subplots. The first one contains the Euler PC and Euler' method estimates while the second one contains an estimated solution calculated by odeint. As always, both contain graphs of the actual solutions as well. One interesting thing to note is that both methods had about the same level of accuracy but converged towards the solution from opposite sides. Euler's method converged from the top, while Euler PC converged from the bottom. This is much more apparent in Figure 8, which shows a

zoomed in image of the solution graph. Both methods appear to converge at about the same rate, which is expected since they have the same error ($O(h^2)$). Since each estimate has a step-size of half the previous step-size, they should be getting about 4x more accurate, though it tends to be more like 2x. Neither method comes close to being as accurate as odeint, but they were not chosen for their accuracy anyways. The code to create the graphs is in Figure 23.

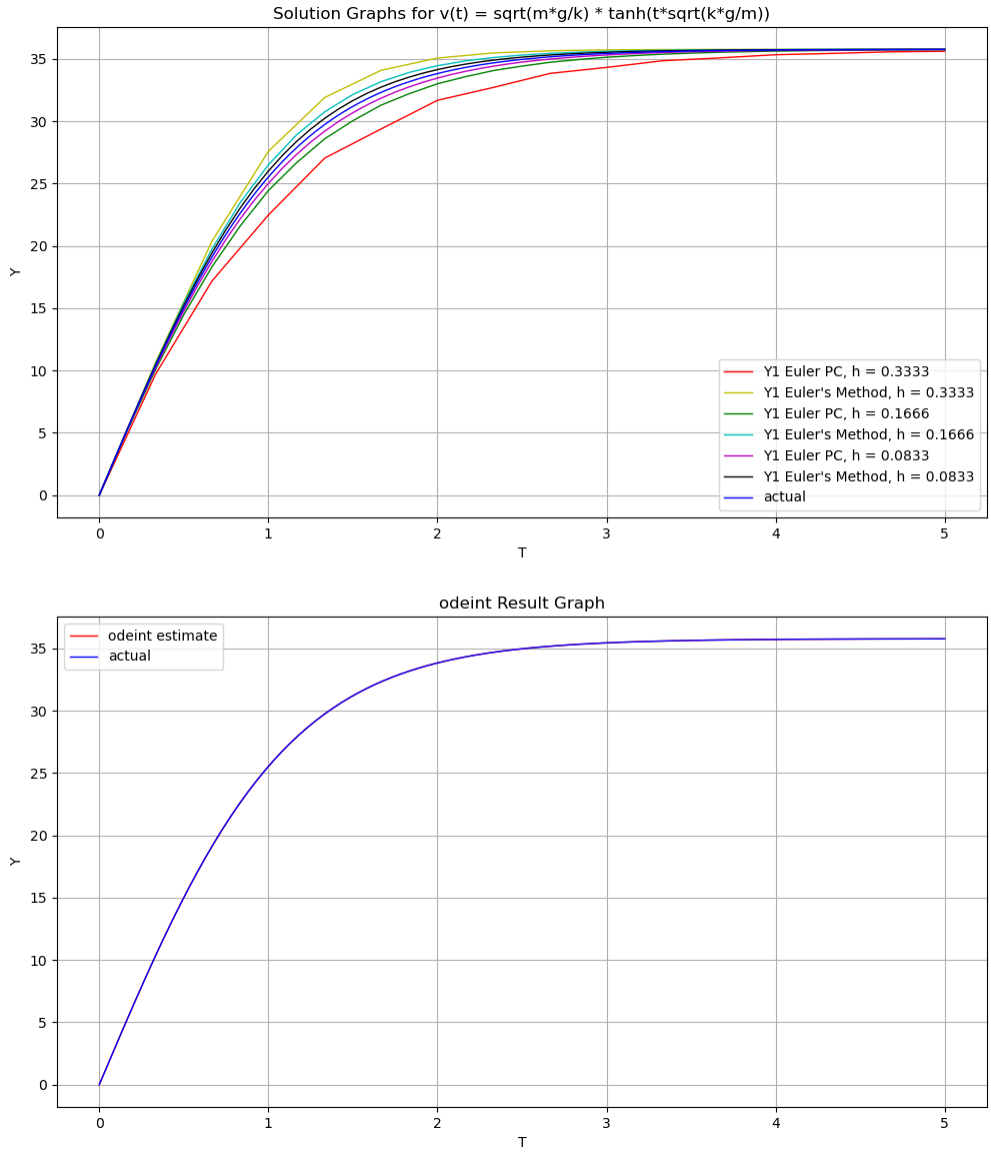

Figure 7: Solution Graphs for Part 2

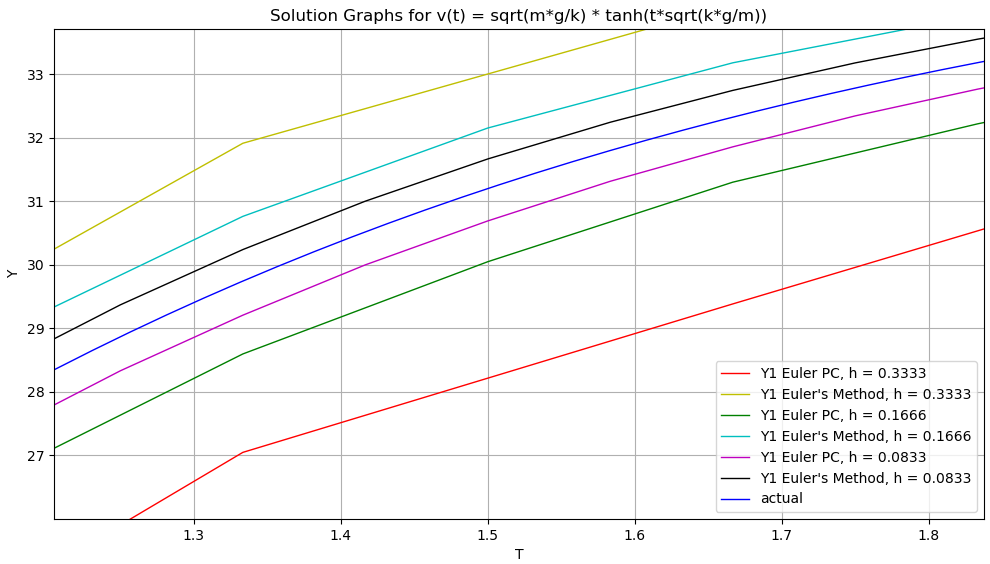

Figure 8: Solution Graphs for Part 2 - Zoomed in

Figure 9 shows the total number of iterations each method took to create the graphs shown above. Overall, even though Euler PC took many more iterations due to the corrections, it did not become noticeably (if at all) more accurate than Euler's method. This may be due to the solution having a large gradient, which implicit methods, like backward Euler's, tend to perform worse on.

```
EulerPC (h = 0.3333): 106
Euler's Method (h = 0.3333): 15
EulerPC (h = 0.1666): 101
Euler's Method (h = 0.1666): 30
EulerPC (h = 0.0833): 133
Euler's Method (h = 0.0833): 60
```

Figure 9: Iterations for Euler PC and Euler's Method

Part 3

The objective of Part 3 was to solve the second-order boundary-value problems shown in Figure 10 by solving a coefficient matrix created using the FOCFD and SOCFD methods. The implementation of both methods is shown in Figure 11. Basically, FOCFD was substituted for y' and SOCFD was substituted for y'' in the ODEs. Then a coefficient matrix was created to calculate the y-values at all the steps between the boundaries by solving the system Ay = b, where A is the coefficient matrix, y is the vector we are solving for, and b is a vector containing the constant values in the system. The code for the finite-difference method is shown in Figure 24.

$$\mathbf{1)}\ y'' = y' + 2y + \cos(x), y(0) = -0.3, y\left(\frac{\pi}{2}\right) = -0.1\ on\ \left[0, \frac{\pi}{2}\right]$$

$$y = -0.3\cos(x) - 0.1\sin(x)$$

$$\mathbf{2)}\ y'' = \frac{1}{x}y' - \frac{1}{x^2}y + \ln(x), y(1) = 0, y(2) = -2\ on\ [1,2]$$

$$y = -2x - \frac{1}{2}x\ ln(x) + \ln(x) + 2$$

Figure 10: ODEs and Solutions for Part 3

$$FOCFD$$

$$y' = \frac{y(x+h) - y(x-h)}{2h} + O(h^2)$$

$$SOCFD$$

$$y'' = \frac{y(x-h) - 2y(x) + y(x+h)}{h^2} + O(h^2)$$

Figure 11: FOCFD and SOCFD Implementations

Figure 12 contains the graphs for the first ODE. The subplots are the same as before; the first one has estimated solutions found by solving Ay = b, and the second one has a solution calculated by solve_bvp (a built-in BVP solver). Both subplots also have the exact solution. Overall, the finite-difference method worked fairly well. It is not all that accurate, but with small-enough step sizes, it gets a very close approximation of the solution. The lack of accuracy is expected since the finite-difference methods used in this project had $O(h^2)$ error. As the step-sizes decrease by half, the solutions quickly converge to the exact solution. The most obvious example of this is seen between the solutions with h = 0.2617 and h = 0.5235, where the solution improves a lot when the step-size decreases. In this case, solve_bvp, did not seem to outperform the finite-difference method, at least the solution with h = 0.1308. This is likely due to solve_bvp being made for all BVPs, so it uses some form of shooting method, whereas the finite-difference method is restricted to linear ODEs but performs better on them.

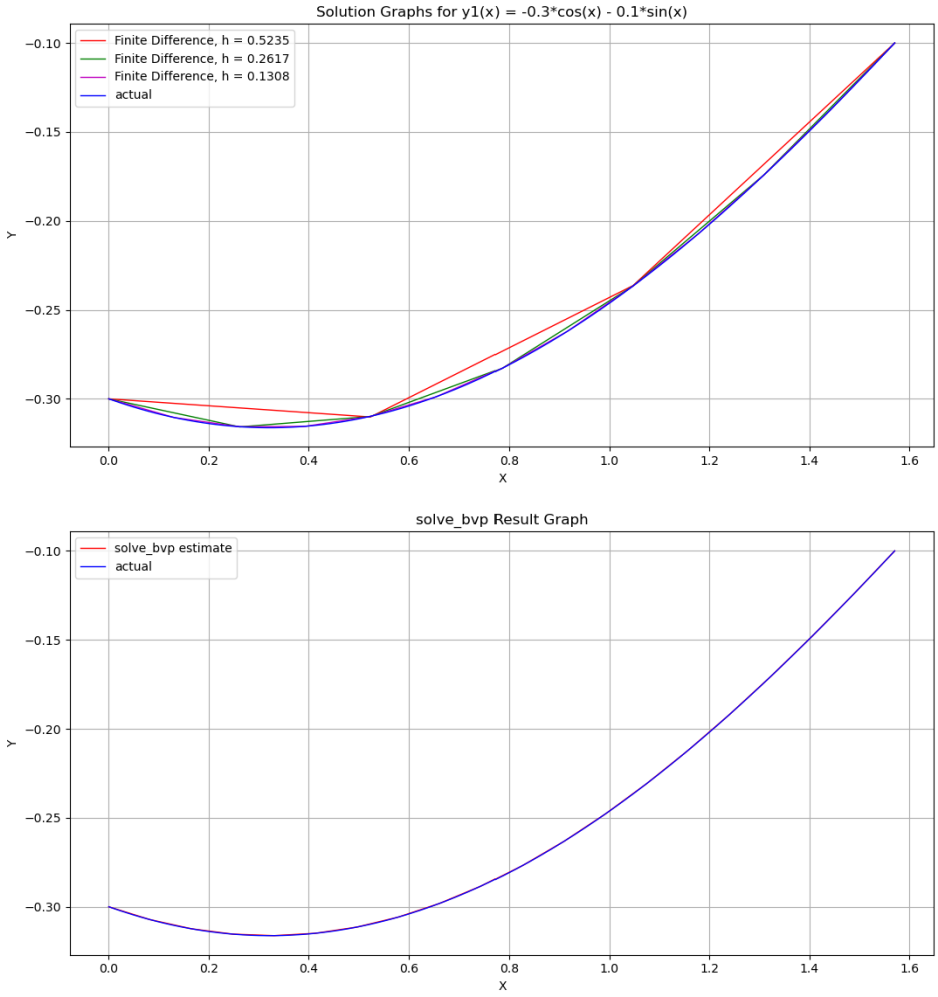

Figure 12: Part 3 Solution Graphs for First ODE

Figure 13 is exactly like Figure 12, just with solutions for the second ODE. One thing that is immediately obvious is that solve_bvp performed very poorly on this ODE, though that may be due to me not fully understanding how to use the function. The shooting method estimate graphs are easier to see in Figure 14, which has the same solutions, just zoomed in. As the step-sizes decrease, the estimated solutions converge towards the exact solution, with each estimate appearing to become about 2x more accurate when the step-size is cut in-half. Figure 25 shows the code for creating the graphs.

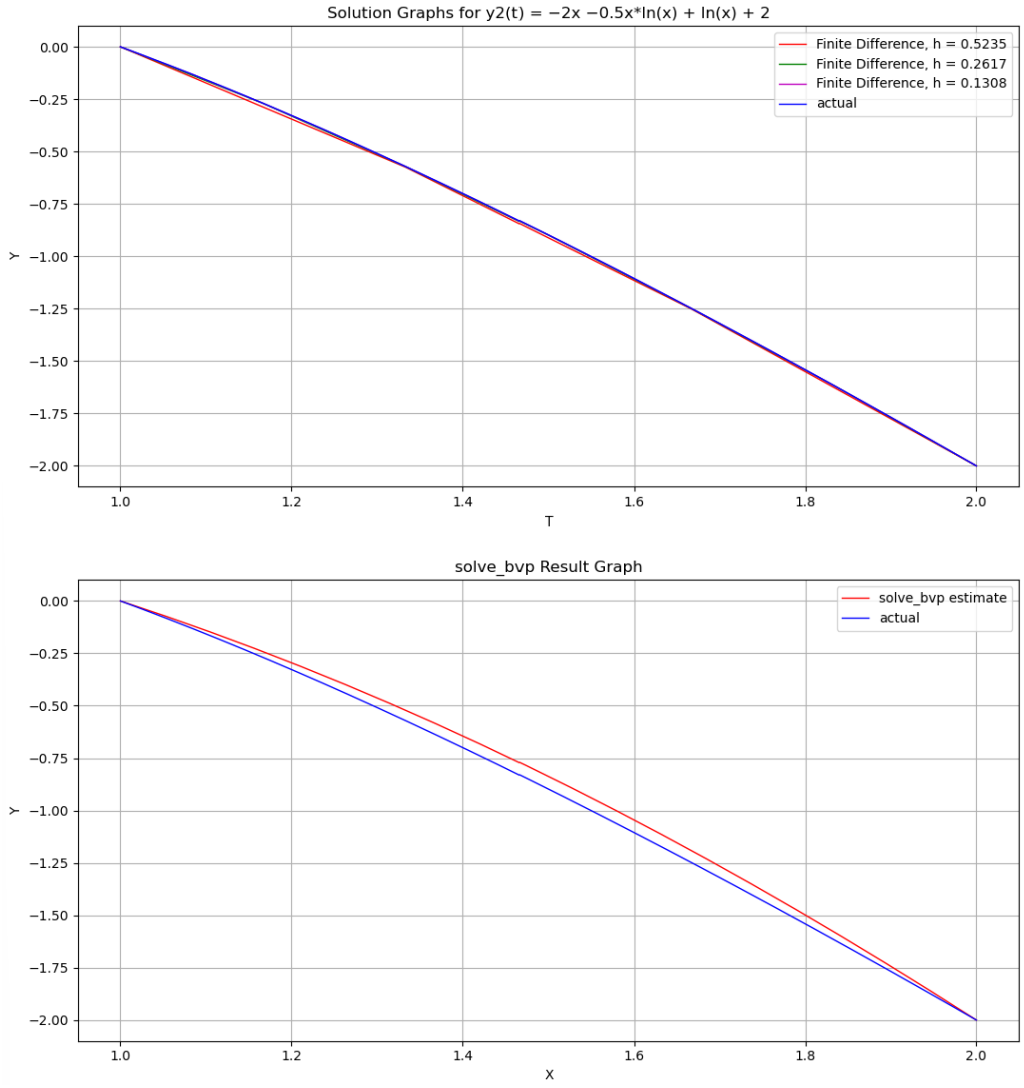

Figure 13: Part 3 Solution Graphs for Second ODE

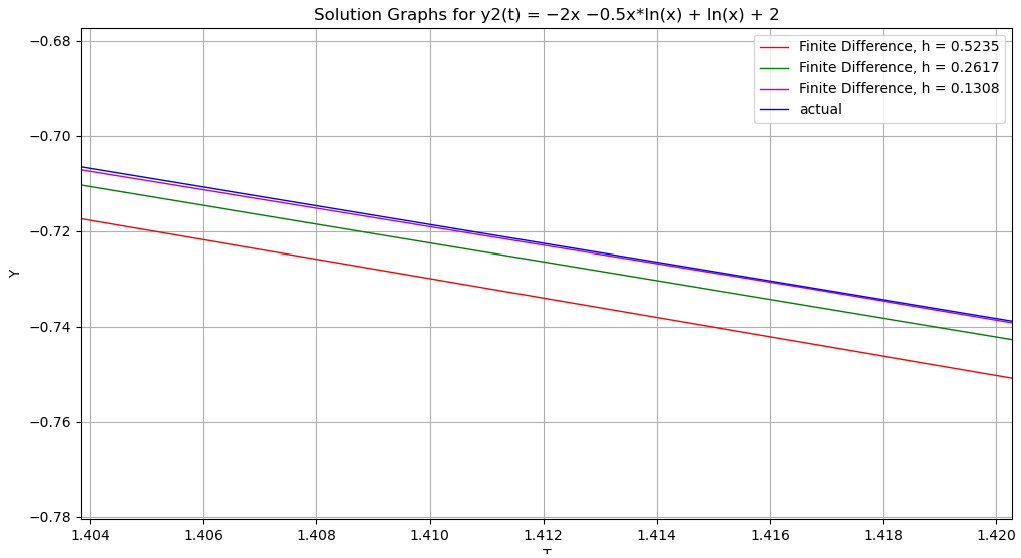

Figure 14: Part 3 Solutions to Second ODE - Zoomed in

Part 4

The objective of part four was to implement the shooting method to solve boundary-value problems. The benefit of the shooting method over the finite-difference method is that it can solve non-linear ODEs. The ODEs solved in this part are shown in Figure 15, along with their analytical solutions. The shooting method created for this project uses Euler's method to "shoot" and the secant method to improve the estimates for y'(0). The Euler's method implementation is the same as before, and the secant method implementation is shown in Figure 16. The x_n values are the initial y' values and the $f(x_n)$ are the error between the right boundary value and the estimated values of the right boundary (y1(b) - B). The code for Euler's method and the shooting method is shown in Figure 26.

$$\mathbf{1)}\ \mathrm{y}'' = \mathrm{x}(\mathrm{y}')^2, \mathrm{y}(0) = \frac{\pi}{2}, \mathrm{y}(2) = \frac{\pi}{4}\ on\ [0,2]$$

$$y = \operatorname{arccot}\left(\frac{x}{2}\right)$$

$$\mathbf{2)}\ y'' = y\cos^2(x) - \sin(x)\, e^{\sin(x)}, y(0) = 1, y(\pi) = 1\ on\ [0,\pi]$$

$$y = e^{\sin(x)}$$

Figure 15: ODEs and Solutions for Part 4

$$Secant\ Method$$

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

Figure 16: Secant Method Implementation

Figure 17 has a subplot containing the estimated solutions of the first ODE and a second subplot containing a solution estimated by solve_bvp. Both also contain the exact solution. Since the shooting method is using Euler's method, which has $O(h^2)$ error, I did not expect the estimates to be very accurate, however, it still performed well. A better image of the estimated graphs is shown in Figure 18 which shows the same solutions zoomed in. The solutions were estimated with 10, 20, and 40 steps. As expected, the 20-step estimate was the least accurate, and the estimates converged toward the actual solution as the step-sizes (h) decreased. Like in the previous parts, when the step-sizes decrease by half, the estimates become around 2x more accurate. Unlike with the finite-difference method, the solve_bvp function outperformed the shooting method estimates. An easy way to improve my current shooting method would be to implement it with RK4 instead of Euler's method. The code for creating the graphs for the first ODE is shown in Figure 27.

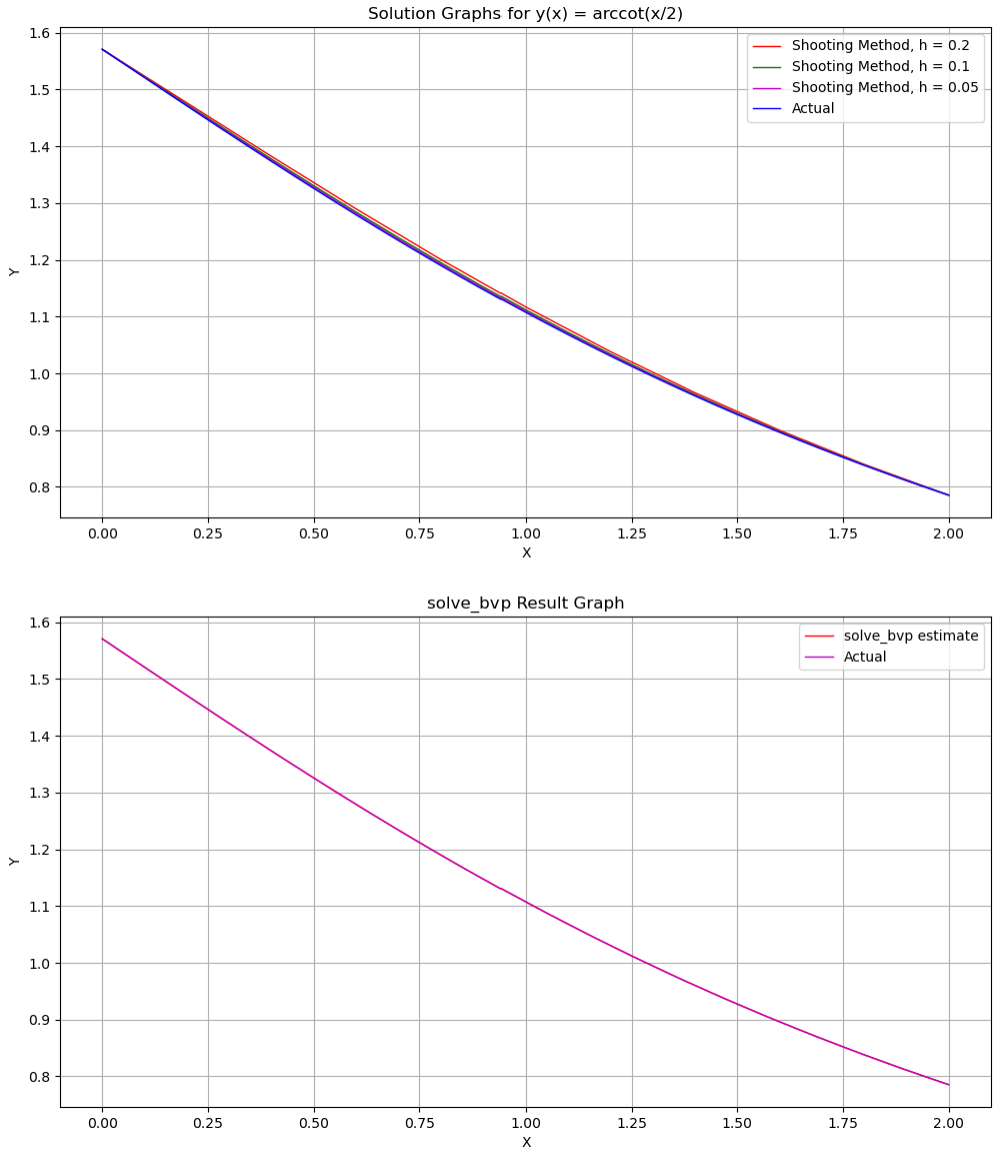

Figure 17: Part 4 Solution Graphs for the First ODE

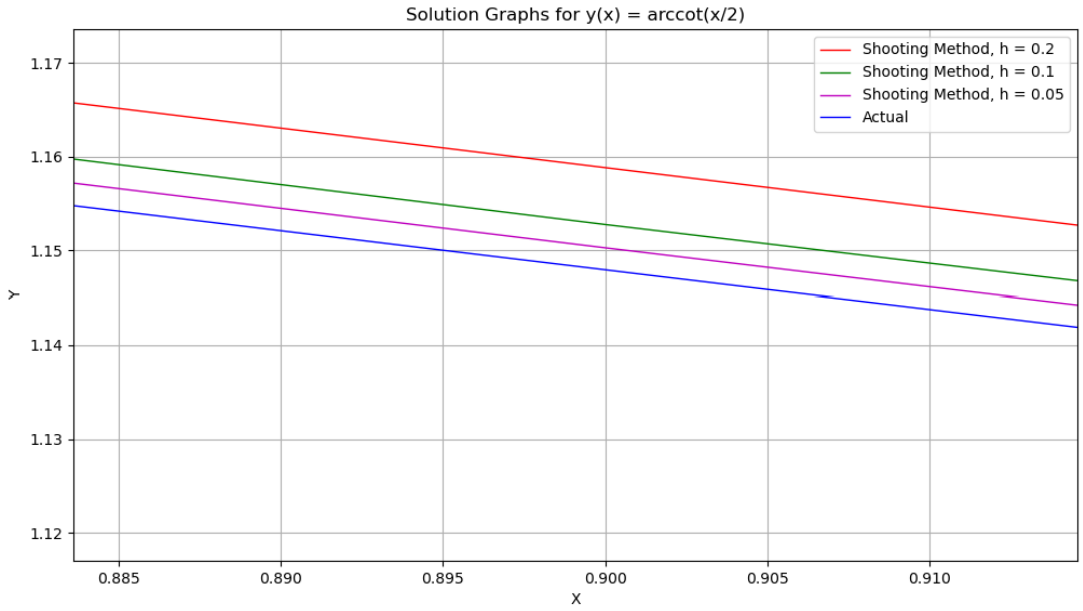

Figure 18: Part 4 Solution Graphs for ODE 1 - Zoomed in

Figure 19 contains graphs of the solutions to the second ODE. The format is the exact same as in Figure 17. One thing that is interesting is that as the estimated solutions improve, they shift left in addition to converging towards the solution from the bottom. It seems that the method undershoots the positive slopes and overshoots the negative slopes. This may be a result of the guesses for y'(0) being negative when they should be positive. Although it still works, a better method would be to allow the user to send in the initial guesses rather than hard-coding them. Nonetheless, the solutions still improve as the step-sizes decrease, though the method still performs much worse than it did on the first ODE. Similarly, solve_bvp also performs a lot worse on this ODE than it did on the first ODE, though it still does much better than the shooting method estimates. The code for creating the graphs for the second ODE is shown in Figure 28.

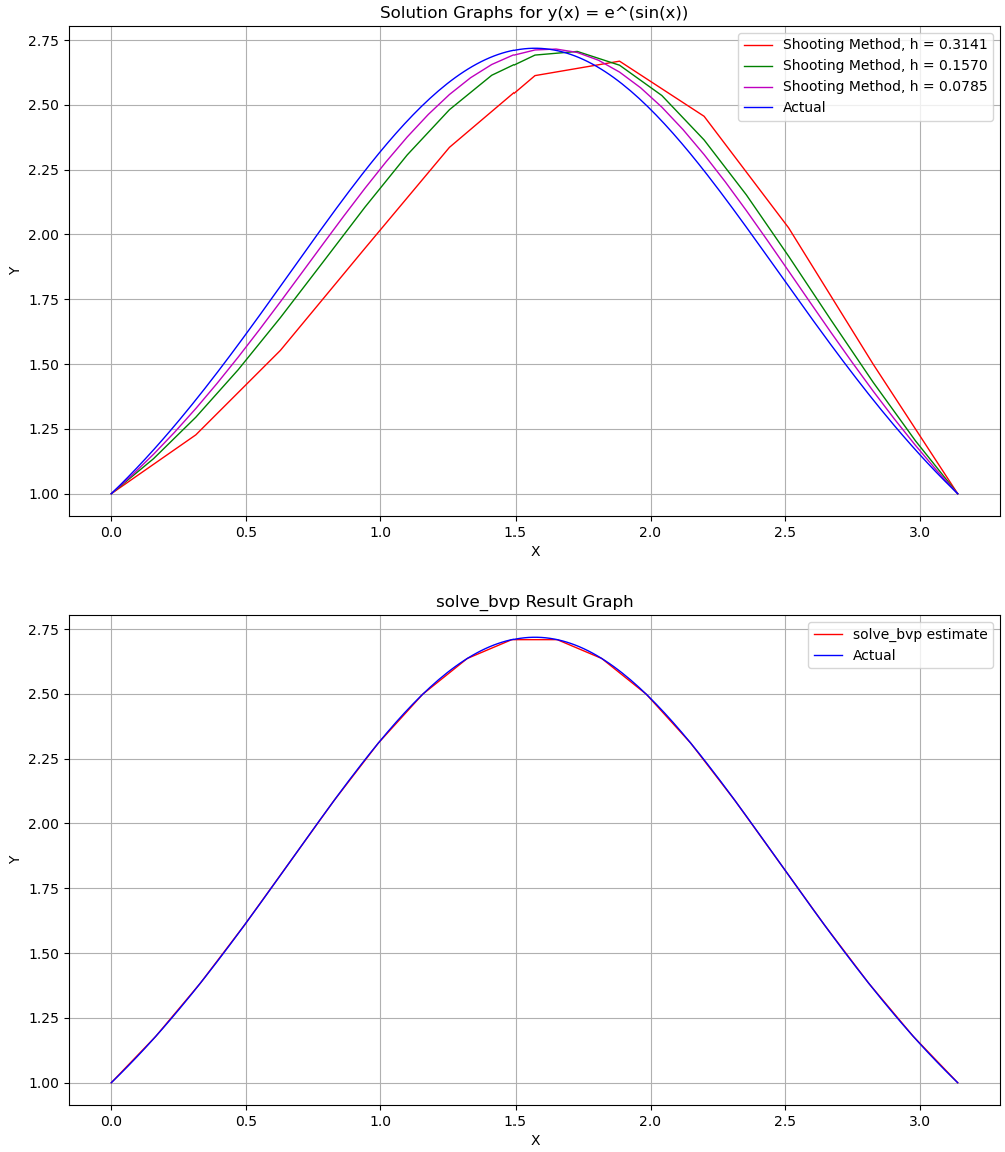

Figure 19: Part 4 Solution Graphs for ODE 2

Appendix

```python
def v_prime(t,v):
    return 32 - 2.2*v

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
    print("RK2 Solution: " + str(y_points[-1]))
    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))
    print("Solution: " + str(y_points[-1]))
    return t_points, y_points
```

Figure 20: RK2 and Euler PC Code for Part 1

```
t1_linspace = np.linspace(0,3,200)
steps = 13
colors1 = ['r', 'g', 'm']
colors2 = ['y', 'c', 'k']
fig1 = plt.figure(1, 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(v_prime, [0,3], 0, 3/steps, 0.0001)
    plt.subplot(2,1,1)
    #Graph Y1 Euler PC
    plt.plot(t1, y1, color = colors1[i], linewidth=1, label = 'Euler PC, h = ' + str(3/steps)[:6])
    #Solve Y1 RK2
    t1, y1 =  RK2_Midpoint(v_prime, [0,3], 0, 3/steps)
    #Graph Y1 RK2
    plt.plot(t1, y1, color = colors2[i], linewidth=1, label = 'RK2 Midpoint, h = ' + str(3/steps)[:6])
    steps *= 2


fig1 = plt.figure(1, figsize=(12,14))
plt.subplot(2,1,1)
plt.grid()
plt.plot(t1_linspace, (32/2.2) * (1 - e**(-2.2*t1_linspace)), color = 'b', label = 'actual', linewidth=1) #Graph Exact
plt.title('Solution Graphs for v(t) = (g/2.2) * (1 - e^(-2.2t))')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(v_prime, 0, 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, (32/2.2) * (1 - e**(-2.2*t1_linspace)), color = 'b', label = 'actual', linewidth=1)
plt.title('odeint Result Graph')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()
```

Figure 21: Graphing Code for Part 1

```python
def v_prime(t,v):
    return g - (k/m)*(v**2)

def EulersMethod(f, bounds, y0, h):
    totalIterations = 0
    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])
        y_points.append(y_points[-1] + h*f(t_points[-1], y_points[-1]))
        t_points.append(t_points[-1] + h)
        totalIterations += 1
    print("Euler\'s Method (h = " + str(h)[:6] + "): " + str(totalIterations)) #Gets most recent t-value and increments it by h
    print("Solution: " + str(y_points[-1]))
    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))
    print("Solution: " + str(y_points[-1]))
    return t_points, y_points
```

Figure 22: Euler PC and Euler's Method Code for Part 2

```
t1_linspace = np.linspace(0,5,200)
steps = 15
colors1 = ['r', 'g', 'm']
colors2 = ['y', 'c', 'k']
fig1 = plt.figure(1, 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(v_prime, [0,5], 0, 5/steps, 0.0001)
    plt.subplot(2,1,1)
    #Graph Y1 Euler PC
    plt.plot(t1, y1, color = colors1[i], linewidth=1, label = 'Euler PC, h = ' + str(5/steps)[:6])
    #Solve Y1 RK2
    t1, y1 =  EulersMethod(v_prime, [0,5], 0, 5/steps)
    #Graph Y1 RK2
    plt.plot(t1, y1, color = colors2[i], linewidth=1, label = 'Euler\'s Method, h = ' + str(5/steps)[:6])
    steps *= 2

fig1 = plt.figure(1, figsize=(12,14))
plt.subplot(2,1,1)
plt.grid()
print((m*g/k)**(0.5) * np.tanh(5 * (k*g/m)**(0.5)))
plt.plot(t1_linspace, (m*g/k)**(0.5) * np.tanh(t1_linspace * (k*g/m)**(0.5)), color = 'b', label = 'actual', linewidth=1) #Graph Exact
plt.title('Solution Graphs for v(t) = sqrt(m*g/k) * tanh(t*sqrt(k*g/m))')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph odeint
sol = odeint(v_prime, 0, 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.sqrt(m*g/k) * np.tanh(t1_linspace * np.sqrt(k*g/m)), 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 2

```python
def y1_sys(x, y):
    return np.vstack((y[1], y[1] + 2*y[0] + np.cos(x)))

def y2_sys(x, y):
    return np.vstack((y[1], y[1]*(1/x) + y[0]*(1/(x**2)) + np.log(x)))

def bc1(a, b):
    return np.array([a[0] + 0.3, b[0] + 0.1])

def bc2(a, b):
    return np.array([a[0], b[0] + 2])

def p1(x):
    return -1
def q1(x):
    return -2
def r1(x):
    return np.cos(x)

def p2(x):
    return -1/x
def q2(x):
    return 1/x**2
def r2(x):
    return np.log(x)/(x**2)

def finiteDifferenceBVP(p, q, r, A, B, bounds, h):
    x_points = [bounds[0]]
    y_points = [A]
    dim = int(((bounds[1] - bounds[0])/h) - 1)
    #Create Matrices
    A_matrix = np.zeros((int(dim), int(dim)))
    b_matrix = np.zeros((int(dim), 1))
    a = bounds[0]
    b = bounds[1]
    col = 0

    #Fill in first row
    A_matrix[0,:2] = [(-2/(h**2) + q(a + h)), (1/(h**2) + p(a + h)/(2*h))]
    b_matrix[0] = [r(a + h) - A/(h**2) + (p(a + h)*A)/(2*h)]
    a = a + h
    x_points.append(a)
    #Fill in middle rows of matrices
    for row in range(1, dim-1):
        a = a + h
        A_matrix[row, col:(col+3)] = [(1/(h**2) - p(a)/(2*h)), (q(a) - 2/(h**2)), (1/(h**2) + p(a)/(2*h))]
        b_matrix[row] = [r(a)]
        x_points.append(a)
        col = col + 1
    #Fill in last row
    A_matrix[dim - 1, dim-2:] = [(1/(h**2) - p(b - h)/(2*h)), (-2/(h**2) + q(b - h))]
    b_matrix[dim-1] = [r(b - h) - B/(h**2) - (p(b - h)*B)/(2*h)]
    x_points.append(a + h)
    #Solve Matrix
    y_matrix = np.linalg.solve(A_matrix, b_matrix)
    #Get points for y from y_matrix
    for i in range(0, len(y_matrix[:,0])):
        y_points.append(y_matrix[i, 0])
    #Append right bound
    x_points.append(bounds[1])
    y_points.append(B)
    return x_points, y_points
```

Figure 24: Finite Difference Method Code for Part 3

```
fig1 = plt.figure(1, figsize=(12,14))
plt.subplot(2,1,1)
plt.grid()
plt.plot(x1_linspace, -0.3*np.cos(x1_linspace) - 0.1*np.sin(x1_linspace), color = 'b', label = 'actual', linewidth=1) #Graph Exact
plt.title('Solution Graphs for y1(x) = -0.3*cos(x) - 0.1*sin(x)')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()

#Graph solve_bvp
x_a = np.linspace(0, np.pi/2, 20)
y_a = np.zeros((2, x_a.size))
sol = solve_bvp(y1_sys, bc1, x_a, y_a)
plt.subplot(2,1,2)
plt.grid()
plt.plot(x_a, sol.y[0], color = 'r', linewidth=1, label = 'solve_bvp estimate')
plt.plot(x1_linspace, -0.3*np.cos(x1_linspace) - 0.1*np.sin(x1_linspace), color = 'b', label = 'actual', linewidth=1) #Graph Exact
plt.title('solve_bvp Result Graph')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()

#-------------------- EQN 2 ------------------------
x2_linspace = np.linspace(1,2,200)
fig2 = plt.figure(2, figsize=(12,14))
plt.subplot(2,1,1)
plt.grid()
plt.plot(x2_linspace, -2*x2_linspace -0.5*x2_linspace*np.log(x2_linspace) + np.log(x2_linspace) + 2, color = 'b', label = 'actual', linewidth=1) #Graph Exact Y2
plt.title('Solution Graphs for y2(t) = -2x -0.5x*ln(x) + ln(x) + 2')
plt.xlabel('T')
plt.ylabel('Y')
plt.legend()

#Graph solve_bvp
x_b = np.linspace(1, 2, 20)
y_b = np.zeros((2, x_b.size))
sol = solve_bvp(y2_sys, bc2, x_b, y_b)
plt.subplot(2,1,2)
plt.grid()
plt.plot(x_b, sol.y[0], color = 'r', linewidth=1, label = 'solve_bvp estimate')
plt.plot(x2_linspace, -2*x2_linspace -0.5*x2_linspace*np.log(x2_linspace) + np.log(x2_linspace) + 2, color = 'b', label = 'actual', linewidth=1) #Graph Exact
plt.title('solve_bvp Result Graph')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
```

Figure 25: Graphing Code for Part 3

```python
#First ODE Functions
def y1_2(x, y1, y2):
    return x*(y2**2)
def y1_1(x, y1, y2):
    return y2

def y1_sys(x, y):
    return np.vstack((y[1], x*(y[1])**2))

def bc1(a, b):
    return np.array([a[0] - np.pi/2, b[0] - np.pi/4])

#Second ODE Functions
def y2_2(x, y1, y2):
    return y1*(np.cos(x))**2 - np.sin(x)*e**(np.sin(x))

def y2_1(x, y1, y2):
    return y2

def y2_sys(x, y):
    return np.vstack((y[1], y[0]*(np.cos(x))**2 - np.sin(x)*e**(np.sin(x))))

def bc2(a, b):
    return np.array([a[0] - 1, b[0] - 1])

def EulersMethod(f1, f2, bounds, y10, y20, h):
    steps = int((bounds[1] - bounds[0]) / h)
    y1 = [y10]
    y2 = [y20]
    t = [bounds[0]]
    tempVal = 0
    for i in range(steps):
        tempVal = y1[-1] + h*f1(t[-1], y1[-1], y2[-1])
        y2.append(y2[-1] + h*f2(t[-1], y1[-1], y2[-1]))
        y1.append(tempVal)
        t.append(t[-1] + h)
    return t, y1, y2

def ShootingMethod(f1, f2, A, B, bounds, h):
    t1, y1, y2 = EulersMethod(f1, f2, bounds, A, -0.7, h)#Get an initial guess
    #prev_x, cur_x, next_x are the guesses for the initial value of y2. Used in root finding
    prev_x = -0.7
    cur_x = -0.9
    g = y2[-1] #g is the most recent estimate of B
    while(abs(B - y1[-1]) > 0.00000001):
        t1, y1, y2 = EulersMethod(f1, f2, bounds, A, cur_x, h)#Get next guess
        #Secant Method: guess values are x, (g - B) and (y22[-1] - b) are f(x) (errors)
        next_x = (prev_x * (y1[-1] - B) - cur_x * (g - B)) / (y1[-1] - g)
        prev_x = cur_x
        cur_x = next_x
        g = y1[-1]
    return t1, y1 ,y2
```

Figure 26: Euler's Method and Shooting Method Code for Part 4

```
colors1 = ['r', 'g', 'm']
colors2 = ['y', 'c', 'k']
fig1 = plt.figure(1, figsize=(12,14))
plt.subplot(2,1,1)

x1 = np.linspace(0, 2, 200)
steps = 10
x, y1, y2= ShootingMethod(y1_1, y1_2, np.pi/2, np.pi/4, [0,2], 2/steps)
#Plot Shooting Method Estimate
plt.plot(x, y1, color = colors1[0], linewidth=1, label = 'Shooting Method, h = ' + str(2/steps)[:6])
#Plot with 20 steps
steps = 20
x, y1, y2= ShootingMethod(y1_1, y1_2, np.pi/2, np.pi/4, [0,2], 2/steps)
plt.plot(x, y1, color = colors1[1], linewidth=1, label = 'Shooting Method, h = ' + str(2/steps)[:6])
#Plot with 40 steps
steps = 40
x, y1, y2= ShootingMethod(y1_1, y1_2, np.pi/2, np.pi/4, [0,2], 2/steps)
plt.plot(x, y1, color = colors1[2], linewidth=1, label = 'Shooting Method, h = ' + str(2/steps)[:6])

#Plot Actual
cotf = []
for i in x1:
    cotf.append(sp.acot(i/2))#Does not like graphing arccot directly
plt.plot(x1, cotf, color = 'b', linewidth=1, label = 'Actual')
plt.grid()
plt.title('Solution Graphs for y(x) = arccot(x/2)')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()

#Graph solve_bvp
x_a = np.linspace(0, 2, 20)
y_a = np.zeros((2, x_a.size))
sol = solve_bvp(y1_sys, bc1, x_a, y_a)
plt.subplot(2,1,2)
plt.grid()
plt.plot(x_a, sol.y[0], color = 'r', linewidth=1, label = 'solve_bvp estimate')
plt.plot(x1, cotf, color = colors1[2], linewidth=1, label = 'Actual')  #Graph Exact
plt.title('solve_bvp Result Graph')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
```

Figure 27: Graphing Code for Part 4 ODE 1

```
fig2 = plt.figure(2, figsize=(12,14))
plt.subplot(2,1,1)
x2 = np.linspace(0, np.pi, 200)
steps = 10
x, y1, y2= ShootingMethod(y2_1, y2_2, 1, 1, [0,np.pi], np.pi/steps)
#Plot Shooting Method Estimate
plt.plot(x, y1, color = colors1[0], linewidth=1, label = 'Shooting Method, h = ' + str(np.pi/steps)[:6])
#Plot with 20 steps
steps = 20
x, y1, y2= ShootingMethod(y2_1, y2_2, 1, 1, [0,np.pi], np.pi/steps)
plt.plot(x, y1, color = colors1[1], linewidth=1, label = 'Shooting Method, h = ' + str(np.pi/steps)[:6])
#Plot with 40 steps
steps = 40
x, y1, y2= ShootingMethod(y2_1, y2_2, 1, 1, [0,np.pi], np.pi/steps)
plt.plot(x, y1, color = colors1[2], linewidth=1, label = 'Shooting Method, h = ' + str(np.pi/steps)[:6])
plt.plot(x2, e**(np.sin(x2)), color = 'b', linewidth=1, label = 'Actual')  #Graph Exact
plt.grid()
plt.title('Solution Graphs for y(x) = e^(sin(x))')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()

#Graph solve_bvp
x_a = np.linspace(0, np.pi, 20)
y_a = np.zeros((2, x_a.size))
sol = solve_bvp(y2_sys, bc2, x_a, y_a)
plt.subplot(2,1,2)
plt.grid()
plt.plot(x_a, sol.y[0], color = 'r', linewidth=1, label = 'solve_bvp estimate')
plt.plot(x2, e**(np.sin(x2)), color = 'b', linewidth=1, label = 'Actual')  #Graph Exact
plt.title('solve_bvp Result Graph')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
```

Figure 28: Graphing Code for Part 4 ODE 2