MACHINE PROBLEM NO. 3

ES 204 1st Sem 2023-2024

As a student of the University of the Philippines, I pledge to act ethically and uphold the values of honor and excellence.

I understand that suspected misconduct on this Assignment will be reported to the appropriate office and if established, will result in disciplinary action in accordance with University rules, policies and procedures. I may work with others only to the extent allowed by the Instructor.

Name: **Jeryl Salas**
Student Number: **202321128**

GENERAL INSTRUCTIONS: Solve all the problems using appropriate numerical and programming techniques, independently and completely. Cite all references or any assistance that you received during the development of your solution. Submit a brief but comprehensive EXECUTIVE SUMMARY of your solution to the problems. Add the complete solution(s) and source codes in the APPENDIX.

PROBLEM 1 [30 points]: A solid object is cooled at an ambient temperature of $300\ K$. The heat lost is given by the equation

$$\frac{dT}{dt} = -3 \times 10^{-11}(T^4 - 300^4), \quad T(0) = 1000K$$

where T is the temperature of the object and t is time in seconds. Plot the values of T from $t = 0$ to $t = 250$ using the forward and the backward Euler methods. Justify your choice of step sizes.

1.1 Method(s) of solution

We were tasked to plot the values of T using forward and backward Euler methods to create an approximate solution of the Ordinary Differential Equation (ODE). The ODE represents the rate of change of temperature (T) with respect to time (t). Non-linear ODE are very difficult to solve analytically so we treat this differential equation as an Initial Value Problem (IVP) assuming that this equation is continuous.

$$\frac{dT}{dt} = f(t,T)\ for\ t > t_o \qquad T(t_o) = T_o$$

The formula for forward Euler method is given by:

$$T_{i+1} = T_i + \Delta t\, f(t_i, T_i) \quad (\Delta t = h) \tag{1}$$

The formula for backward Euler method is given by:

$$T_{i+1} = T_i + \Delta t\, f(t_{i+1}, T_{i+h}) \quad (\Delta t = h) \tag{2}$$

This equation is difficult to implement since it's implicit. We will instead treat this as a root finding problem. The equation can be expressed as:

$$f(T_{i+1}) = T_i - T_{i+1} + \Delta t\, f(t_{i+1}, T_{i+h}) \tag{3}$$

Solve root finding problem $f(T_{i+1})$ to find T_{i+1}. We will be using Newton Raphson method to solve the root finding problem. The equation to solve the roots of $f(T_{i+1})$ can be expressed as:

$$T_{i+1}^{(k+1)} = T_{i+1}^{(k)} - \frac{f(T_{i+1}^{(k)})}{f'(T_{i+1}^{(k)})} \tag{4}$$

Where:

- $T_{i+1}^{(k)}$ is the current estimation of T_{n+1} at time k.
- $f(T_{i+1}^{(k+1)})$ is the function evaluated at $T_{i+1}^{(k+1)}$.
- $f'(T_{i+1}^{(k+1)})$ is the derivative of the function f evaluated at $T_{i+1}^{(k)}$.

The heat loss function can't be solved analytically but we can find the equilibrium temperature where $\frac{dT}{dt} = 0$. This will be plot on the chart as well for reference.

$$0 = -3 \times 10^{-11}(T^4 - 300^4)$$

$$T = 300, \qquad where \ \frac{dT}{dt} = 0$$

The steps used to find the approximate solution are as follows:

1. Import numpy (for matrix operations) and matplotlib (for plotting the temperature with respect to time)
2. Define heat loss function
3. For different step sizes (h), perform the forward Euler method and backward Euler method.

```
function FORWARD_EULER_METHHOD
	initialize T values as an array
	loop do
		T = current_T + step_size * HEAT_LOSS(current_T)
		add to T values
	return T values

function BACKWARD_EULER_METHOD
	initialize T values as an array
	loop do
		current_T = T_values[-1]
		loop do
			next_T = current_T / (1 + h)
			if abs(next_T - current_T) < tolerance
				end loop
			current_T = next_T
		next_T add to T values
	return T values
```

Figure 1: Forward and Backward Euler method algorithm

4. Plot the results for different step sizes using Matplotlib

1.2 Results

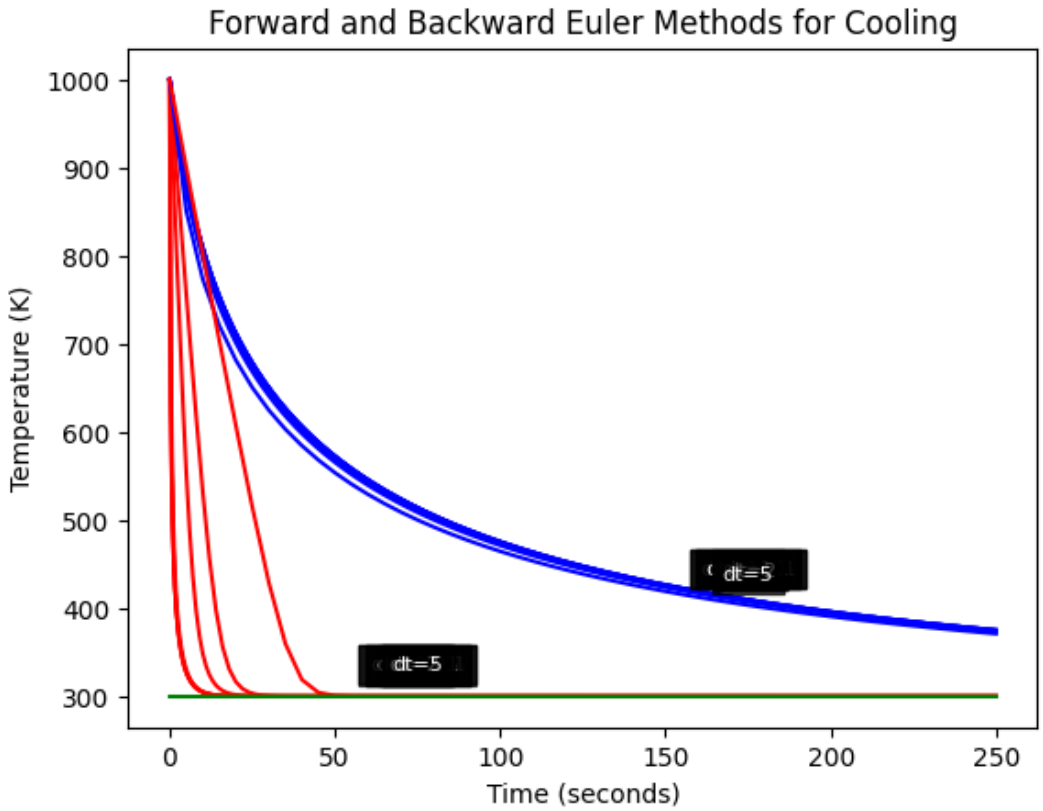


Figure 2: Forward and Backward Euler method for cooling approximated curves for each step size (dt). dt = 0.005, 0.001, 0.05, 0.01, 1, 2, 5

Figure 2 shows a plot of approximated curves using Forward Euler (FE) and Backward Euler (BE) using different step sizes (dt). All forward Euler approximations are highlighted by blue while backward Euler approximations are highlighted by red. As we can see, the curves are smooth where the BE approaches equilibrium temperature $T = 300$ more aggressively than FE approximated curves.

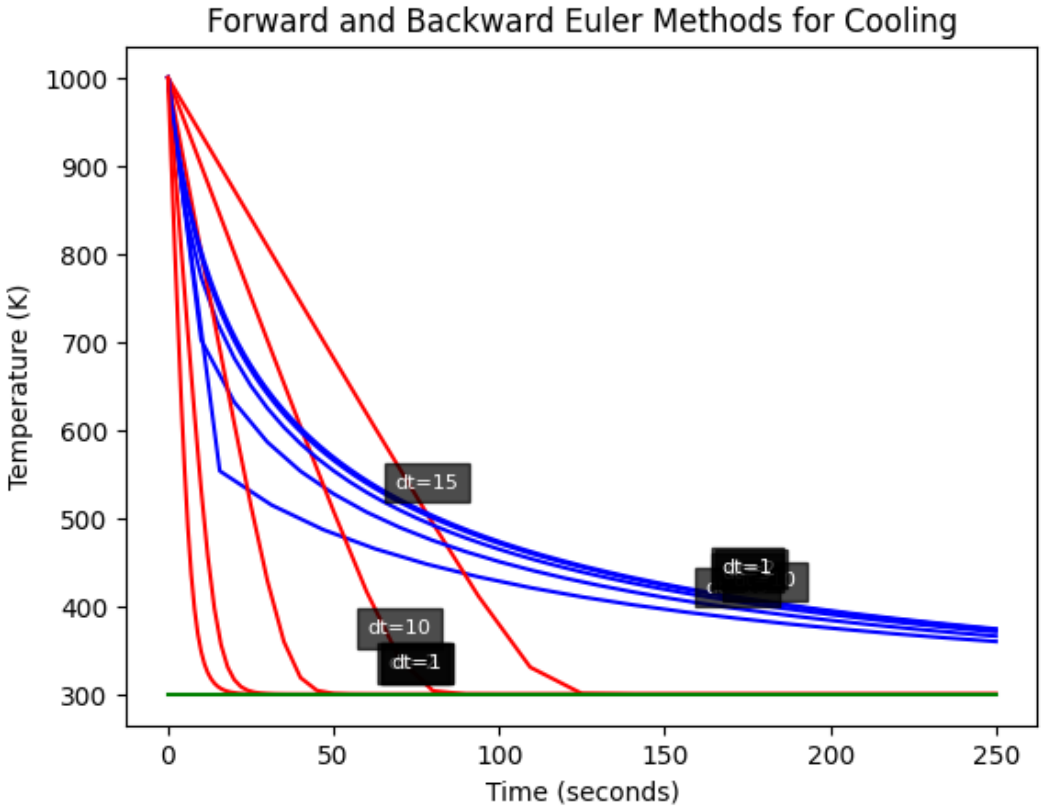


Figure 3: Forward and Backward Euler method for cooling approximated curves for each step size (dt). dt = 15, 10, 5, 2, 1

Figure 3 shows a plot of approximated curves using FE and BE using bigger step sizes (dt). As we can see, the curves are sharper with curves with larger step size takes more time to reach the equilibrium temperature. The behavior of FE and BE are more of ways the same as FE curves take more time to reach equilibrium temperature than the BE.

1.3 Problems encountered

When employing Forward and Backward Euler to approximate solutions for differential equations determining the local truncation error (LTE) and global error can be challenging when an exact solution is unattainable for the heat loss function. If it were possible to compute LTE and global error, we would've find the discrepancy between the approximated value and the exact value either at each step or global. Without these error computations, we would get a hard time determining the accuracy of FE and BE. Because of this, we can't determine the reliability of these methods for the heat loss function.

1.4 References

- Brorson, S. (n.d.). 1.3: Backward Euler method. In Numerically Solving Ordinary Differential Equations. Northeastern University. Retrieved from https://math.libretexts.org/Bookshelves/Differential_Equations/Numerically_Solving_Ordinary_Differential_Equations_(Brorson)/01%3A_Chapters/1.03%3A_Backward_Euler_method
- Zeltkevic, M. (1998, April 15). Forward and Backward Euler Methods. Retrieved from https://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node3.html
- ES_204_L6_Ordinary_Differential_Equations.pdf

PROBLEM 2 [30 points]: Solve and plot in one graph

$$\frac{dy}{dt} = \sin t + \cos y + \sin z, \quad y(0) = 0$$

$$\frac{dz}{dt} = \cos t + \sin z, \qquad z(0) = 0$$

$t \in [0,20]$ with 100 intervals. Use a fourth-order *Runge-Kutta* method.

2.1 Method(s) of solution

We were tasked to solve and plot two function in one graph using the fourth-order *Runge-Kutta* method. Given $\frac{dy}{dt} = funcy(t, y, z)$ and $\frac{dz}{dt} = funcz(t, z)$. The formula of the *Runge-Kutta* method is given by:

$$k_1^y = h \cdot funcy(t_n, y_n, z_n) \tag{5}$$

$$k_1^z = h \cdot funcz(t_n, z_n) \tag{6}$$

$$k_2^y = h \cdot funcy\left(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}, z_n + \frac{k_1}{2}\right) \tag{7}$$

$$k_2^z = h \cdot funcz\left(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}, z_n + \frac{k_1}{2}\right) \tag{8}$$

$$k_3^y = h \cdot funcy\left(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}, z_n + \frac{k_2}{2}\right) \tag{9}$$

$$k_3^z = h \cdot funcz(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}, z_n + \frac{k_2}{2}) \tag{10}$$

$$k_4^y = h \cdot funcy(t_n + h, y_n + k_3, z_n + k_3)) \tag{11}$$

$$k_4^z = h \cdot funcz(t_n + h, y_n + k_3) \tag{12}$$

$$y_{n+1}^y = y_n^y + \frac{k_1^y}{6} + \frac{k_2^y}{3} + \frac{k_3^y}{3} + \frac{k_4^y}{6} + o(h^5) \tag{13}$$

$$y_{n+1}^z = y_n^z + \frac{k_1^z}{6} + \frac{k_2^z}{3} + \frac{k_3^z}{3} + \frac{k_4^z}{6} + o(h^5) \tag{14}$$

$$h = \frac{t[1] - t[0]}{n} \tag{15}$$

The steps used to find the approximate solution are as follows:

1. Import numpy (for matrix operations) and matplotlib (for plotting two functions with respect to time)
2. Define function y and z.
3. Compute step size (h) by subtracting the spans and divide by number of intervals. Please see *eqn. 15*
4. For number of intervals (n), update y_n^y and y_n^z using equations 5 to 14.

```
function RANGE_KUTTA_FOURTH_ORDER
   compute h step size
   initialize t values
   initialize y values and append y0
   initialize z values and append z0

   loop do
         compute k1z, k1y
         compute k2z, k2y
         compute k3z, k3y
         compute k4z, k4y
         compute new y_y and append to y values
         compute new y_z and append to z values
   return t values, y values, and z values
```

Figure 4: Fourth order Range-Kutta Algorithm

5. Plot the y values for both $funcy(t, y, z)$ and $funcz(t, z)$

2.2 Results

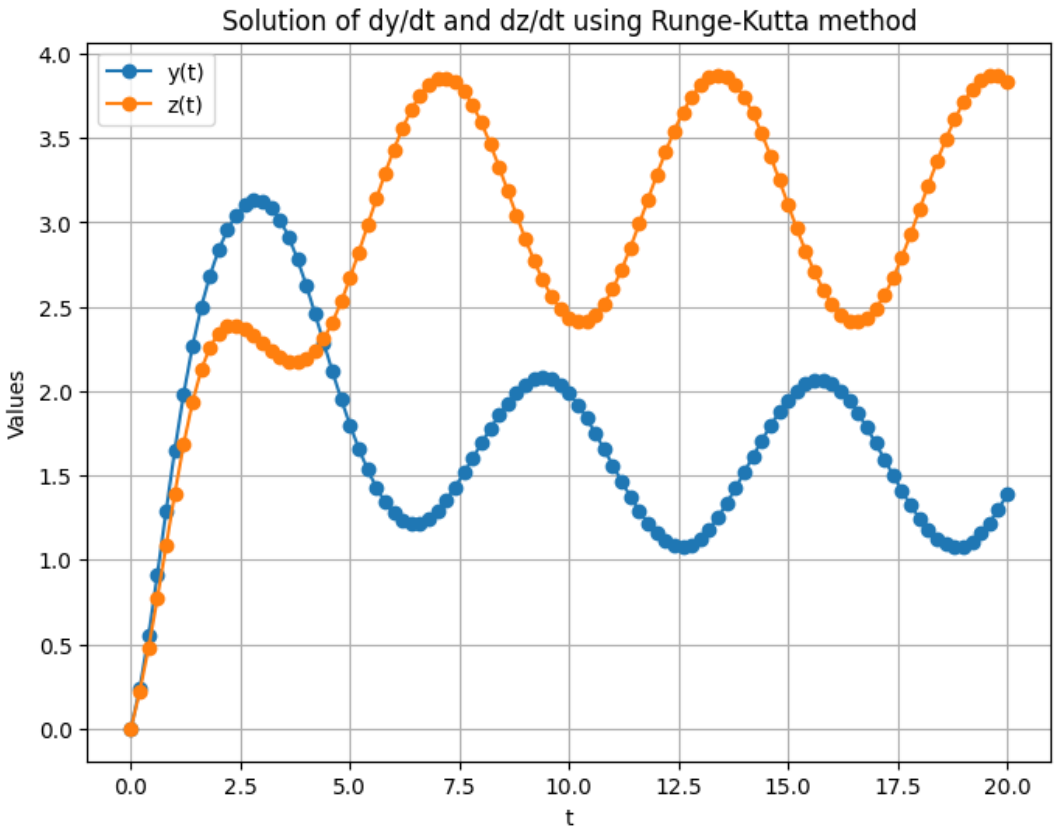


Figure 5: Plot for the solution of dy/dt and dz/dt using Runge-Kutta method.

From figure 5 we can see the approximated curves of the solution for y(t) and z(t). The approximated curves of y(t) is highlighted in blue while the approximated curve of z(t) is displayed in orange. Both of the functions display sort of a wavy pattern which is not that surprising given the nature of the equations. Since both equations can't be solve analytically, we can only see the behavior of the approximated curves as we change the parameters like the span of t and the step size.

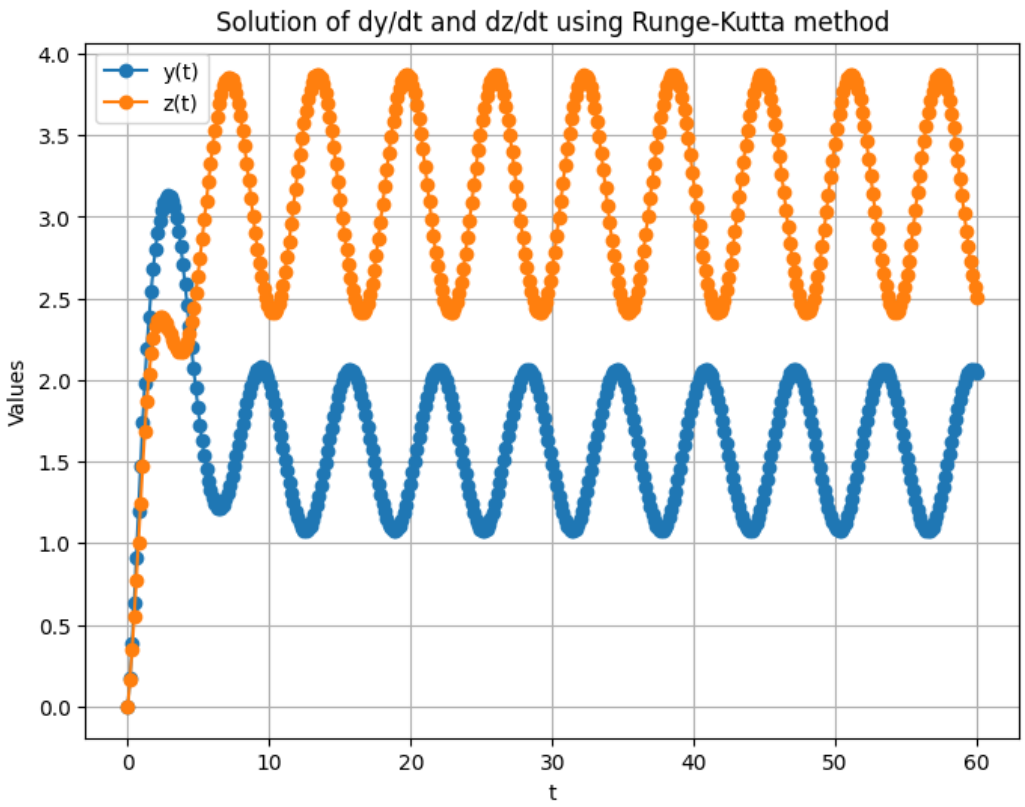


Figure 6: Plot for the solution of dy/dt and dz/dt using Runge-Kutta method with $t \in [0, 60]$ *and 400 intervals*

From figure 6 we can see the approximated curves of the solution for y(t) and z(t), with different set of parameters, $t \in [0, 60]$ with 400 intervals. Increasing the t span helps us observe the behavior described by the differential equation in a longer times span. As we can see the plot remains consistently the same throughout.

2.3 Problems encountered

The issue encountered in this problem is the same as the previous problem since we're not able to check the reliability of the approximated solutions computed by the Runge-Kutta method. This is because the differential equations given are not solvable analytically and therefore would have to rely on these approximations.

2.4 References

- ES_204_L6_Ordinary_Differential_Equations.pdf

PROBLEM 3 [40 points]: (a) A projectile is launched directly upwards with an initial velocity, $v_0$and its motion is governed solely by a uniform gravitational force

$$\ddot{y} = -9.8 \text{ m/s}^2.$$

After some time, the projectile is expected to reach a maximum elevation of 100 m. Determine v_0 by implementing an appropriate *shooting method* scheme. (b) However, it was found that the initial v_0 was not enough to reach the expected maximum elevation. Further analysis showed that a force due to drag contributed to the projectile's motion which can be represented as

$$\ddot{y} = -9.8 - \tfrac{1}{40}\dot{y} \text{ m/s}^2.$$

Recalculate v_0 and compare this with your previous calculations in part (a).

3.1 Method(s) of solution

We were tasked to determine initial velocity, v_0 that would make the projectile reach 100 meters using the shooting method scheme. Consider the projectile equations below:

$$y = v_0 - \frac{1}{2}gt^2 \quad (16)$$

$$\dot{y} = v_o - gt \quad (17)$$

$$\ddot{y} = -g \quad (18.a)$$

$$\ddot{y} = -g - \tfrac{1}{40}\dot{y} \quad (18.b)$$

We can emulate these equations with forward Euler method.

$$y_{t+1} = y_t + v_t \Delta t \quad (19)$$

$$v_{t+1} = v_t + \ddot{y}\Delta t \quad (20)$$

Let's consider a second-order ordinary differential equation (ODE) of the form:

$$\frac{d^2y}{dx^2} = f\left(x, y, \frac{dy}{dx}\right) \quad (21)$$

Subject to boundary conditions:

$$y(a) = \alpha \quad and \quad y(b) = \beta \qquad (22)$$

Choose “initial” conditions:

$$y(a) = \alpha \quad and \quad \frac{dy(a)}{dt} = \lambda \qquad (23)$$

Where the constant λ is constant so that we advance the solution to $t = b$ we find $y(b) = \beta$

The steps used to find the approximate solution are as follows:

1. Import numpy for matrix operations, matplotlib for plotting projectile motions in different initial velocities v_o with respect to time (t) and zip_longest to find the longest array of values which is also used for plotting projectile motion.
2. Define second order derivative of y. See *eqn. 18.a* and *eqn. 18.b*
3. Set parameters like target height, step size denoted as Δt, v_o lower and upper bound, and a specified tolerance for convergence
4. Use shooting method function along with forward Euler to output the computed initial velocity v_o that approximates $y = 100$ at $v_t = 0$ within a certain tolerance $tol = 0.1$.

```
function EULER METHOD
      initialize y as 0 and velocity as guessed initial_velocity
      initialize velocities, time, and heights array which will be used
for plotting
      dt = step size
      t = 0
      while loop
            v = vo - gdt # Euler method for velocity
            y = y + v*dt # Euler method for height
```

```
            append computed v, y, and t to the arrays of velocities,
heights, and time
            t = t + dt
            if v <= 0
                  end loop
      return y, velocities, time, and heights

function SHOOTING METHOD
      initialize vo mid as half of vo upper and lower bound
      initialize velocities, time, and heights as array of arrays.
      # Each array is the projectile motion at specific vo
      while loop
            y, velocities, time, and heights = EULER METHOD input vo
mid
            append velocities, time, and heights to the array
            if y > target height
                  v0 upper bound = v0 mid
            else if target height - tolerance < y < target height +
tolerance
                  end loop
            else
                  v0 lower bound = v0 mid
      return v0 mid, velocities, time, and heights
```

Figure 7: Pseudocode for Euler method and Shooting method scheme

5. Plot projectiles in different initial velocities v_o for visualization.

3.2 Results

a. For $\ddot{y} = -g$

```
Velocity within tolerance limit. Breaking...
The initial velocity required for a maximum elevation of 100
meters is approximately 44.3206787109375 m/s.
```

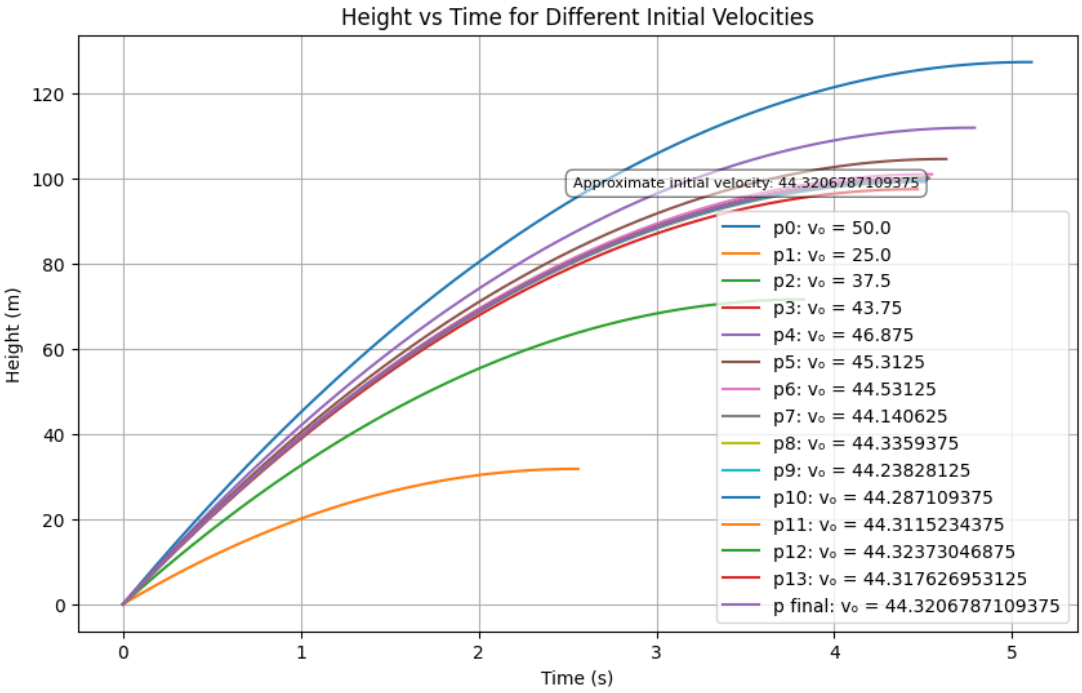


Figure 8: Projectile motion for different initial velocities v_o with respect to time (t) for $\ddot{y} = -g$.

b. For $\ddot{y} = -g - \frac{1}{40}\dot{y}$

```
Velocity within tolerance limit. Breaking...
The initial velocity required for a maximum elevation of 100
meters is approximately 46.0113525390625 m/s.
```

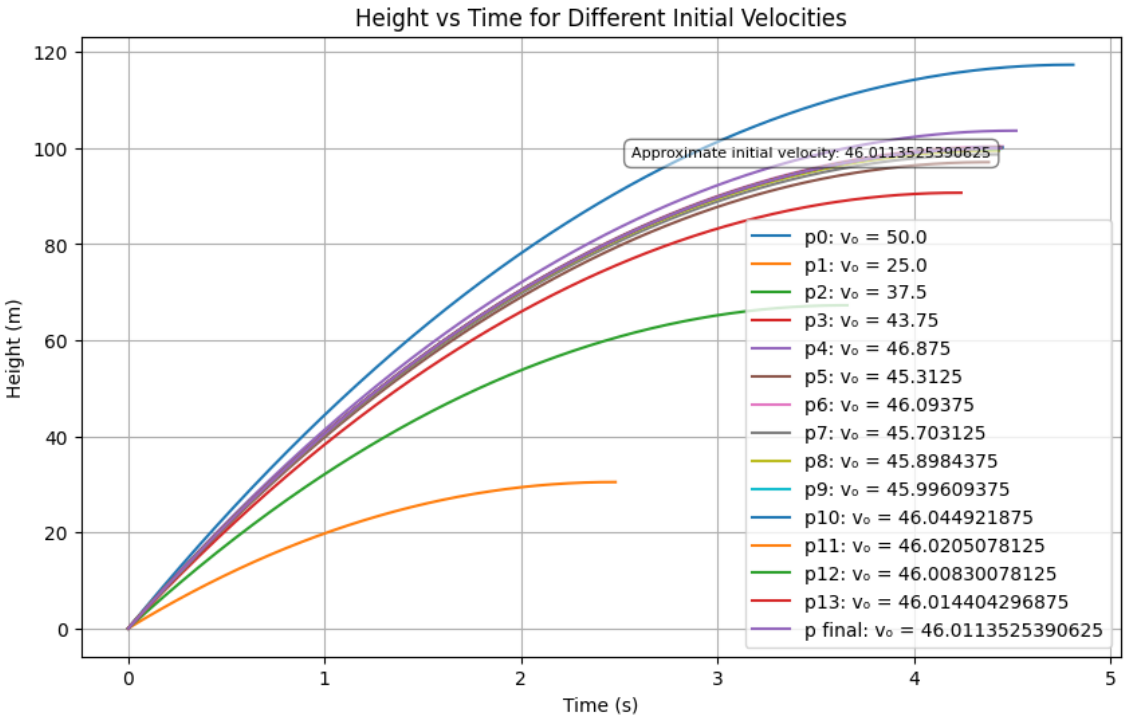


Figure 9: Projectile motion for different initial velocities v_o with respect to time (t) for $\ddot{y} = -g - 1/40\,\dot{y}$

From results section a, $\ddot{y} = -g$, we can see that approximated initial velocity is $v_o \approx 44.32\,m/s$. Figure 8 shows all projectile motions for different initial velocities that were tried by the shooting method scheme. The very first initial velocity guessed was $v_o = 50\,m/s$. As shown in the figure, this initial velocity overshoots the projectile to above 120 meters when velocity is zero. The shooting method then tries $v_o = 25\,m/s$ where the projectile vastly undershoots the target height. The shooting method scheme then increases iteratively the initial velocity until the target height reached approximately 100 meters in zero velocity within 0.1 tolerance. The results from section b, $\ddot{y} = -g - \frac{1}{40}\dot{y}$, shows a different solution with the approximated initial velocity being $v_o \approx 46.01\,m/s$. That makes sense that a higher initial velocity is needed since there is now air resistance with $-\frac{1}{40}\dot{y}$. Both results show that it took 14 iterations to converge to a final solution.

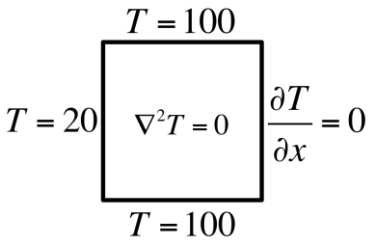
3.3 Problems encountered

The main challenge in this problem is we can't solve this problem analytically since the time (t) it takes for the projectile to reach a height (y) of 100 meters is not given. This is why we solve this numerically by iteratively solving velocity (v) and y by increasing t with a constant amount of step size. One thing I noticed is having a worse upper and lower boundary of v_o and smaller step size would increase the time it takes to find the solution. We also can't check the accuracy of the approximations just like the last problems but at least here we have a metric to see how close it is to the actual solution because of the target height and the desired tolerance limit.

3.4 References

- ES_204_L6_Ordinary_Differential_Equations.pdf

PROBLEM 4 [30 points]: Solve the *Laplace* equation $T(x, y)$ where $x, y \in [0,1]$ using the given boundary conditions.



Solve using 2×2, 4×4, 6×6 internal points.

4.1 Method(s) of solution

We were tasked to solve the Laplace equation using the given boundary conditions. For this case, we'll be using the Neumann Boundary condition. Consider the following figure:

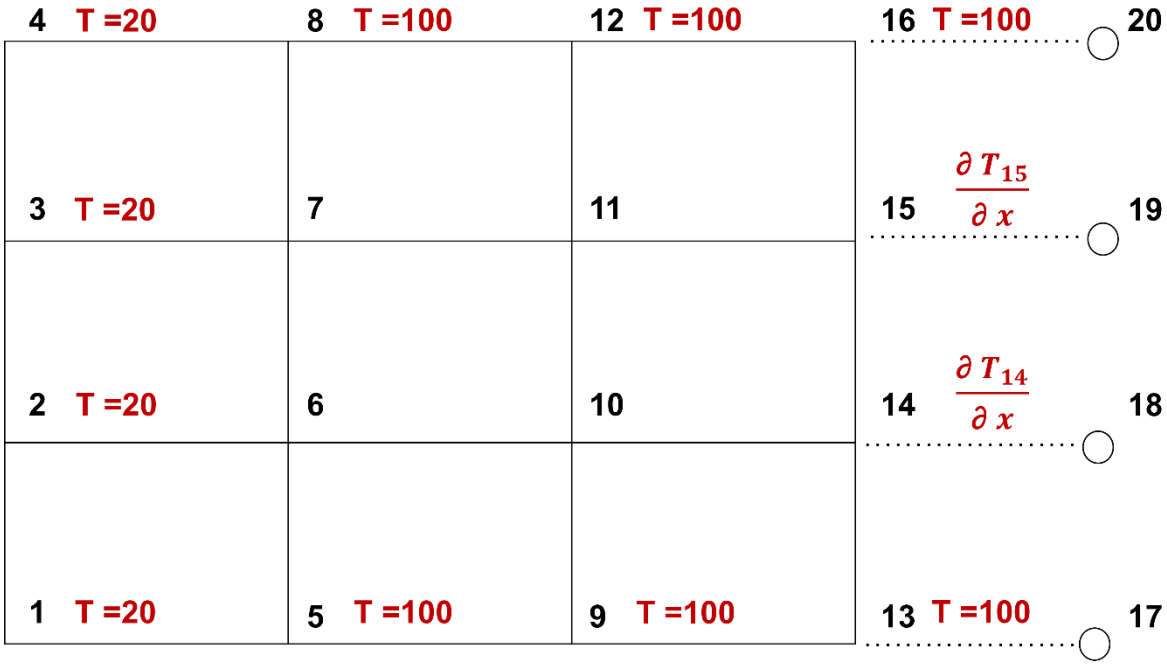


Figure 10: Decomposed domain with 2x2 internal points with ghost points on the right side

$$\begin{cases} \nabla^2 T(x,y) = 0 \ \ (0 < x < a, \ \ 0 < x < b) \\ T(0,y) = 20, \dfrac{\partial T(a,y)}{\partial x} = 0, T(x,0) = 100, T(x,b) = 100 \end{cases} \tag{24}$$

$$T_1, T_2, T_3, T_4 = 20$$

$$T_5, T_6, T_7, T_8 = 100$$

$$g_{13} = \frac{\partial T_{13}}{\partial x} = \frac{T_{17} - T_9}{2\Delta x}, \qquad T_{17} = T_9 \tag{25}$$

$$g_{14} = \frac{\partial T_{14}}{\partial x} = \frac{T_{18} - T_{10}}{2\Delta x}, \qquad T_{18} = T_{10} \tag{26}$$

$$g_{15} = \frac{\partial T_{15}}{\partial x} = \frac{T_{19} - T_{11}}{2\Delta x}, \qquad T_{19} = T_{11} \tag{27}$$

$$g_{16} = \frac{\partial T_{16}}{\partial x} = \frac{T_{20} - T_{12}}{2\Delta x}, \qquad T_{20} = T_{12} \tag{28}$$

$$T_{xx} + T_{yy} = 0 \tag{29}$$

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{h^2} \tag{30}$$

$$\frac{\partial^2 T}{\partial y^2} \approx \frac{T_{i,j-1} - 2T_{i,j} + T_{i,j-1}}{h^2} \tag{31}$$

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{32}$$

$$\frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{h^2} + \frac{T_{i,j-1} - 2T_{i,j} + T_{i,j+1}}{h^2} \tag{33}$$

$$4T_{i,j} - \left(T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1}\right) = 0 \tag{34}$$

$$
\begin{aligned}
4T_6 - T_7 - T_{10} &= T_2 + T_5 \\
-T_6 + 4T_7 - T_{11} &= T_3 + T_8 \\
-T_6 + 3T_{10} - T_{11} &= T_9 \\
-T_7 - T_{10} + 4T_{11} - T_{15} &= T_{12} \\
-2T_{10} + 4T_{14} - T_{15} &= T_9 \\
-2T_{11} - T_{14} + 4T_{15} &= T_{12}
\end{aligned}
\tag{35}
$$

$$
\begin{bmatrix}
4 & -1 & -1 & 0 & 0 & 0 \\
-1 & 4 & 0 & -1 & 0 & 0 \\
-1 & 0 & 3 & -1 & 0 & 0 \\
0 & -1 & -1 & 4 & 0 & -1 \\
0 & 0 & -2 & 0 & 4 & -1 \\
0 & 0 & 0 & -2 & -1 & 4
\end{bmatrix}
\begin{bmatrix}
T_6 \\ T_7 \\ T_{10} \\ T_{11} \\ T_{14} \\ T_{15}
\end{bmatrix}
=
\begin{bmatrix}
120 \\ 120 \\ 100 \\ 100 \\ 100 \\ 100
\end{bmatrix}
\tag{36}
$$

The steps used to find the approximate solution are as follows:

1. Import numpy for matrix operations, matplotlib for plotting temperature distribution for all grids.
2. Define parameters like the grid size (N), grid spacing (dx).
3. Initialize temperature grid, T and apply all boundary conditions that were given.
4. Initialize matrix A, b, and x. We want to be able to create $Ax = b$ similar to *eqn. 36.*
5. For all interior points, apply stencil. We use a different function for the Neumann boundaries. This will help us create matrix A and b. All of the exterior points in the equation were placed on matrix b. See *eqn. 35.* and *eqn.36.*

```
initialize N # grid size
initialize L # square domain
dx = L / N # grid spacing
```

```
initialize T # temperature grid
T[0, :] = 100 and T[-1, :] = 100
T[:, 0] = 20

initialize matrix A
initialize matrix b

function STENCIL METHOD
      for T[i-1, j], T[i+1, j], T[i, j-1], T[i, j+1] # surrounding
points of T[i, j]
            if exterior point
                  add value to matrix b
            else
                  add -1 to matrix A


function NEUMANN BOUNDARY STENCIL METHOD
      for T[i, j-1], T[i, j+1] # left and right neighbours of T[i, j]
            add -1 to matrix A
      for T[i-1, j], T[i+1, j] # above and bottom neighbours of T[i, j]
            add value to matrix b

x = BICONJUGATE GRADIENT STABILIZED METHOD input matrix A and b
update T # temperature grid
plot T
```

Figure 11: Pseudocode on solving Laplace equation with Neumann boundary conditions.

6. With matrix A and b, solve for matrix x using Biconjugate gradient stabilized. We'll be using this since matrix A is non-symmetric.
7. With the solved matrix x, update Temperature grid, T.

8. Plot temperature grid.

4.2 Results

a. 2x2 internal points

```
The resuling matrix is:
[[ 20.         100.         100.         100.        ]
 [ 20.          38.66666667  34.66666667  58.02074074]
 [ 20.          51.28888889  46.48888889  62.74962963]
 [ 20.         100.         100.         100.        ]]
```

Figure 12: Solved 2x2 temperature grid.

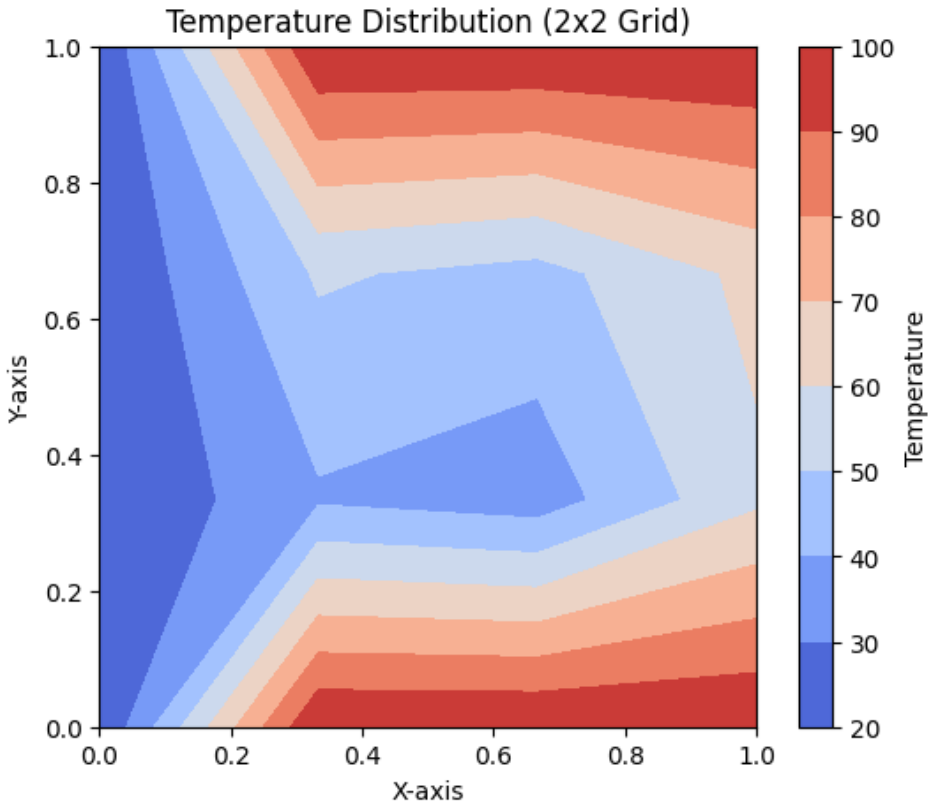


Figure 13: Temperature distribution plot for 2x2 grid

b. 4x4 internal points

```
[[ 20.    100.    100.    100.    100.    100.   ]
 [ 20.    33.331  37.172  34.589  27.331  51.086 ]
 [ 20.    16.664  23.078  21.81   15.48   36.097 ]
 [ 20.    21.056  13.324  13.553  9.322   23.597 ]
 [ 20.    39.43   47.559  48.257  36.653  38.639 ]
 [ 20.    100.    100.    100.    100.    100.   ]]
```

Figure 14: Solved 4x4 temperature grid.

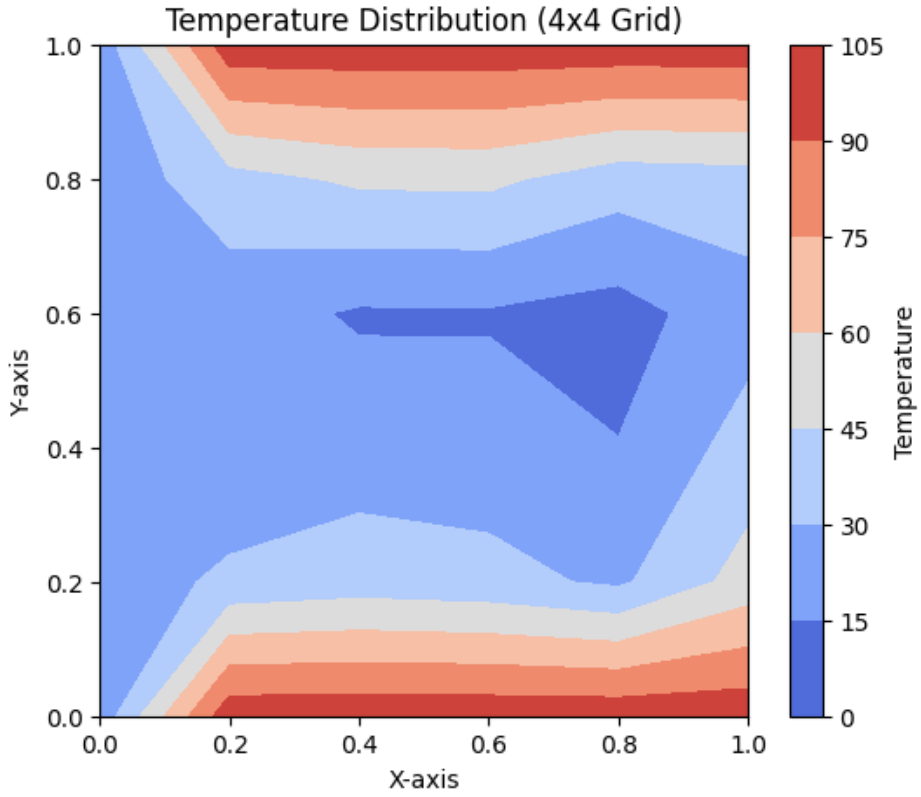


Figure 15: Temperature distribution plot for 4x4 grid

c. 6x6 internal points

```
[[ 20.    100.    100.    100.    100.    100.    100.    100.   ]
 [ 20.    39.958  55.921  53.599  47.837  36.865  27.907  53.858 ]
 [ 20.    24.947  35.554  34.408  29.451  20.305   9.885  30.727 ]
 [ 20.    26.439  34.717  34.401  30.084  21.231  11.63   29.949 ]
 [ 20.    27.836  43.74   46.463  41.53   31.502  14.595  30.142 ]
 [ 20.    24.806  39.831  42.504  38.465  29.762  11.632  19.455 ]
 [ 20.    43.16   60.809  64.73   61.122  52.986  36.633  34.806 ]
 [ 20.    100.    100.    100.    100.    100.    100.    100.   ]]
```

Figure 16: Solved 6x6 temperature grid.

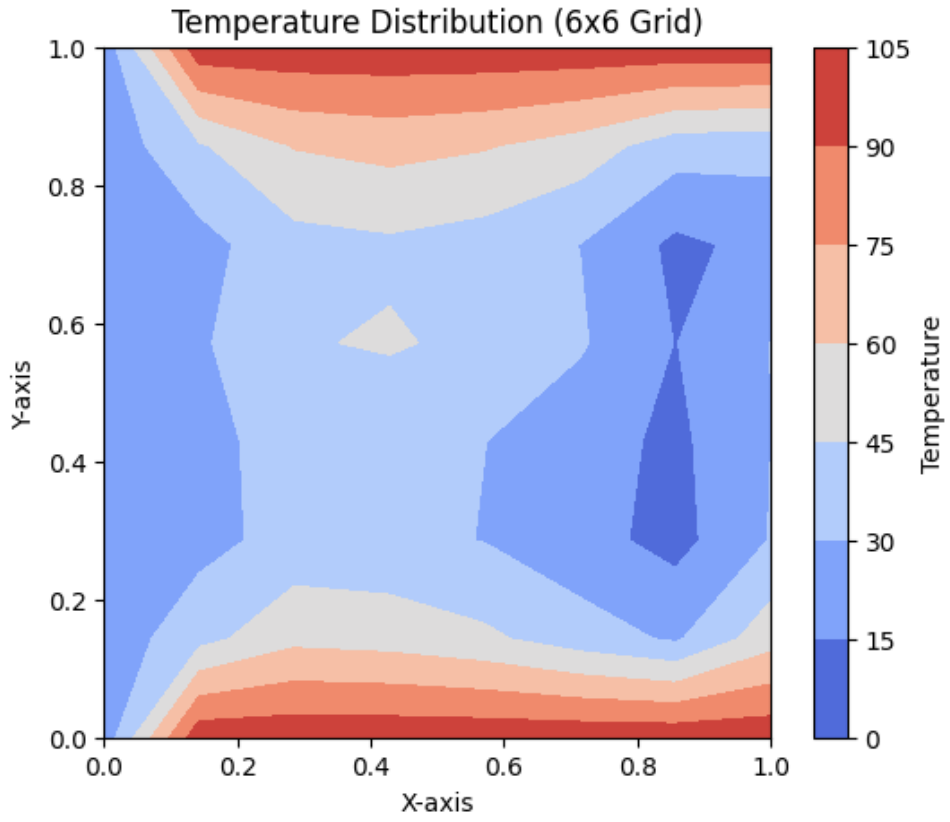


Figure 17: Solved 6x6 temperature grid.

The results were split to three sections for different internal point sizes, 2x2, 4x4, and 6x6 with the solved temperature grids in figures 12, 14, and 16 respectively. The boundary points on the right side are also considered as internal points due to the Neumann boundary condition on the right side. Figures 13, 15, and 17 shows the visual representation of these temperature distributions. As you can see the middle portions of the grid are the cooler parts of the grid but as you go near the upper and lower boundaries, the temperature goes higher given by the upper and lower boundary condition of 100. The right-side temperatures are varied but tend to be cooler on the middle portion and higher on the upper and lower portions.

4.3 Problems encountered

One of the challenges on this problem is applying the stencil method as an algorithm. I had to ensure that the computations for each element in matrix A is correct. I did the computations handwritten first before proceeding with the programming to ensure that the process is correct. The main challenge is applying the Neumann boundary condition. We had to add additional column called the “ghost points” to solve the boundary points on the right side of the domain. Another challenge is solving the $Ax = b$ since matrix A is non-symmetric and thus we cannot use the conjugate gradient method.

4.4 References

- Zeng, X. (n.d.). Lecture Note 7: Biconjugate Gradient Stabilized (BICGSTAB). MATH 5330: Computational Methods of Linear Algebra. Department of Mathematical Sciences, University of Texas at El Paso (UTEP). Retrieved from https://utminers.utep.edu/xzeng/2017spring_math5330/MATH_5330_Computational_Methods_of_Linear_Algebra_files/ln07.pdf
- Roth, J. (2020, September 15). Finite Differences [Video]. Numerical Analysis by Julian Roth. YouTube. https://www.youtube.com/watch?v=YotrBNLFen0&list=LL&index=2&t=136s
- Munna, S. (2022, September 28). Neumann Boundary Condition - PDE Solutions [Video]. YouTube. https://www.youtube.com/watch?v=uliD6MMbitQ
- ES_204_L7_Partial_Differential_Equations.pdf

APPENDIX

PROBLEM 1

SOURCE CODE:

```
import numpy as np
import matplotlib.pyplot as plt

# Heat loss function
def heat_loss(T):
    return -3e-11*(T**4 - 300**4)

# Equilibrium temperature
def equilibrium_temp():
    return (300 ** 4) ** (1/4)

# Forward Euler method
def forward_euler(T0, max_t, h):
    iter = int(max_t / h) + 1
    T_values = [T0]
    for _ in range(iter - 1):
        T_values.append(T_values[-1] + h * heat_loss(T_values[-1]))
    return T_values

# Backward Euler Method
def backward_euler(T0, max_t, h, tol=1e-6, max_iter=100):
    iter = int(max_t / h) + 1
    T_values = [T0]
    for _ in range(iter - 1):
        T_curr = T_values[-1]  # Initial T value as current
        for _ in range(max_iter):
            f = lambda T: T - T_curr - h * heat_loss(T)
            f_prime = lambda T: 1 - h * heat_loss(T)

            T_next = T_curr - f(T_curr) / f_prime(T_curr)
```

```
            if abs(T_next - T_curr) < tol:  # Convergence check
                break
            T_curr = T_next  # Update current estimate+
        T_values.append(T_next)
    return T_values


# parameters
max_t = 250
step_sizes = [0.005, 0.001, 0.05, 0.01, 1, 2, 5]
T0 = 1000
T_eq = equilibrium_temp()

for h in step_sizes:
    # For computing time points
    time_points = np.linspace(0, max_t, int(max_t / h) + 1)

    # Forward Euler method
    T_forward = forward_euler(T0, max_t, h)
    plt.plot(time_points, T_forward, label=f"Forward Euler (dt={h})", color='blue')

    # Backward Euler method
    T_backward = backward_euler(T0, max_t, h)
    plt.plot(time_points, T_backward, label=f"Backward Euler (dt={h})", color='red')

    # Adding annotations for Forward Euler
    idx_forward = int(len(time_points) * 0.7)
    plt.annotate(f"dt={h}", (time_points[idx_forward], T_forward[idx_forward]),
textcoords="offset points", xytext=(0, 10),
                 ha='center', color='white', fontsize=8, bbox=dict(facecolor='black',
alpha=0.7))

    # Adding annotations for Backward Euler
    idx_backward = int(len(time_points) * 0.3)
    plt.annotate(f"dt={h}", (time_points[idx_backward], T_backward[idx_backward]),
textcoords="offset points", xytext=(0, 10),
```

```
                ha='center', color='white', fontsize=8, bbox=dict(facecolor='black',
alpha=0.7))

# Plotting the equilibrium temperature
time_points = np.linspace(0, max_t, int(max_t) + 1)
T_eq_values = np.full_like(time_points, T_eq)
plt.plot(time_points, T_eq_values, label="Equilibrium Temperature", color='green')


plt.xlabel('Time (seconds)')
plt.ylabel('Temperature (K)')
plt.title('Forward and Backward Euler Methods for Cooling')
plt.show()
```

SOLUTION:

$$T_{i+1} = T_i + \Delta t\, f(t_i, T_i) \quad (\Delta t = h) \tag{1}$$

$$T_{i+1} = T_i + \Delta t\, f(t_{i+1}, T_{i+h}) \quad (\Delta t = h) \tag{2}$$

$$f(T_{i+1}) = T_i - T_{i+1} + \Delta t\, f(t_{i+1}, T_{i+h}) \tag{3}$$

$$T_{i+1}^{(k+1)} = T_{i+1}^{(k)} - \frac{f(T_{i+1}^{(k)})}{f'(T_{i+1}^{(k)})} \tag{4}$$

$$0 = -3 \times 10^{-11}(T^4 - 300^4)$$

$$T = 300, \qquad where\ \frac{dT}{dt} = 0$$

PROBLEM 2

SOURCE CODE:

```
import numpy as np
import matplotlib.pyplot as plt

# Define function y
def y_function(t, y, z):
    return np.sin(t) + np.cos(y) + np.sin(z)

# Define function z
def z_function(t, z):
    return np.cos(t) + np.sin(z)

# Fourth-order Runge-Kutta method
def runge_kutta(funcy, funcz, t_span, y0, z0, num_intervals):
    h = (t_span[1] - t_span[0]) / num_intervals
    t_values = np.linspace(t_span[0], t_span[1], num_intervals + 1)
    y_values = [y0]
    z_values = [z0]

    for i in range(num_intervals):
        k1_y = h * funcy(t_values[i], y_values[i], z_values[i])
        k1_z = h * funcz(t_values[i], z_values[i])

        k2_y = h * funcy(t_values[i] + (h/2), y_values[i] + k1_y / 2, z_values[i] +
k1_z / 2)
        k2_z = h * funcz(t_values[i] + (h/2), z_values[i] + k1_z / 2)

        k3_y = h * funcy(t_values[i] + h / 2, y_values[i] + k2_y / 2, z_values[i] +
k2_z / 2)
        k3_z = h * funcz(t_values[i] + h / 2,  z_values[i] + k2_z / 2)

        k4_y = h * funcy(t_values[i] + h, y_values[i] + k3_y, z_values[i] + k3_z)
        k4_z = h * funcz(t_values[i] + h, z_values[i] + k3_z)
```

```
        y_values.append(y_values[i] + (k1_y + 2 * k2_y + 2 * k3_y + k4_y) / 6)
        z_values.append(z_values[i] + (k1_z + 2 * k2_z + 2 * k3_z + k4_z) / 6)

    return t_values, y_values, z_values


# parameters
t_span = [0, 20]
num_intervals = 100
y0 = 0  # Initial value for y
z0 = 0  # Initial value for z

# Solve using Fourth-order Runge-Kutta method
t_values, y_values, z_values = runge_kutta(y_function, z_function, t_span, y0, z0,
num_intervals)

plt.figure(figsize=(8, 6))

# Plot numerical approximations for both function y and z
plt.plot(t_values, y_values, label='y(t)', marker='o', linestyle='-')
plt.plot(t_values, z_values, label='z(t)', marker='o', linestyle='-')

plt.xlabel('t')
plt.ylabel('Values')
plt.title('Solution of dy/dt and dz/dt using Runge-Kutta method')
plt.legend()
plt.grid(True)
plt.show()
```

SOLUTION:

$$k_1^y = h \cdot funcy(t_n, y_n, z_n) \quad (5)$$

$$k_1^z = h \cdot funcz(t_n, z_n) \quad (6)$$

$$k_2^y = h \cdot funcy\left(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}, z_n + \frac{k_1}{2}\right) \quad (7)$$

$$k_2^z = h \cdot funcz\left(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}, z_n + \frac{k_1}{2}\right) \quad (8)$$

$$k_3^y = h \cdot funcy\left(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}, z_n + \frac{k_2}{2}\right) \quad (9)$$

$$k_3^z = h \cdot funcz(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}, z_n + \frac{k_2}{2}) \quad (10)$$

$$k_4^y = h \cdot funcy(t_n + h, y_n + k_3, z_n + k_3)) \quad (11)$$

$$k_4^z = h \cdot funcz(t_n + h, y_n + k_3) \quad (12)$$

$$y_{n+1}^y = y_n^y + \frac{k_1^y}{6} + \frac{k_2^y}{3} + \frac{k_3^y}{3} + \frac{k_4^y}{6} + o(h^5) \quad (13)$$

$$y_{n+1}^z = y_n^z + \frac{k_1^z}{6} + \frac{k_2^z}{3} + \frac{k_3^z}{3} + \frac{k_4^z}{6} + o(h^5) \quad (14)$$

$$h = \frac{t[1] - t[0]}{n} \quad (15)$$

PROBLEM 3

SOURCE CODE:

```
import numpy as np
import matplotlib.pyplot as plt


from itertools import zip_longest

def d2y_dx2():
    return -9.8

def euler_method(dt, v0, g):
    y = 0 # initialize at y = 0
    vy = v0 # initialize velocity
    velocities = [v0]
    heights = [y]
    time = [0]

    t = 0
```

```
    while True:  # Simulate the projectile's motion
        vy = vy +  g * dt  # Update velocity using Euler's method
        y = y + vy * dt  # Update height using Euler's method
        time.append(t + dt)
        velocities.append(vy)
        heights.append(y)
        t += dt
        if vy <= 0:
            break    # Stop the simulation at the maximum height which is if
velocity is zero (vy = 0)

    return y, velocities, heights, time # Return maximum height is reached


def shooting_method(target_height, dt, v0_lower, v0_upper, tolerance):
    v0_mid = 0.5*(v0_lower + v0_upper)
    v_values = []
    y_values = []
    t_values = []
    vo_values = []

    while True:
        y, v_value, y_value, t_value = euler_method(dt, v0_mid, d2y_dx2())
        v_values.append(v_value)
        y_values.append(y_value)
        t_values.append(t_value)
        vo_values.append(v0_mid)
        if y > target_height:
            v0_upper = v0_mid
        elif target_height - tolerance <=  y <= target_height + tolerance:
            print("Velocity within tolerance limit. Breaking...")
            break
        else:
            v0_lower = v0_mid
        v0_mid = 0.5 * (v0_lower + v0_upper)

    return v0_mid, v_values, y_values, t_values, vo_values
```

```
def plot_projectile_simulation(v, y, t, vo):
    # Plotting
    plt.figure(figsize=(10, 6))
    i = 0
    for v_val, y_val, t_val, vo_val in zip_longest(v, y, t, vo, fillvalue=[]):
        if i == len(v)-1:
            plt.plot(t_val, y_val, label=f"p final: v₀ = {vo_val}")
            plt.annotate(f"Approximate initial velocity: {vo_val}", (t_val[-
1], y_val[-1]), textcoords="offset points", xytext=(-5,-5), ha='right',
fontsize=8, bbox=dict(facecolor='white', alpha=0.5, edgecolor='black',
boxstyle='round,pad=0.5'))
        else:
            plt.plot(t_val, y_val, label=f"p{i}: v₀ = {vo_val}")

        i += 1



    plt.title('Height vs Time for Different Initial Velocities')
    plt.xlabel('Time (s)')
    plt.ylabel('Height (m)')
    plt.legend()
    plt.grid(True)
    plt.show()

target_height = 100  # Target maximum elevation
dt = 0.01  # Step size for Euler's method
v0_lower_bound = 0  # Lower bound for initial velocity
v0_upper_bound = 100  # Upper bound for initial velocity
tolerance = 0.01  # Tolerance for the shooting method

initial_velocity, v_values, y_values, t_values, vo_values =
shooting_method(target_height, dt, v0_lower_bound, v0_upper_bound, tolerance)
plot_projectile_simulation(v_values, y_values, t_values, vo_values)

print(f"The initial velocity required for a maximum elevation of 100 meters is
approximately {initial_velocity} m/s.")
```

SOLUTION:

$$y = v_0 - \frac{1}{2}gt^2 \quad (16)$$

$$\dot{y} = v_o - gt \quad (17)$$

$$\ddot{y} = -g \quad (18.a)$$

$$\ddot{y} = -g - \frac{1}{40}\dot{y} \quad (18.b)$$

$$y_{t+1} = y_t + v_t \Delta t \quad (19)$$

$$v_{t+1} = v_t + \ddot{y}\Delta t \quad (20)$$

$$\frac{d^2y}{dx^2} = f\left(x, y, \frac{dy}{dx}\right) \quad (21)$$

$$y(a) = \alpha \quad and \quad y(b) = \beta \quad (22)$$

$$y(a) = \alpha \quad and \quad \frac{dy(a)}{dt} = \lambda \quad (23)$$

PROBLEM 4

SOURCE CODE:

```
import numpy as np
import matplotlib.pyplot as plt

# Constants
N = 5  # Grid size (3x3 grid: 2 internal points along x and y)
L = 1.0  # Length of the square domain

dx = L / N  # Grid spacing

# Initialize temperature grid
T = np.zeros((N+1, N+1))  # Initialize temperature grid
```

```
# boundary conditions
T[:, 0] = 20.0
T[0, :] = 100.0
T[-1, :] = 100.0
T[0, 0] = 20.0
T[-1, 0] = 20.0
for i in range(1, N):
    T[i, -1] = 1

print(T)
np.set_printoptions(precision=3, suppress=True, linewidth=100)

A = np.zeros(((N-1)**2 + (N-1), (N-1)**2 + (N-1)))
b = np.zeros(((N-1)**2 + (N-1), 1))


def stencil_method(i, j, k):
    A[k, k] = 4

    n = 0

    if T[i-1, j] != 0:
        b[k, 0] += T[i-1, j]
    else:
        n += 1
        A[k, k+n] = -1

    if T[i+1, j] != 0:
        b[k, 0] += T[i+1, j]
    else:
        n += 1
        A[k+n*-1, k] = -1

    if T[i, j-1] != 0:
        b[k, 0] += T[i, j-1]
    else:
        n += 1
```

```
        A[k, k+n*-1] = -1


    if T[i, j+1] != 0:
        if T[i, j+1] != 1:
            b[k, 0] += T[i, j+1]
    else:
        n += 1
        A[k, k+n] = -1

    return None

def neumann_boundary_stencil_method(i, j, k):
    A[k, k] = 4
    n = 0

    if T[i-1, j] != 1:
        b[k, 0] += T[i-1, j]
    else:
        n += 1
        A[k, k+n] = -1

    if T[i+1, j] != 1:
        b[k, 0] += T[i+1, j]
    else:
        n += 1
        A[k, k+n*-1] = -1

    if T[i, j-1] != 0:
        b[k, 0] += T[i, j-1]
    else:
        n += 1
        A[k, k+n*-1] += -2

    return None
```

```
k = 0
for j in range(N):
    for i in range(N-1, 0, -1):
        if k < (N-1)**2:
            stencil_method(i, j+1, k)
            print(A)
        else:
            neumann_boundary_stencil_method(i, j+1, k)
        k += 1



# Test BiCGStab method with your matrices
x_0 = np.ones_like(b)
print(A)
print(b)

def BICGSTAB(A, B, x_0, max_iterations=10000, tolerance=1e-8):
    x = x_0
    r = B - np.dot(A, x)
    p = r
    r_tilde = r.copy()
    rsold = np.dot(r.T, r).item()

    for i in range(max_iterations):
        Ap = np.dot(A, p)
        alpha = rsold / np.dot(r_tilde.T, Ap).item()
        x = x + alpha * p
        r = r - alpha * Ap
        s = r.copy()

        rsnew = np.dot(r.T, r).item()
        if np.sqrt(rsnew) < tolerance:
            print(f"Convergence achieved in {i+1} iterations using BiCGSTAB.")
            return x

        beta = (rsnew / rsold) * (alpha / np.dot(r_tilde.T, s).item())
        r_tilde = r - beta * (r_tilde - np.dot(A, s))
```

```
        p = r_tilde + beta * p
        rsold = rsnew

    print("Maximum iterations reached without convergence.")
    return x

x = BICGSTAB(A, b, x_0)
print(x)

k = 0
for j in range(N):
    for i in range(N-1, 0, -1):
        T[i, j+1] = x[k]
        k += 1


print("The resuling matrix is: ")
print(T)

# Visualize the temperature distribution
x = np.linspace(0, L, N+1)
y = np.linspace(0, L, N+1)
X, Y = np.meshgrid(x, y)

plt.figure(figsize=(6, 5))
plt.contourf(X, Y, T, cmap='coolwarm')
plt.colorbar(label='Temperature')
plt.title(f'Temperature Distribution ({N-1}x{N-1} Grid)')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.grid(False)
plt.show()
```

SOLUTION:

$$\begin{cases} \nabla^2 T(x,y) = 0 \ \ (0 < x < a, \ \ 0 < x < b) \\ T(0,y) = 20, \dfrac{\partial T(a,y)}{\partial x} = 0, T(x,0) = 100, T(x,b) = 100 \end{cases} \tag{24}$$

$$T_1, T_2, T_3, T_4 = 20$$
$$T_5, T_6, T_7, T_8 = 100$$

$$g_{13} = \frac{\partial T_{13}}{\partial x} = \frac{T_{17} - T_9}{2\Delta x}, \qquad T_{17} = T_9 \tag{25}$$

$$g_{14} = \frac{\partial T_{14}}{\partial x} = \frac{T_{18} - T_{10}}{2\Delta x}, \qquad T_{18} = T_{10} \tag{26}$$

$$g_{15} = \frac{\partial T_{15}}{\partial x} = \frac{T_{19} - T_{11}}{2\Delta x}, \qquad T_{19} = T_{11} \tag{27}$$

$$g_{16} = \frac{\partial T_{16}}{\partial x} = \frac{T_{20} - T_{12}}{2\Delta x}, \qquad T_{20} = T_{12} \tag{28}$$

$$T_{xx} + T_{yy} = 0 \tag{29}$$

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{h^2} \tag{30}$$

$$\frac{\partial^2 T}{\partial y^2} \approx \frac{T_{i,j-1} - 2T_{i,j} + T_{i,j-1}}{h^2} \tag{31}$$

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{32}$$

$$\frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{h^2} + \frac{T_{i,j-1} - 2T_{i,j} + T_{i,j+1}}{h^2} \tag{33}$$

$$4T_{i,j} - \left(T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1}\right) = 0 \tag{34}$$

$$
\begin{aligned}
4T_6 - T_7 - T_{10} &= T_2 + T_5 \\
-T_6 + 4T_7 - T_{11} &= T_3 + T_8 \\
-T_6 + 3T_{10} - T_{11} &= T_9 \\
-T_7 - T_{10} + 4T_{11} - T_{15} &= T_{12} \\
-2T_{10} + 4T_{14} - T_{15} &= T_9 \\
-2T_{11} - T_{14} + 4T_{15} &= T_{12}
\end{aligned}
\tag{35}
$$

$$
\begin{bmatrix}
4 & -1 & -1 & 0 & 0 & 0 \\
-1 & 4 & 0 & -1 & 0 & 0 \\
-1 & 0 & 3 & -1 & 0 & 0 \\
0 & -1 & -1 & 4 & 0 & -1 \\
0 & 0 & -2 & 0 & 4 & -1 \\
0 & 0 & 0 & -2 & -1 & 4
\end{bmatrix}
\begin{bmatrix}
T_6 \\ T_7 \\ T_{10} \\ T_{11} \\ T_{14} \\ T_{15}
\end{bmatrix}
=
\begin{bmatrix}
120 \\ 120 \\ 100 \\ 100 \\ 100 \\ 100
\end{bmatrix}
\tag{36}
$$