Assignment

Week Assignment 9

Name TN 1: Tim Peinkofer

tim.peinkofer@student.uni-tuebingen.de

Name TN 2: Fabian Kostow

fabian.kostow@student.uni-tuebingen.de

Tutor: Jose Carlos Olvera Meneses

Date: January 28, 2025

Contents

1 Problem I

In the first step we need to develop the shooting method based on our Runge-Kutta 4th order and linear interpolation functions we created a few weeks ago. The code is at follows:

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

# Define the ODE system
def ode(x, y):
    # We have defined the ODE the following: y[0] = u, y[1] =
                                    u'
    dy0 = y[1]
    dy1 = 2*y[0]
    return np.array([dy0, dy1])

# Linear interpolation for root finding (based on the method
                               we discussed in class)
def linear_interpol(f, a, b, tol=1e-6, max_iter=1000):
    """Finds the root of f in [a, b] using linear
                                   interpolation."""
    if f(a) * f(b) > 0:
        raise ValueError("No root in interval")

    iterations = []  # To store intermediate solutions (c
                                   values)

    for i in range(max_iter):  # Maximum 1000 iterations
        print(f"Iteration {i+1}: a = {a}, b = {b}")

        if abs(f(b) - f(a)) < 1e-12:  # Avoid division by
                                       zero
            raise ValueError("Function values too close;
                                           division by zero."
                                           )

        c = a - f(a) * (b - a) / (f(b) - f(a))

        iterations.append(c)  # Save the current value of c

        if abs(f(c)) < tol:  # Tolerance for convergence
            print(f"Root found after {i + 1} iterations.")
            break
        if f(c) * f(a) < 0:
            b = c
        else:
            a = c
    else:
        print("Maximum iterations reached.")

    for idx, v in enumerate(iterations):
        y_0 = [u_1, v]
        t_values, y_values = runge_kutta(ode, y_0, x_1, x_2,
                                       n)
        plt.plot(t_values, y_values[:, 0], label=f"Iteration
                                       {idx + 1}")

    plt.xlabel("x")
    plt.ylabel("u(x)")
    plt.title("Solutions at Each Iteration of Root Finder")
    plt.legend()
    plt.show()
    return c

# 4th Order Runge-Kutta method
def runge_kutta(f, y_0, t_start, t_end, n):
    """Solves an ODE using the 4th-order Runge-Kutta method.
                                   """
    h = (t_end - t_start) / n  # Step size
    t_values = np.linspace(t_start, t_end, n + 1)
    y_values = np.zeros((n + 1, len(y_0)))
    y_values[0] = y_0

    for i in range(n):
        t_i = t_values[i]
        y_i = y_values[i]

        k1 = h * f(t_i, y_i)
        k2 = h * f(t_i + 0.5 * h, y_i + 0.5 * k1)
        k3 = h * f(t_i + 0.5 * h, y_i + 0.5 * k2)
        k4 = h * f(t_i + h, y_i + k3)

        y_values[i + 1] = y_i + (1 / 6) * (k1 + 2 * k2 + 2 *
                                       k3 + k4)

    return t_values, y_values
```

```python
# Shooting method function
def solve_by_shooting(ode, x_1, x_2, n, v_0, u_1, u_2):

    def difference(v):
        y_0 = [u_1, v]
        _, y_values = runge_kutta(ode, y_0, x_1, x_2, n)
        return y_values[-1, 0] - u_2

    # Debug: Check function values at endpoints
    print(f"difference({v_0[0]}) = {difference(v_0[0])}")
    print(f"difference({v_0[1]}) = {difference(v_0[1])}")

    v_corr = linear_interpol(difference, v_0[0], v_0[1])

    if v_corr is None:
        raise RuntimeError("Root finding failed.")

    y_0 = [u_1, v_corr]
    x, y = runge_kutta(ode, y_0, x_1, x_2, n)
    return v_corr, x, y

# Parameters
x_1, x_2 = 0, 1
n = 1000
u_1, u_2 = 1.2,0.9
v_0 = [-10, 10]

# Solve the BVP
v, x, y = solve_by_shooting(ode, x_1, x_2, n, v_0, u_1, u_2)

print(f"Corrected initial slope u'(x1) = {v}")
print(f"Solution at x2: {y[-1, 0]}")
print(f'Error for x2: {y[-1, 0] - u_2}')

# Plot the final solution
plt.plot(x, y[:, 0], label="Final Solution (u(x))", color="
                               blue")
plt.xlabel("x")
plt.ylabel("u(x)")
plt.title("Solution of the BVP using Shooting Method")
plt.legend()
plt.grid()
plt.show()
```

We can now use this code to solve the following linear boundary value problem:

$$
\begin{aligned}
u'' &= \left(1 - \frac{x}{5}\right) u + x \\
u(1) &= 2 \\
u(3) &= -1
\end{aligned}
$$

As a result we get after one iteration for the inital guess v = [-10,10] :

$$u'(1) \approx -3.4949$$

But why do we have just one iteration? The reason is that we have a linear problem which leads to a fast convergence of our method.

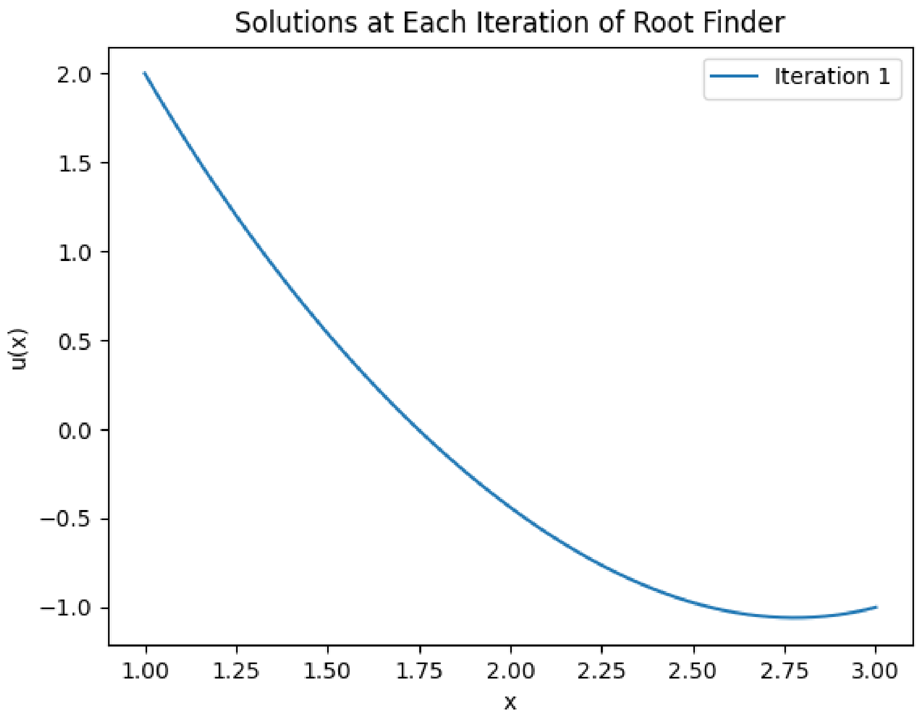

Figure 1: Result for every iteration

In the next step we want to do the same thing again with a nonlinear problem given by:

$$u'' = \left(1 - \frac{x}{5}\right) uu' + x$$
$$u(1) = 2$$
$$u(3) = -1$$

As a result we get after 12 iterations (accuracy 10^-4) for the initial guess v = [-3,-1]:

$$u'(1) \approx -2.0161$$

We can now plot the solution for every iterations:

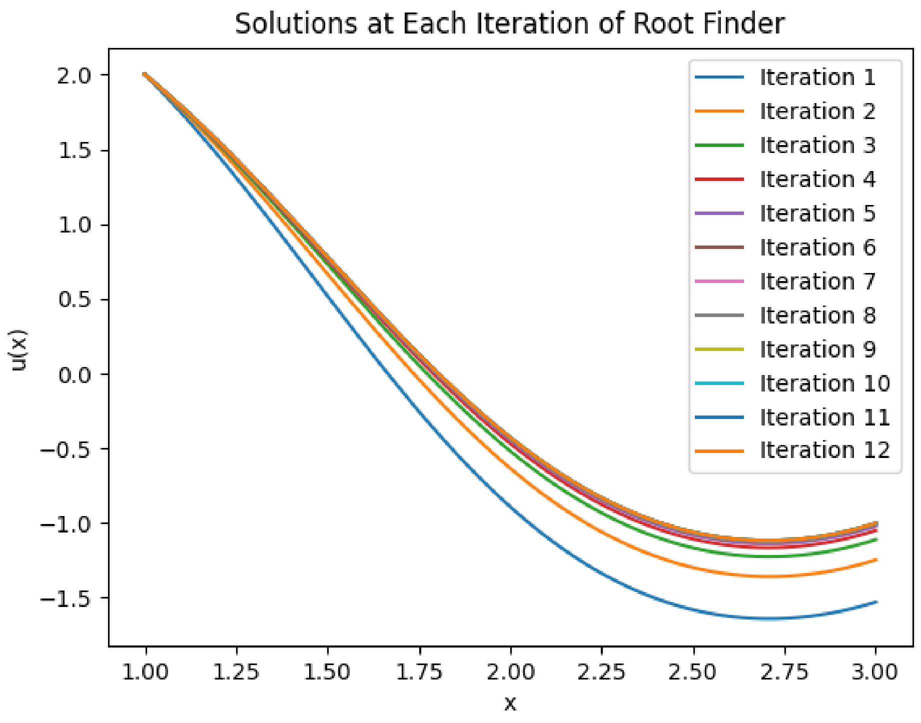

Figure 2: Result for every iteration

And also the result:

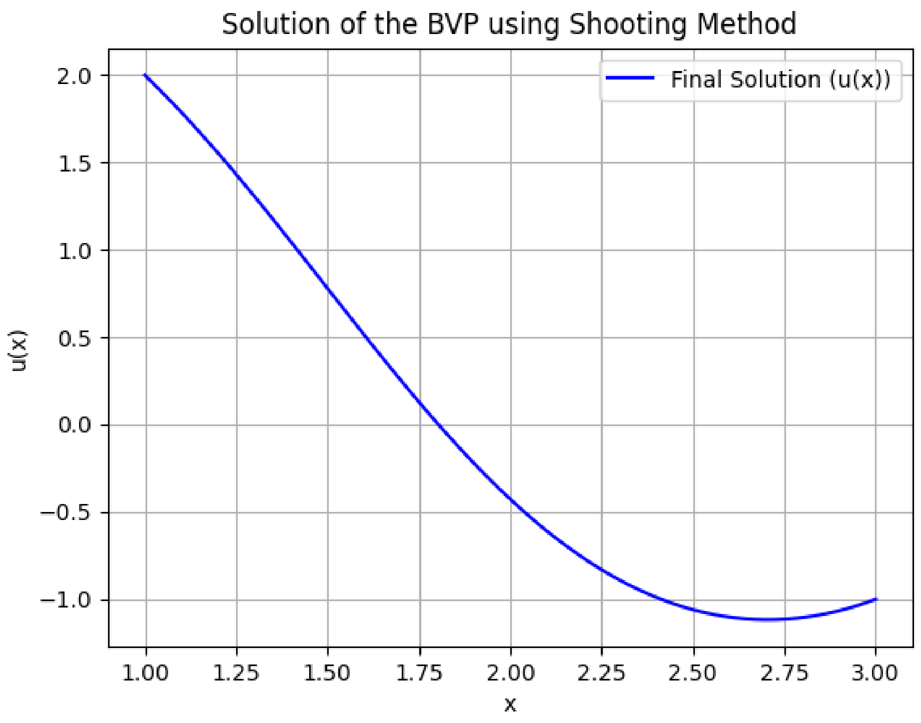

Figure 3: Result of the nonlinear ODE

2 Problem II

We now want to solve the following problem using the shooting method:

$$\frac{d^2y}{dx^2} = 2y$$
$$y(0) = 1.2$$
$$y(1) = 0.9$$

Using the code from Question I we get the following result for the initial guess v = [-10,10] after one iteration:

$$y'(0) = -1.2525$$

The solution looks the following:

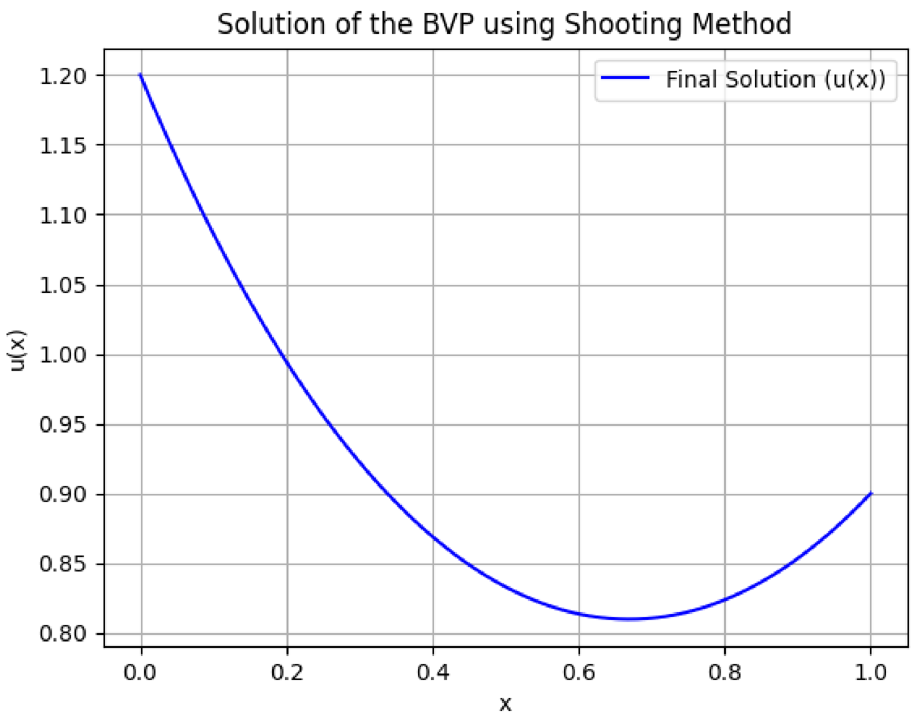

Figure 4: Solution of Problem II

3 Problem III

We want to solve the problem above again

$$\begin{aligned}\frac{d^2y}{dx^2} &= 2y \\ y(0) &= 1.2 \\ y(1) &= 0.9\end{aligned}$$

this time we want to use matrix method.

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

def Matrix(h, n):
    diag_main = -(2 + 2 * h**2) * np.ones(n)
    diag_upper = np.ones(n - 1)
    diag_lower = np.ones(n - 1)
    A = np.diag(diag_main) + np.diag(diag_upper, 1) + np.diag
                                    (diag_lower, -1)
    return A

def Vector(n, y0, yn):
    b = np.zeros(n)
    b[0] -= y0
    b[-1] -= yn
    return b

n = 100
x0, xn = 0, 1
y0, yn = 1.2, 0.9
h = (xn - x0) / (n + 1)

list_x = np.linspace(x0, xn, n + 2)
list_y = np.zeros(n + 2)
list_y[0] = y0
list_y[-1] = yn

A = Matrix(h, n)
b = Vector(n, y0, yn)
y = np.linalg.solve(A, b)

for i in range(1, n + 1):
    list_y[i] = y[i - 1]
```

```
# Plot solution
plt.figure(figsize=(8, 6))
plt.plot(list_x, list_y, color='b')
plt.title('')
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True)
plt.legend()
plt.show()
```

If we plot the solution we get:

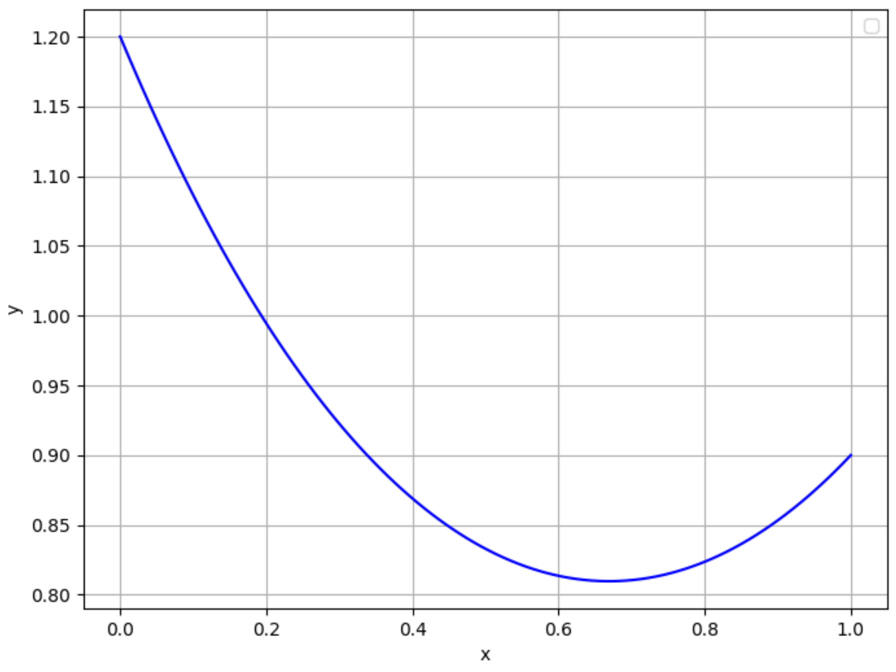

Figure 5: Solution for Problem III

Now we want to compare the matrix method with the shooting method:

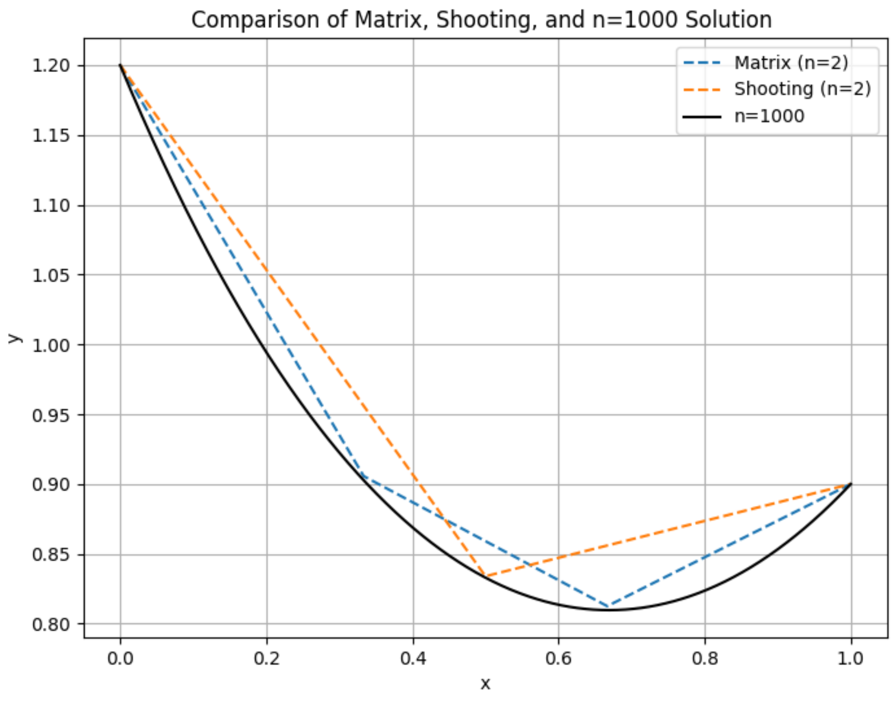

Figure 6: comparison for n=2

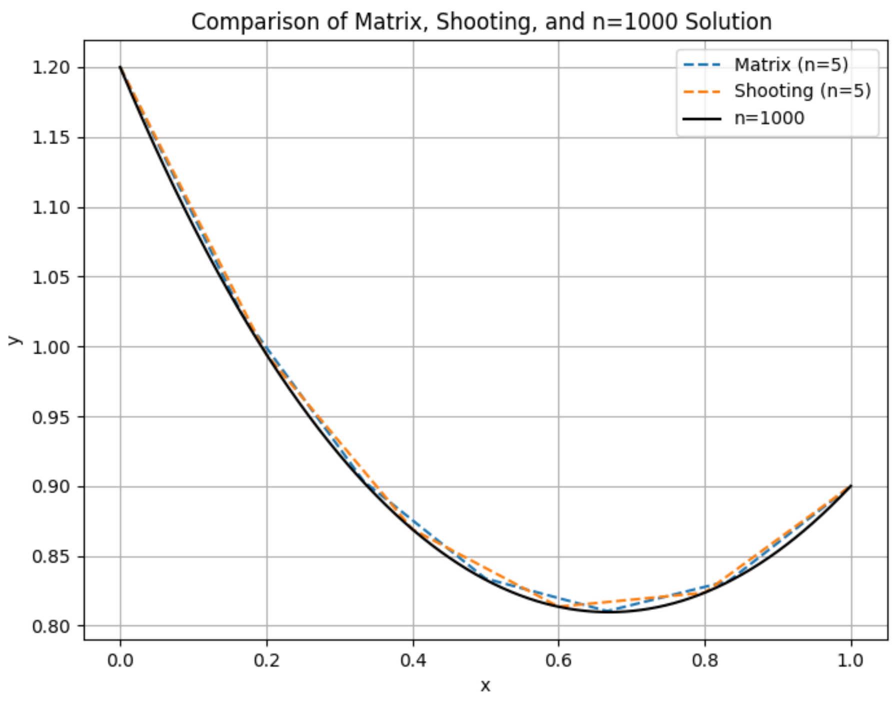

Figure 7: comparison for n=5

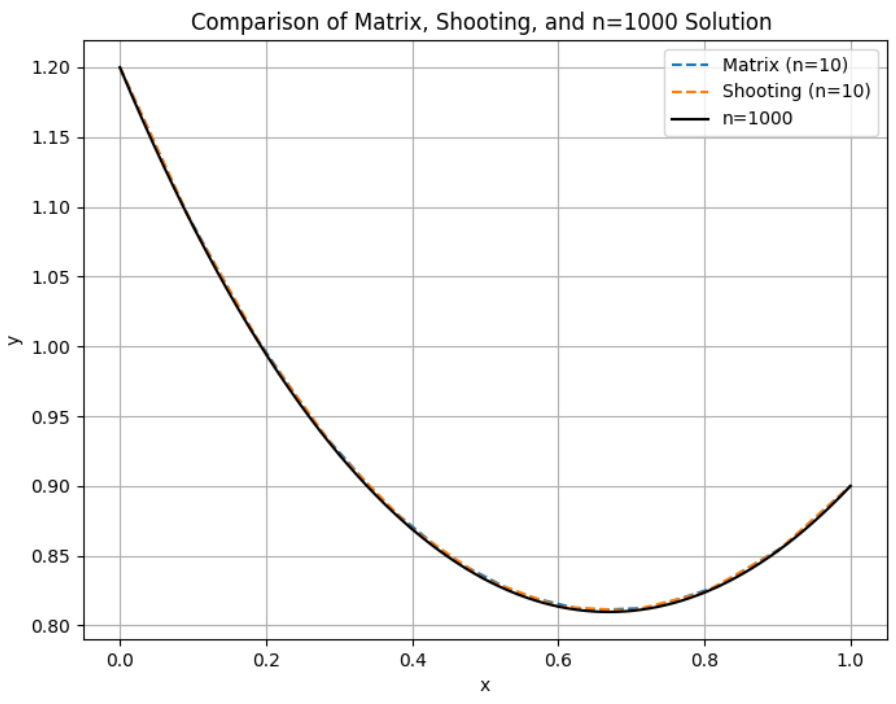

Figure 8: comparison for n=10

We see that for small n, matrix method is slightly better but both methods converge relatively fast.

4 Problem IV

We now want to solve the following problem using the shooting method:

$$\begin{aligned} \frac{d^2y}{dx^2} &= 1 - \frac{(2+y^2)y}{1+y^2} \\ y(0) &= 0 \\ y(2) &= 3 \end{aligned}$$

Using the code from Question I we get the following result for the initial guess v = [-5,5] after four iterations:

$$y'(0) \approx 2.33$$

We can also plot the solution:

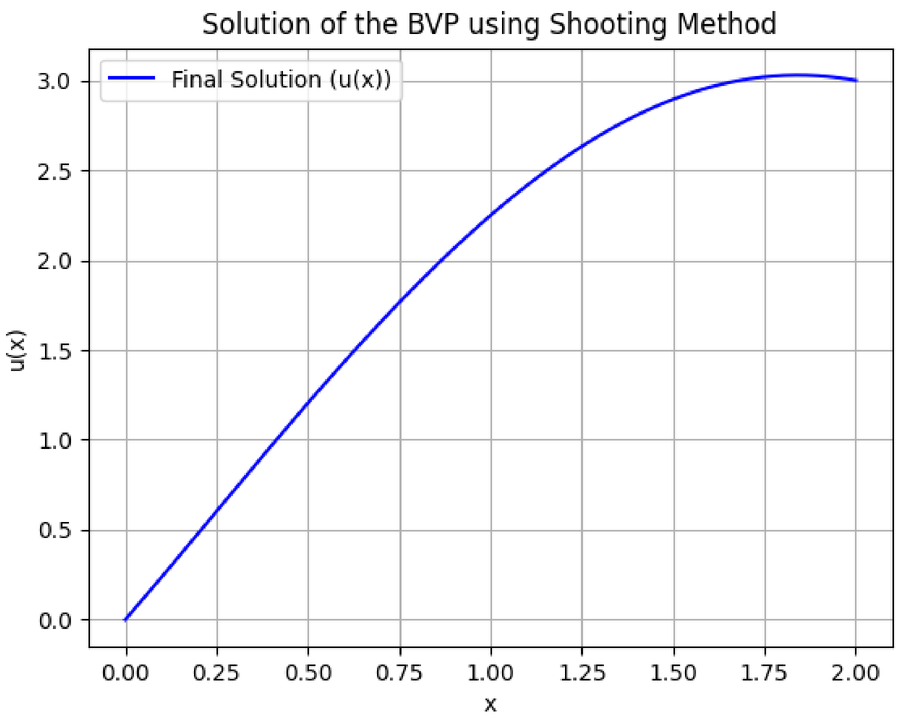

Figure 9: Solution for Problem IV