Numerical Analysis I - Homework 1

Fall Term 2023-2024

Istanbul Technical University

Student: Dilan Kilic
Student ID: 511232119
Department: Aeronautical and Astronautical Engineering Dept.
Due: Dec 08, 2023 @Ninova 23:30

Since I could not upload the file with an extension of '.zip', I uploaded all the codes for this homework in this GitHub repository URL.

Question-1

Will the following iterations converge to the indicated fixed point β where x_0 is sufficiently close to β? If it converges, give the order of convergence; give the rate of linear convergence for linear convergence.

a)

$$x_{n+1} = -22 + 7x_n + \frac{12}{x_n} \qquad \text{for } \beta = 3$$

b)

$$x_{n+1} = \frac{2}{x_n + 1} \qquad \text{for } \beta = 1$$

Solution: Q1

In this problem, we first need to be sure whether the given β point is a fixed point. Then, we can use the derivative information at the same point to determine the order of convergence. The formula that we have for this problem can be given as follows

$$|x_{n+1} - \beta| \leq C|x_n - \beta|^p \tag{1}$$

Here, $p = 1$ shows the linear convergence whereas $p = 2$ indicates the quadratic convergence. Let us find the function and its derivates at this point, respectively,

a)

$$\begin{aligned}
f(x) &= -22 + 7x + \frac{12}{x} \\
f(3) &= -22 + 7 * 3 + \frac{12}{3} = 3 \quad (\text{ so } \beta \text{ is a fixed point.}) \\
f'(x) &= 7 - \frac{12}{x^2} \\
f'(3) &= 7 - \frac{12}{3^2} = 5.66667
\end{aligned}$$

Since $|f'(3)| \geq 1$, it **cannot** be guaranteed convergence for a given x_0 which is sufficiently close to β.

b)

$$\begin{aligned}
f(x) &= \frac{2}{x+1} \\
f(1) &= -\frac{2}{1+1} = 1 \quad (\text{ so } \beta \text{ is a fixed point.}) \\
f'(x) &= -\frac{2}{(x+1)^2} \\
f'(1) &= -\frac{2}{(x+1)^2} = -0.5000
\end{aligned}$$

Since $|f'(1)| \leq 1$ and $|f'(1)| \neq 0$, it can **linearly** convergence for a given x_0 which is sufficiently close to β with $p = 1$. The convergence rate is $C \approx 0.5$.

Question-2

Use the Newton-Fourier method to solve the following equation by using an error tolerance of $\epsilon = 10^{-7}$.

$$x^2 + x + 1 = x^3$$

Use computer for your calculations. Show your work clearly. Your answer should include

1. Matlab / Python / C / C++ code
2. Output
3. Interpretation / discussion

Solution: Q2

The Newton-Raphson method is referred to as one of the most commonly used techniques for finding the roots of given equations. It is based on the idea of using the tangent line to approximate the behavior of the function near a root.

Let $f(x)$ be a differentiable function, and x_0 is an initial guess for a root of $f(x)$. The iterative formula can be given as follows:

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

Here $f'(x)$ is the first derivative of $f(x)$ with respect to x. Using this information, we can extend the idea with the assumption of $f(a) < 0$, $f(b) > 0$, and the root lies in this interval. If we modify the Newton-Raphson method using the following two steps, which is called the **Newton-Fourier** method, we obtain:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{3}$$

$$z_{n+1} = z_n - \frac{f(z_n)}{f'(x_n)} \tag{4}$$

$$\tag{5}$$

Here, the initial points for the steps: $x_0 = b$ and $z_0 = a$. For this problem, Python code is provided with a class of $NewtonRaphsonFourier()$. The user can different one-variable functions and the derivative with an initial guess to make a root estimation. Provided Jupyter [1] Notebook file in Appendix 1.1 contains the source code. For this problem, the initial guess point for the Newton-Raphson is $x_0 = 0.5$ and the interval for the Newton-Fourier is determined as $[a, b] = [0.5, 2.0]$. With these values, one can obtain the iteration number for the methods, respectively, **22** and **9** with an error tolerance of $\epsilon = 10^{-7}$. Besides, the Newton-Raphson code is also validated with the results of the built-in library (scipy, see Appendix 1.1) and the results are in good agreement in terms of iteration number and the resulting root value.

Starting Point	Method	Iteration No	Result
0.5	Newton-Raphson	22	1.8392868
$[0.5, 2.0]$	Newton-Fourier	9	1.8392868
0.5	Built-in Function	22	1.8392868
2.5	Newton-Raphson	7 ↑	1.8392868
$[0.5, 2.0]$	Newton-Fourier	9	1.8392868
2.5	Built-in Function	6 ↑	1.8392868

Table 1: Comparison of Methods with Shared Value

If we investigate the results from Table 1, we run the code for two different starting points (left-sided and right-sided) whereas the interval for the Newton-Fourier is kept constant. For all the cases and methods, we obtain the same result of **1.8392868**. For the Newton-Raphson method, we can converge to the root more quickly when we start from the right side (closer to the root). Therefore, even if the results are the same, the convergence iteration highly depends on the starting point or estimated interval. As a result, we may use the Newton-Fourier method as an alternative to the Newton-Raphson, and we can check which method gives less number of iterations for a certain problem. We can also say that the convergence problem may occur in Newton-Raphson when we select a starting point far away from the root. Furthermore, for the Newton-Fourier method, the convergence problem may be related to interval selection.

Question-3

Find an upper bound for the error on $[-\pi/2, 2\pi]$ when the degree 5 Chebyshev interpolating polynomial is used to approximate $f(x) = sin(x)$.

Solution: Q3

Given a function f on the interval $[-1, +1]$ and $x_1, x_2, \cdot, x_n$ are the data points, the interpolation polynomial is that unique polynomial S_{n-1} with a degree of at most $n-1$. The interpolation error at x is

$$f(x) - S_{n-1}(x) = \frac{f^n(\zeta)}{n!}\prod_{i=1}^{n}(x - x_i) \tag{6}$$

for some ζ in the interval $[-1, +1]$. To minimize the error, we should think of the maximum value of the product term. In Chebyshev polynomials, this bound is 2^{1-n} and for the scaled version, the bound is $2^{1-n}T_n$. Therefore, the equality in Equation for the scaled version can be rewritten as:

$$\begin{aligned} f(x) - S_{n-1}(x) &= \frac{(\frac{b-a}{2})^n}{n!2^{n-1}}|f^n(\zeta)| \\ sinx - S_5(x) &= \frac{(\frac{2\pi+\pi/2}{2})^6}{6!2^{6-1}}|f^6(-\pi/2)| \\ &= \mathbf{0.159176} \end{aligned}$$

Here, the sixth derivative of the function is $-sinx$, and the maximum value is achieved at $-sin(x = -\pi/2) = 1$. Therefore, the fifth degree of interpolation function for $sinx(x)$ has an error of **0.159176**.

Question-4

Find and plot the cubic spline S satisfying $S(0) = 1, S(1) = 3, S(2) = 1, S(3) = 4, S(4) = 2$ and with $S'(0) = 2$ and $S'(4) = -1$.

Solution: Q4

Let $x_0, x_1, x_2, \cdots x_n$ be the data points and $y_0, y_1, y_2, \cdots y_n$ be the corresponding function values. The cubic interpolation polynomial for each two-point interval can be defined as, in general,

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + a_j(x - x_j)^3$$

for each $j = 0, 1, \cdots n-1$.

In this problem, first, Algorithm 3.4 (Natural Cubic Spline) and Algorithm 3.5 (Clamped Cubic Spline) from the textbook [2] are implemented. The source code, attached in Appendix 1.2 for two methods is validated with Example 1 and 2 from Section 3.4 in the textbook (see Table 4 and 5 in Appendix 1.2). The validated results with the textbook for both methods are in good agreement. Then, we investigate the given homework problem with natural and clamped spline methods. We have 5 data points, so we expect 4 splines to be created for the given dataset.

Spline, j	a_j	b_j	c_j	d_j
0	1.	3.51785714	0.	-1.51785714
1	3.	-1.03571429	-4.55357143	3.58928571
2	1.	0.625	6.21428571	-3.83928571
3	4.	1.53571429	-5.30357143	1.76785714
4	**2.**			

Table 2: Homework Problem (Natural) Spline Coefficient Results

For natural spline, the computed piecewise function coefficients are listed in Table 2, and the corresponding interpolation plot is depicted in Figure 1. Furthermore, for clamped spline, the calculated piecewise spline coefficients and plotting splines are illustrated in Table 3 and Figure 2, respectively. From Figure 1 and 2, we can clearly see that there is a difference at the tails of the function due to the derivative information.

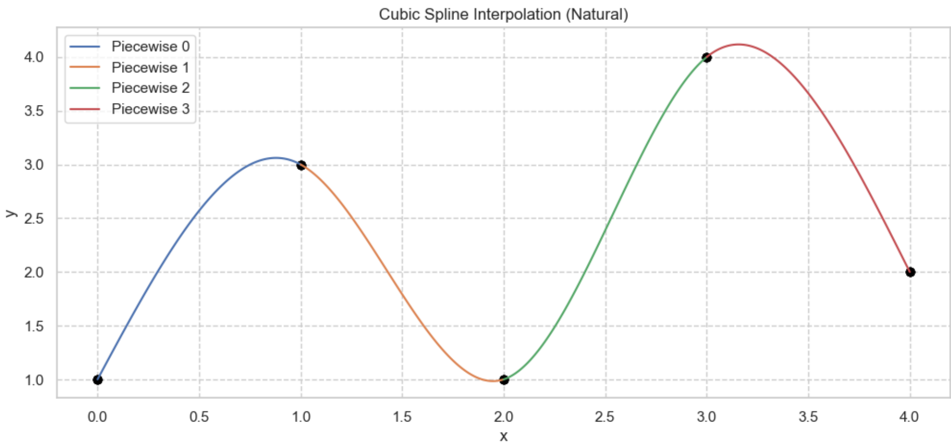

Figure 1: Homework Problem (Natural) Spline

Spline, j	a_j	b_j	c_j	d_j
0	1.	2.	2.67857143	-2.67857143
1	3.	-0.67857143	-5.35714286	4.03571429
2	1.	0.71428571	6.75	-4.46428571
3	4.	0.82142857	-6.64285714	3.82142857
4	**2.**			

Table 3: Homework Problem (Clamped) Spline Coefficient Results

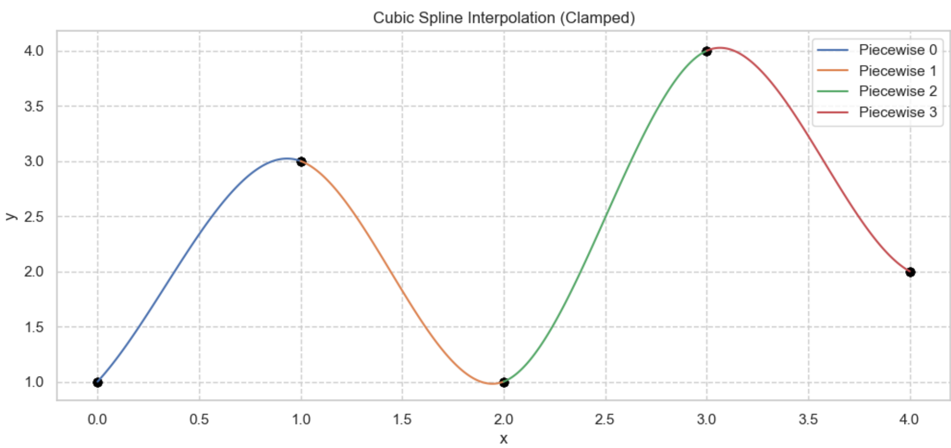

Figure 2: Homework Problem (Clamped) Spline

We can also calculate the integral approximation for the natural and clamped spline methods for the homework problem as follows:

$$\begin{aligned}\int_0^4 S(x)dx &= (a_0+a_1+a_2+a_3)+\frac{1}{2}(b_0+b_1+b_2+b_3)\\ &+\frac{1}{3}(c_0+c_1+c_2+c_3)++\frac{1}{4}(d_0+d_1+d_2+d_3)\\ (Natural) &= 10.107143\\ (Clamped) &= 9.7500000\end{aligned}$$

References

[1] K. Thomas, R.-K. Benjamin, P. Fernando, G. Brian, B. Matthias, F. Jonathan, K. Kyle, H. Jessica, G. Jason, C. Sylvain, and et al., “Jupyter notebooks - a publishing format for reproducible computational workflows,” *Stand Alone*, vol. 0, no. Positioning and Power in Academic Publishing: Players, Agents and Agendas, p. 87–90, 2016.

[2] R. L. Burden and J. D. Faires, *Numerical Analysis*. The Prindle, Weber and Schmidt Series in Mathematics, Boston: PWS-Kent Publishing Company, fourth ed., 1989.

Appendix

1.1 Question-2

```python
# import libraries
import numpy as np
import imageio
import matplotlib.pyplot as plt
import os


'''
NewtonRaphsonFourier: This class contains 4 main functions.
1) equation()       : Returns the function value for a given x point.
2) derivative()     : Returns the derivative function value for a given x point.
3) newton_fourier() : The main function for Newton-Raphson and Newton-Fourier Methods.
    - Provide 1) float initial guess and 2) fourier=False  ==> Newton-Raphson
    - Provide 1) 2-element list initial guess and 2) fourier=True  ==> Newton-Fourier
4) plot_animation() : Plots and saves the convergence animation.

'''
class NewtonRaphsonFourier():
    def __init__(self):
        pass

    def equation(self,x):
        return x**2 + x + 1 - x**3

    def derivative(self,x):
        return 2*x + 1 - 3*x**2

    def newton_fourier(self,initial_guess, tolerance=1e-7, max_iterations=100,fourier=False):

        # Check the type of initial guess interval
        if isinstance(initial_guess,list) and fourier is False:
            TypeError('For the Newton-Fourier Method, an initial float value should be defined.')
        else:
            if fourier is False:
                ##################################################
                # Newton-Raphson Method
                ##################################################
```

```python

            print('>>>>>>>> Newton-Raphson Method Running...')
            # Collect the x values
            x_values = [initial_guess]
            # Make the initial guess the first x value
            x = initial_guess
            # Screen-print for the first iteration
            print(f'Iteration No: {1:5d}, solution: {x:.5f}, error: -')

            for iteration in range(1,max_iterations):
                # Calculate the function value at a certain x point
                f_x = self.equation(x)
                # Calculate the derivative function value at a certain x point
                f_prime_x = self.derivative(x)

                if abs(f_prime_x) < tolerance:
                    print("Derivative is close to zero. Newton-Raphson method failed.")
                    return None

                # Calculate the new iteration point
                delta_x = -f_x / f_prime_x
                x = x + delta_x
                x_values.append(x)

                # Screen-print for each iteration
                print(f'Iteration No: {iteration + 1:5d}, solution: {x:.5f}, error: {abs(delta_x):.5f}')

                # Check the convergence
                if abs(delta_x) < tolerance:
                    print(f">>>>>> Converged in {iteration + 1} iterations with the convergence rate {abs(delta_x):.5f}.")
                    return x, x_values

        else:
            #################################################
            # Newton-Fourier Method
            #################################################

            print('>>>>>>>> Newton-Fourier Method Running...')
            # Check the type of initial guess interval
            if isinstance(initial_guess,list):
                # Define a and b interval values
                a = initial_guess[0]
                b = initial_guess[1]
                # Collect the x and z values
                x_values = [b]
                z_values = [a]
                # Make the initial guess the first x and z values
                x = b
                z = a
                # Screen-print for the first iteration
                print(f'Iteration No: {1:5d}, solution: {x:.5f}, error: -')

                for iteration in range(1,max_iterations):
                    # Calculate the function value at a certain x point and z point
                    f_x = self.equation(x)
                    f_z = self.equation(z)
                    # Calculate the derivative function value at a certain x point
                    f_prime_x = self.derivative(x)

                    if abs(f_prime_x) < tolerance:
                        print("Derivative is close to zero. Newton-Raphson method part failed.")
```

```
                            return None

                        # Calculate the new iteration point
                        delta_x = -f_x / f_prime_x
                        x = x + delta_x
                        x_values.append(x)

                        # Calculate the new iteration point
                        delta_z = -f_z / f_prime_x
                        z = z + delta_z
                        z_values.append(z)

                        # Screen-print for each iteration
                        print(f'Iteration No: {iteration + 1:5d}, solution: {x:.5f}, error: {abs(delta_z):.5f}')

                        # Check the convergence
                        if abs(delta_z) < tolerance:
                            print(f">>>>>> Converged in {iteration + 1} iterations with the convergence rate {abs(delta_z):.5f}.")
                            return z, z_values

    def plot_animation(self,x_values,plot_name='Raphson'):
        # Initialize an empty list to collect the figures
        images = []

        # Create fig folder
        try:
            os.makedirs('fig')
        except:
            pass

        for i in range(len(x_values)):
            fig, ax = plt.subplots(figsize=(7,5), constrained_layout=True)
            x_plot = np.linspace(min(x_values) - 1, max(x_values) + 1, 1000)
            y_plot = self.equation(x_plot)

            ax.plot(x_plot, y_plot, color='blue', label='Function')

            ax.scatter(x_values[i], self.equation(x_values[i]), color='red', label='Root Estimation', facecolor=None)
            if i == len(x_values)-1:
                pass
            else:
                # Plot the line between the two points
                plt.plot([x_values[i], x_values[i+1]], [self.equation(x_values[i]), 0], color='black',linewidth=0.8)

            ax.legend()
            plt.grid(which = "major", linewidth = 1)
            plt.grid(which = "minor", linewidth = 0.5,linestyle=':')
            plt.minorticks_on()
            plt.xlabel('$x$')
            plt.ylabel('$f(x)$')
            plt.title(f'Newton-{plot_name} Method')
            plt.savefig(f'fig/Newton_{plot_name}_iter{i}.png', transparent = False, facecolor = 'white')
            plt.close()

            images.append(imageio.v2.imread(f'fig/Newton_{plot_name}_iter{i}.png'))

        # Save the list of images as a GIF
        imageio.mimsave(f'animation_{plot_name}.gif', # output gif
                    images,          # array of input frames
```

```python
                     duration = 500, loop = 0)         # optional: frames per second


# ------------------------------------------------------------ USER-DEFINED INPUTS ------------------------------------------------------------ #
# call main class
nrf = NewtonRaphsonFourier()

# ----------------------------------------------------------- NEWTON-RAPHSON METHOD ----------------------------------------------------------- #
# Initial guess
initial_guess = 0.5 # or 2.5

# Solve the equation using Newton-Fourier method and get the values for visualization
solution_raphson, x_values = nrf.newton_fourier(initial_guess=initial_guess,fourier=False)

# Plot the animation
if solution_raphson is not None:
    print("Solution: %.7f" % solution_raphson)
    # Visualize the Newton-Raphson method as an animation
    nrf.plot_animation(x_values,plot_name='Raphson')

# ----------------------------------------------------------- NEWTON-FOURIER METHOD ----------------------------------------------------------- #
# Initial guess
initial_guess = [0.5,2.0]

# Solve the equation using Newton-Fourier method and get the values for visualization
solution_fourier, z_values = nrf.newton_fourier(initial_guess=initial_guess,fourier=True)

# Plot the animation
if solution_fourier is not None:
    print("Solution: %.7f" % solution_fourier)
    # Visualize the Newton-Fourier method as an animation
    nrf.plot_animation(z_values,plot_name='Fourier')


# Check the results from Scipy Library
from scipy import optimize
root= optimize.newton(nrf.equation, 0.5, fprime=nrf.derivative, full_output=True)
print(root)

# Check the results from Scipy Library
from scipy import optimize
root= optimize.newton(nrf.equation, 2.5, fprime=nrf.derivative, full_output=True)
print(root)
```

1.2 Question-4

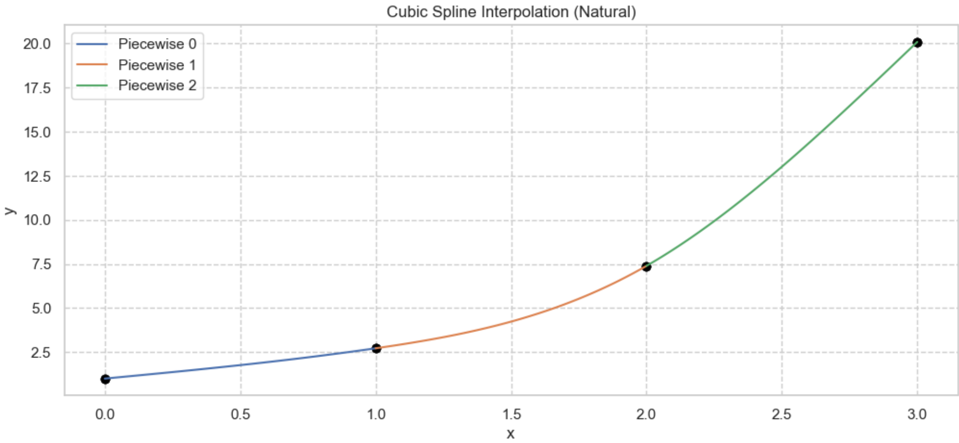

Figure 3: Example 1 (Natural) Spline (Validation)

Spline, j	a_j	b_j	c_j	d_j
0	1.	1.46599761	0.	0.25228421
1	2.71828183	2.22285026	0.75685264	1.69107137
2	7.3890561	8.80976965	5.83006675	-1.94335558
3	**20.08553692**			

Table 4: Example 1 (Natural) Spline Coefficient Results (Validation)

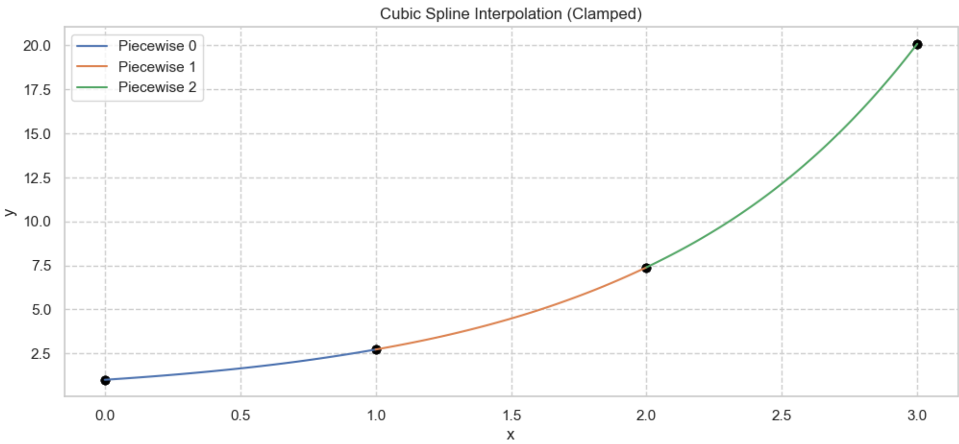

Figure 4: Example 2 (Clamped) Spline (Validation)

Spline, j	a_j	b_j	c_j	d_j
0	1.	1.	0.4446825	0.27359933
1	2.71828183	2.71016299	1.26548049	0.69513079
2	7.3890561	7.32651634	3.35087286	2.01909162
3	**20.08553692**			

Table 5: Example 2 (Clamped) Spline Coefficient Results (Validation)

```python
# import libraries
import math
import numpy as np
from sympy import symbols
```

```python
import matplotlib.pyplot as plt
# optional libraries for plotting
import seaborn as sns
sns.set()
sns.set_style("whitegrid", {'grid.linestyle': '--'})

'''
CubicSpline: This class contains 4 main functions.
1) cubic_spline()         : The main function for Natural Spline method and returns the
    coefficients of piecewise functions.
2) clamped_cubic_spline() : The main function for Clamped Spline method and returns the
    coefficients of piecewise functions.
3) create_Cspline_func()  : Creates the function with the defined coefficients and x-y points,
    and returns a symbolic function.
4) plot_Cspline_func()    : Plots the piecewise splines.

'''

class CubicSpline():
    def __init__(self):
        pass

    def cubic_spline(self,xData,yData):

        # Step 1
        h = np.zeros((len(xData)))
        for i in range(len(xData)-1):
            h[i] = xData[i+1] - xData[i]

        # Step 2
        alpha = np.zeros((len(xData)))
        for i in range(1,len(xData)-1):
            alpha[i] = (3/h[i])*(yData[i+1]-yData[i]) - (3/h[i-1])*(yData[i]-yData[i-1])

        # Step 3
        l = np.zeros((len(xData))); l[0]=1
        nu = np.zeros((len(xData))); nu[0]=0
        z = np.zeros((len(xData))); z[0]=0

        # Step 4
        for i in range(1,len(xData)-1):
            l[i] = 2*(xData[i+1]-xData[i-1]) - h[i-1]*nu[i-1]
            nu[i] = h[i]/l[i]
            z[i] = (alpha[i]-h[i-1]*z[i-1])/l[i]

        # Step 5
        l[-1] = 1; z[-1] = 0
        c = np.zeros((len(xData))); c[-1]=0

        # Step 6
        b = np.zeros((len(xData)))
        d = np.zeros((len(xData)))
        for i in list(range(0,len(xData)-1))[::-1]:
            c[i] = z[i] - nu[i]*c[i+1]
            b[i] = (yData[i+1]-yData[i])/h[i] - h[i]*(c[i+1]+2*c[i])/3
            d[i] = (c[i+1]-c[i])/(3*h[i])

        a = yData

        coef = [a,b,c,d]

        return coef

    def clamped_cubic_spline(self,xData,yData,d_left,d_right):
```

```

        # Step 1
        h = np.zeros((len(xData)))
        for i in range(len(xData)-1):
            h[i] = xData[i+1] - xData[i]

        # Step 2
        alpha = np.zeros((len(xData)))
        alpha[0] = 3*(yData[1]-yData[0])/h[0] - 3*d_left
        alpha[-1] = 3*d_right - 3*(yData[-1]-yData[-2])/h[-2]


        # Step 3
        for i in range(1,len(xData)-1):
            alpha[i] = (3/h[i])*(yData[i+1]-yData[i]) - (3/h[i-1])*(yData[i]-yData[i-1])

        # Step 4
        l = np.zeros((len(xData))); l[0]=2*h[0]
        nu = np.zeros((len(xData))); nu[0]=0.5
        z = np.zeros((len(xData))); z[0]=alpha[0]/l[0]

        # Step 5
        for i in range(1,len(xData)-1):
            l[i] = 2*(xData[i+1]-xData[i-1]) - h[i-1]*nu[i-1]
            nu[i] = h[i]/l[i]
            z[i] = (alpha[i]-h[i-1]*z[i-1])/l[i]

        # Step 6
        l[-1] = h[-2]*(2-nu[-2])
        z[-1] = (alpha[-1]-h[-2]*z[-2])/l[-1]
        c = np.zeros((len(xData))); c[-1]= z[-1]


        # Step 7
        b = np.zeros((len(xData)))
        d = np.zeros((len(xData)))
        for i in list(range(0,len(xData)-1))[::-1]:
            c[i] = z[i] - nu[i]*c[i+1]
            b[i] = (yData[i+1]-yData[i])/h[i] - h[i]*(c[i+1]+2*c[i])/3
            d[i] = (c[i+1]-c[i])/(3*h[i])

        a = yData
        c[-1] = 0

        coef = [a,b,c,d]

        return coef

    def create_Cspline_func(self,coefs,x,y):
        x_sym = symbols('x')

        fun = []
        for i in range(len(xData)-1):
            fun.append(coef[0][i] + coef[1][i]*(x_sym-xData[i]) + coef[2][i]*(x_sym-xData[i])**2
             ↪  + coef[3][i]*(x_sym-xData[i])**3)

        return fun

    def plot_Cspline_func(self,piecewise_func,xData,yData,plot_type='Clamped'):

        x_sym = symbols('x')

        # Create a Matplotlib figure and axis
        fig = plt.figure(figsize = (12,5))
```

```python

        for i in range(len(piecewise_func)):
            x_val = np.linspace(xData[i],xData[i+1],100)
            y_val = [piecewise_func[i].subs(x_sym, val) for val in x_val]

            plt.plot(x_val, y_val, label=f'Piecewise {i}')
            plt.scatter(xData,yData,color='black')

        # Customize the plot
        plt.xlabel('x')
        plt.ylabel('y')
        plt.legend()
        plt.grid(True)
        plt.title(f'Cubic Spline Interpolation ({plot_type})')

        # Save the figure for later use
        #fig.savefig('spline_plot.png')

        plt.show()

        return fig




# call main class
cs = CubicSpline()
# ---------------------------------------------------------- EXAMPLE 1 (VALIDATION - NATURAL SPLINE) ---------------------------------------------------------- #
# xData = np.array([0.0,1.0,2.0,3.0])
# yData = np.array([1.0,math.e,(math.e)**2,(math.e)**3])
# coef = cs.cubic_spline(xData,yData)
# piecewise_func = cs.create_Cspline_func(coef,xData,yData)
# cs.plot_Cspline_func(piecewise_func,xData,yData,plot_type='Natural');
# print('The integral area: %.7f' % (np.sum(coef[0][:-1]) + (1/2)*(np.sum(coef[1])) + (1/3)*(np.sum(coef[2]))))

# ---------------------------------------------------------- EXAMPLE 2 (VALIDATION - CLAMPED SPLINE) ---------------------------------------------------------- #
# xData = np.array([0.0,1.0,2.0,3.0])
# yData = np.array([1.0,math.e,(math.e)**2,(math.e)**3])
# d_left = 1; d_right = (math.e)**3
# coef = cs.clamped_cubic_spline(xData,yData,d_left,d_right)
# piecewise_func = cs.create_Cspline_func(coef,xData,yData)
# cs.plot_Cspline_func(piecewise_func,xData,yData,plot_type='Clamped');
# print('The integral area: %.7f' % (np.sum(coef[0][:-1]) + (1/2)*(np.sum(coef[1])) + (1/3)*(np.sum(coef[2]))))

# ------------------------------------------------------------- HOMEWORK PROBLEM (NATURAL SPLINE) ------------------------------------------------------------- #
# xData = np.array([0.0,1.0,2.0,3.0,4.0])
# yData = np.array([1.0,3.0,1.0,4.0,2.0])

# coef = cs.cubic_spline(xData,yData)
# piecewise_func = cs.create_Cspline_func(coef,xData,yData)
# cs.plot_Cspline_func(piecewise_func,xData,yData,plot_type='Natural');
# print('The integral area: %.7f' % (np.sum(coef[0][:-1]) + (1/2)*(np.sum(coef[1])) + (1/3)*(np.sum(coef[2])) + (1/4)*(np.sum(coef[3]))))

# ------------------------------------------------------------- HOMEWORK PROBLEM (CLAMPED SPLINE) ------------------------------------------------------------- #
xData = np.array([0.0,1.0,2.0,3.0,4.0])
yData = np.array([1.0,3.0,1.0,4.0,2.0])
d_left = 2; d_right = -1
```

```
coef = cs.clamped_cubic_spline(xData,yData,d_left,d_right)
piecewise_func = cs.create_Cspline_func(coef,xData,yData)
cs.plot_Cspline_func(piecewise_func,xData,yData,plot_type='Clamped');
print('The integral area: %.7f' % (np.sum(coef[0][:-1]) + (1/2)*(np.sum(coef[1])) +
↪  (1/3)*(np.sum(coef[2])) + (1/4)*(np.sum(coef[3]))))


print(coef)
```