An Analysis of Cubic Spline Interpolation

Alexander DeLise

April 5, 2024

1 Statement of the Problem

This project seeks to compare the performance and accuracy of natural and clamped cubic spline interpolation for the function $f(x) = x^5 - 4x^4 + 14x^2 - 17x + 6$ on the interval $[0, 3]$ using the nodes $x_0 = 0, x_1 = 1$ and $x_2 = 3$ to approximate $f(2.5)$. The primary objective is to determine which method provides a more accurate approximation of a given polynomial function on a specified interval. We determine whether the apporixmation is reasonable by comparing its absolute error to an established error bound.

2 Description of the Mathematics

Cubic spline interpolation is among the most popular polynomial interpolation techniques. A typical cubic spline generates a cubic function with four constants between successive nodes of known values. This allows the procedure enough flexibility to ensure that the interpolating polynomial is continuously differentiable on the interval of interpolation, and also that it has a continuous second derivative [**?**]. Using this motivation, one can define a cubic spline as the following: given a function f defined on an interval $[a, b]$ and a set of nodes $a = x_0 < x_1 < \cdots < x_n = b$, a cubic spline interpolant S for f is a function that satisfies the following conditions:

(a) $S(x)$ is a cubic polynomial, denoted $S_j(x)$, on the subinterval $[x_j, x_{j+1}]$ for each $j = 0, 1, \ldots, n-1$;

(b) $S_j(x) = f(x_j)$ and $S_j(x_{j+1}) = f(x_{j+1})$ for each $j = 0, 1, \ldots, n-1$;

(c) $S_{j+1}(x_{j+1}) = S_j(x_{j+1})$ for each $j = 0, 1, \ldots, n-2$;

(d) $S'_{j+1}(x_{j+1}) = S'_j(x_{j+1})$ for each $j = 0, 1, \ldots, n-2$;

(e) $S''_{j+1}(x_{j+1}) = S''_j(x_{j+1})$ for each $j = 0, 1, \ldots, n-2$;

(f) One of the following boundary conditions is satisfied:

  (i) $S''(x_0) = S''(x_n) = 0$ the *natural* boundary condition;

  (ii) $S'(x_0) = f'(x_0)$ and $S'(x_n) = f'(x_n)$, the *clamped* boundary condition.

One can imagine the natural cubic spline as the graph that approximates the shape that a long, flexible rod would take if it were forced to go through the nodes [**?**]. The clamped spine is also generally more accurate than a natural cubic spline since it requires more information about the original function.

Given n nodes for interpolation, a spline within a given interval requires $4n$ constants for the cubic polynomials. Each cubic spline interpolant between nodes has the general form

$$S_j(x) = a_j + b_j h_j + c_j h_j^2 + d_j h_j^3 \tag{1}$$

where $h_j = x_{j+1} - x_j$ for each $j = 0, 1, \ldots, n-1$. By definition, $S_j(x_j) = a_j = f(x_j)$, thus condition (c) can be applied to obtain

$$a_{j+1} = a_j + b_j h_j + c_j h_j^2 + d_j h_j^3. \tag{2}$$

for each $j = 0, 1, \ldots, n-2$. By defining $b_j = S'(x_j)$, condition (d) can be applied to obtain

$$b_{j+1} = b_j + 2c_j h_j + 3d_j h_j^2 \tag{3}$$

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

Moving on to the condition of the second derivative of a cubic spline interpolant, one defines $c_j = S''(x_j)/2$. By applying condition (e), one gets the result

$$c_{j+1} = c_j + 3d_j h_j \tag{4}$$

for each $j = 0, 1, \ldots, n-1$. By solving for d_j and substituting its value into equations (2) and (3) one achieves

$$a_{j+1} = a_j + b_j h_j + \frac{h_j^2}{3}(2c_j + c_{j+1}) \tag{5}$$

and

$$b_{j+1} = b_j + h_j(c_j + c_{j+1}) \tag{6}$$

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

The final relationship appears by solving equation (5) for b_j and then reducing its index by one, resulting in

$$b_{j-1} = \frac{1}{h_{j-1}}(a_j - a_{j-1} - \frac{h_{j-1}}{3})(2c_{j-1} + c - j) \tag{7}$$

for each $j = 0, 1, \ldots, n-1$. Finally, we can substitute the relationship from equation (7) into equation (6) to obtain

$$h_{j-1}c_{j-1} + 2(h_{j-1} + h_j)c_j + h_j c_{j+1} = \frac{3}{h_j}(a_{j+1} - a_j) - \frac{3}{h_{j-1}}(a_j - a_{j-1}) \tag{8}$$

for each $j = 1, 2, \ldots, n-1$. The final linear system in equation (8) contains only the set $\{c_j\}_{j=0}^n$ as unknowns. This occurs since the values of the set $\{h_j\}_{j=0}^n$ are obtained from the spacing of the nodes and the values of the set $\{a_j\}_{j=0}^n$ are the values of f at the nodes. Thus, once one solves for the values of the set $\{c_j\}_{j=0}^n$, one can find the values of $\{b_j\}_{j=0}^n$ from equation (7) and the values of the set $\{d_j\}_{j=0}^n$ from equation (4).

With this motivation, one might naturally question whether a unique solution exists for the natural and clamped cubic spline interpolations of a function. In fact, the following two theorems and their proofs from [**?**] guarantee such a unique interpolation exists for both conditions. The proofs of these theorems encapsulate the underlying process to determine the coefficients of the interpolants in the Python algorithm by solving a linear system.

Theorem 1. *If f is defined at $a = x_0 < x_1 < \ldots < x_n = b$, then f has a unique natural spline interpolant S on the nodes $x_0, x_1, \ldots, x_n$; that is, a spline interpolant that satisfies the natural boundary conditions $S''(a) = 0$ and $S''(b) = 0$.*

Proof. The boundary conditions of a natural cublic spline imply that $c_n = S''(x_n) = 0$ and that

$$0 = S''(x_n) = 2c_0 + 6d_0(x_0 - x_0) \Rightarrow c_0 = 0.$$

The values for c_0 and c_n along with eqation (8) create a linear system that is described by the matrix-vector equation $\boldsymbol{Ax} = \boldsymbol{b}$, where

$$\boldsymbol{A} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ h_0 & 2(h_0+h_1) & h_1 & & \\ 0 & h_1 & 2(h_1+h_2) & h_2 & \\ \vdots & & \ddots & \ddots & 0 \\ & & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & \cdots & 0 & 0 & 1 \end{bmatrix},$$

and $\boldsymbol{b}$ and $\boldsymbol{x}$ are the vectors

$$\boldsymbol{b} = \begin{bmatrix} 0 \\ \frac{3}{h_1}(a_2 - a_1) - \frac{3}{h_0}(a_1 - a_0) \\ \vdots \\ \frac{3}{h_{n-1}}(a_n - a_{n-1}) - \frac{3}{h_{n-2}}(a_{n-1} - a_{n-2}) \\ 0 \end{bmatrix}, \quad \text{and} \quad \boldsymbol{x} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix}.$$

Since the matrix $\boldsymbol{A}$ is strictly diagonally dominant, the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ has a unique solution. □

Theorem 2. *If f is defined at $a = x_0 < x_1 < \cdots < x_n = b$ and differentiable at a and b, then f has a unique clamped spline interpolant S on the nodes $x_0, x_1, \ldots, x_n$; that is, a spline interpolant that satisfies the clamped boundary conditions $S'(a) = f'(a)$ and $S'(b) = f'(b)$.*

Proof. Due to the equality $f'(a) = S'(a) = S'(x_0) = b_0$, equation (7), with an increase in index by one and $j = 0$ implies that

$$f'(a) = \frac{1}{h_0}(a_1 - a_0) - \frac{h_0}{3}(2c_0 + c_1).$$

As a result,

$$2h_0c_0 + h_0c_1 = \frac{3}{h_0}(a_1 - a_0) - 3f'(a). \tag{9}$$

In a similar fashion,

$$f'(b) = b_n = b_{n-1} + h_{n-1}(x_{n-1} + c_n),$$

so equation (7) implies that

$$f'(b) = \frac{a_n - a_{n-1}}{h_{n-1}} + \frac{h_{n-1}}{3}(c_{n-1} + 2n),$$

and

$$h_{n-1}c_{n-1} + 2h_{n-1}c_n = 3f'(b) - \frac{3}{h_{n-1}}(a_n - a_{n-1}). \tag{10}$$

Altogehter, equations (7), (9), and (10) determine the linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ where

$$\boldsymbol{A} = \begin{bmatrix} 2h_0 & h_0 & 0 & \cdots & 0 \\ h_0 & 2(h_0+h_1) & h_1 & & \\ 0 & h_1 & 2(h_1+h_2) & h_2 & \\ \vdots & & \ddots & \ddots & 0 \\ & & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & \cdots & 0 & h_{n-1} & 2h_{n-1} \end{bmatrix},$$

and $\boldsymbol{b}$ and $\boldsymbol{x}$ are the vectors

$$\boldsymbol{b} = \begin{bmatrix} \frac{3}{h_0}(a_1 - a_0) - 3f'(a) \\ \frac{3}{h_1}(a_2 - a_1) - \frac{3}{h_0}(a_1 - a_0) \\ \vdots \\ \frac{3}{h_{n-1}}(a_n - a_{n-1}) - \frac{3}{h_{n-2}}(a_{n-1} - a_{n-2}) \\ 3f'(b) - \frac{3}{h_{n-1}}(a_n - a_{n-1}) \end{bmatrix}, \quad \text{and} \quad \boldsymbol{x} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix}.$$

Once again, since the matrix $\boldsymbol{A}$ is strictly diagonally dominant, then the linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ has a unique solution. □

3 Description of the Algorithm

The code used to generate the cubic splines are based on the two pseudocodes found in [**?**]. Both algorithms take as input the number of data points n, the x-values $x_0, x_1, \ldots, x_n$, and the corresponding function values $a_0 = f(x_0), a_1 = f(x_1), \ldots, a_n = f(x_n)$. Additionally, for the clamped cubic spline, it takes the values of the first and last derivatives, denoted as FPO and FPN respectively. Each step of the algorithms is described as follows:

1. Calculate the differences between consecutive x-values, denoted as $h_i = x_{i+1} - x_i$.
2. Calculate the values of alpha, which depend on the function values and the spacing between the data points.
3. For the clamped cubic spline, calculate specific values for α_0 and α_n using the first and last derivative values.
4. Calculate the tridiagonal matrix coefficients l_i, μ_i, and z_i based on the differences between consecutive x-values and the calculated alpha values.
5. For the clamped cubic spline, adjust the last l_n, z_n, and c_n values accordingly.

6. Perform forward and backward substitutions to solve the tridiagonal system of equations for c_j.

7. Calculate the coefficients b_j and d_j based on the calculated c_j values.

Algorithm 1 Cubic Spline Interpolation

Require: n; $x_0, x_1, \ldots, x_n$; $a_0 = f(x_0), a_1 = f(x_1) \ldots a_n = f(x_n)$; FPO, FPN (only for clamped).
Ensure: a_j, b_j, c_j, d_j for $j = 0, 1, \ldots, n-1$.

procedure NATURAL CUBIC SPLINE(n, $x_0, x_1, \ldots, x_n$, $a_0, a_1, \ldots, a_n$)
    Step 1: For $i = 0, 1, \ldots, n-1$ set $h_i = x_{i+1} - x_i$.
    Step 2: For $i = 1, 2, \ldots, n-1$ set
    $\alpha_i = \frac{3}{h_i}(a_{i+1} - a_i) - \frac{3}{h_{i-1}}(a_i - a_{i-1})$.
    Step 3: Set $l_0 = 1$; $\mu_0 = 0$; $z_0 = 0$.
    Step 4: For $i = 1, 2, \ldots, n-1$ set
    $l_i = 2(x_{i+1} - x_{i-1}) - h_{i-1}\mu_{i-1}$;
    $\mu_i = \frac{h_i}{l_i}$;
    $z_i = \frac{\alpha_i - h_{i-1}z_{i-1}}{l_i}$.
    Step 5: Set $l_n = 1$; $z_n = 0$; $c_n = 0$.
    Step 6: For $j = n-1, n-2, \ldots, 0$ set
    $c_j = z_j - \mu_j c_{j+1}$;
    $b_j = \frac{a_{j+1} - a_j}{h_j} - \frac{h_j(c_{j+1} + 2c_j)}{3}$;
    $d_j = \frac{c_{j+1} - c_j}{3h_j}$.
end procedure

procedure CLAMPED CUBIC SPLINE(n, $x_0, x_1, \ldots, x_n$, $a_0, a_1, \ldots, a_n$, FPO, FPN)
    Step 2: Set $\alpha_0 = \frac{3(a_1 - a_0)}{h_0} - 3FPO$;
    $\alpha_n = 3FPN - \frac{3(a_n - a_{n-1})}{h_{n-1}}$.
    Step 6: Set $l_n = h_{n-1}(2 - \mu_{n-1})$;
    $z_n = \frac{\alpha_n - h_{n-1}z_{n-1}}{l_n}$;
    $c_n = z_n$.
end procedure

Step 7: **Output** (a_j, b_j, c_j, d_j for $j = 0, 1, \ldots, n-1$).

4 Results

To resummarize the problem, we aim to test the performance and accuracy of natural and clamped cubic spline interpolation for the function $f(x) = x^5 - 4x^4 + 14x^2 - 17x + 6$ on the interval $[0, 3]$ using the nodes $x_0 = 0, x_1 = 1$, and $x_2 = 3$ to approximate $f(2.5)$. Plugging in these values into the Python code listed at the end of the paper yields the following natural and clamped splines,

labeled $S_N(x)$ and $S_C(x)$, respectively:

$$S_N(x) = \begin{cases} x^3 - 7x + 6 & x \in [0,1] \\ -0.5(x-1)^3 + 3(x-1)^2 - 4(x-1) & x \in [1,3] \end{cases}$$

$$S_C(x) = \begin{cases} -12x^3 + 23x^2 + -17x + 6 & x \in [0,1] \\ 8.25(x-1)^3 + -13(x-1)^2 - 7(x-1) & x \in [1,3] \end{cases}$$

The graph of the natural and cubic splines, as well as that of the original function on the interval is displayed in Figure 1.

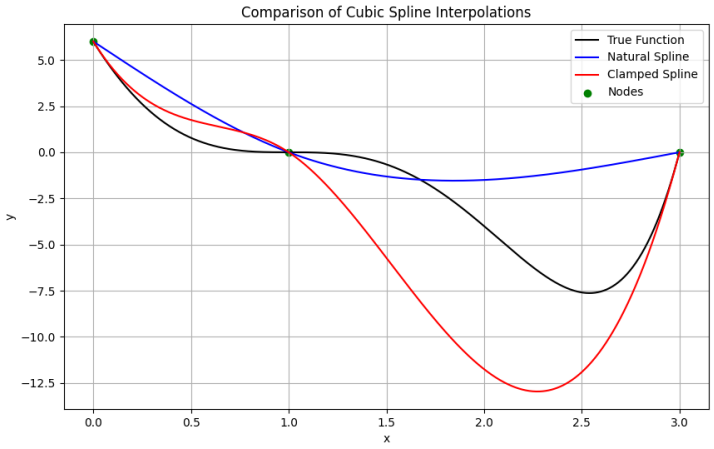

Figure 1: $S_N(x)$ and $S_C(x)$ vs. $f(x)$ for $x \in [0,3]$

Evaluating the splines and the function at $x = 2.5$ gives

$$f(2.5) = -7.59375 \quad S_N(2.5) = -0.9375, \quad S_C(2.5) = -11.90625.$$

Thus the absolute error of the cubic spline interpolations at $x = 2$ are

$$|f(2.5) - S_N(2.5)| = |-7.59375 - (-0.9375)| = 6.65625,$$

and

$$|f(2.5) - S_C(2.5)| = |-7.59375 - (-11.90625)| = 4.3125$$

It is thus oberserved that the better cubic spline interpolation to approximate $f(2.5)$ is the clamped cubic spline.

5 Conclusion

To determine whether the erros found when evaluating the splines at $x = 2.5$ are viable, we must introduce two theorems. Extensive literature exists on creating an error bound for a clamped

cubic spline on an interval, though the construction of an error bound for a natural cubic spline is much more difficult to derive [**?**]. Thus we introduce introduce the following error bounds from [**?**], without proof, for natural and clamped cubic splines, respectively. Note that the error bound for the natural cubic spline is a "softened" version of that of the clamped cubic spline since, in general, natural cubic splines are less accurate than their clamped counterparts [**?**].

Conjecture 1. *Let $f \in C^4[a,b]$ with $\max_{x\in[a,b]} |f^{(4)}(x)| = M$ and $R_N(x) = f(x) - S_N(x)$. If S_N is the unique natural cubic spline interpolant to f with respect to the nodes $a = x_0 < x_1 < \cdots < x_n = b$, then for all $x \in [a,b]$,*

$$|R_N(x)| \leq \frac{M}{60} \max_{0\leq j\leq n-1} (x_{j+1} - x_j)^4.$$

Theorem 3. *Let $f \in C^4[a,b]$ with $\max_{x\in[a,b]} |f^{(4)}(x)| = M$ and $R_C(x) = f(x) - S_C(x)$. If S_C is the unique clamped cubic spline interpolant to f with respect to the nodes $a = x_0 < x_1 < \cdots < x_n = b$, then for all $x \in [a,b]$,*

$$|R_C(x)| \leq \frac{5M}{384} \max_{0\leq j\leq n-1} (x_{j+1} - x_j)^4.$$

We now test the validity of the errors found in the Results section of the paper. First let $f^{(4)}(x) = 120x - 96$, $a = 0$, and $b = 3$. Solving for M yields

$$M = \max_{x\in[0,3]} |f^{(4)}(x)| = \max_{x\in[0,3]} |120x - 96| = |120(3) - 96| = 264,$$

since $f^{(4)}(x)$ monotonically increases on the interval. We determine the other component of the theorems in a modified fashion. Since we are evaluating the error bound at the specific subinterval $[1,3]$ for the point $x = 2.5$, we let $x_j = 1$ and $x_{j+1} = 3$. Performing this calculation gives:

$$(x_{j+1} - x_j)^4 = (x_2 - x_1)^4 = (3-1)^4 = 16.$$

Thus the error bounds for the natural cubic spline $S_N(x)$ and the clamped cubic spline $S_C(x)$ for $x \in [0,3]$ are

$$|R_N(x)| \leq \left(\frac{(264)}{60}\right) \cdot (16) = 70.4,$$

and

$$|R_C(x)| \leq \left(\frac{5(264)}{384}\right) \cdot (16) = 55.$$

Clearly, the two errors $|R_N(2.5)| = 6.65625$ and $|R_C(2.5)| = 4.3125$ fall within the two error bounds generated based on Conjecture 1 and Theorem 3. Despite this occurance, the errors generated by the cubic splines are still very large. In fact, their percent (relative) errors are 88% for the natural cubic spline and 57% for the clamped cubic spline. One insight into these large errors is due to the fact that the interval $[1,3]$, where $f(2.5)$ lies, is larger than the interval $[0,1]$. By examining Figure 1, one can empirically deduce that the errors in the interval $[0,1]$ are much smaller, on average, than the errors found in the interval $[1,3]$.

To demonstrate the power of increasing the number of nodes to minimize error, as an interesting side experiment, let us now consider the splines generated when adding a node at $x = 2$. Indeed it is evident in Figure 2 solely through empirical observation that the error between both types of splines with the original function is reduced on the interval for $x \in [1,3]$.

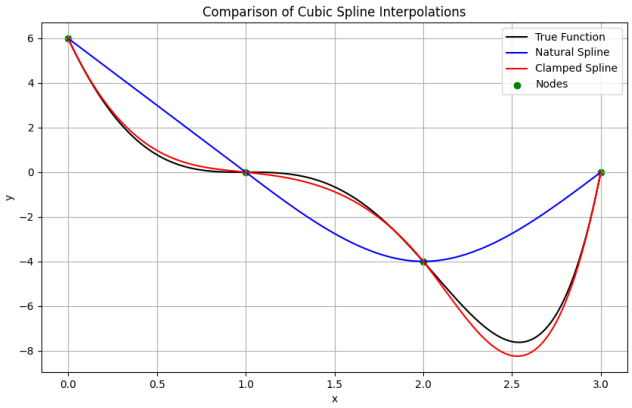

Figure 2: Improved $S_N(x)$ and $S_C(x)$ vs. $f(x)$ for $x \in [0, 3]$

Finally, let us consider one more scenario where nodes are evenly spaced through the interval $[0, 3]$ every 0.5 units, including the endpoints. We thus once again increase the number of nodes to interpolate. One should expect drastically imropoved results, especially in the "problem" subinterval $[1, 3]$. Once again through empirical observation, it is evident in Figure 3 that the error reduces

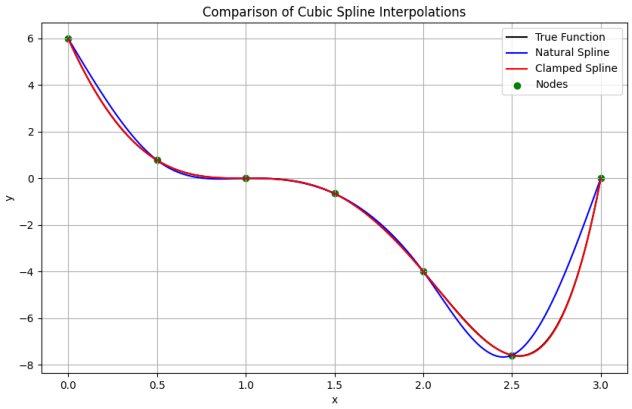

Figure 3: Improved $S_N(x)$ and $S_C(x)$ vs. $f(x)$ for $x \in [0, 3]$

drastically by increasing the number of nodes. It is thus cogent that in the original problem formulation, which divided the main interval into only two unevenly spaced subintervals, resulted in considerable errors, particularly in certain regions. As expected, increasing the number of subintervals for creating cubic splines reduces their errors. Indeed, the clamped cubic spline is essentially identical to the original function.

6 Program Listing

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

def natural_cubic_spline_coefficients(points):
    n = len(points) - 1
    a = [point[1] for point in points]
    x = [point[0] for point in points]

    # Step 1
    h = [x[i+1] - x[i] for i in range(n)]

    # Step 2
    alpha = [(3/h[i]) * (a[i+1] - a[i]) - (3/h[i-1]) * (a[i] - a[i-1]) for i in
                                                range(1, n)]

    # Step 3
    l = [1] * (n+1)
    mu = [0] * (n+1)
    z = [0] * (n+1)

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

    # Step 5
    l[n] = 1
    z[n] = 0
    c = [0] * (n+1)
    b = [0] * (n+1)
    d = [0] * (n+1)

    # Step 6
    for j in range(n-1, -1, -1):
        c[j] = z[j] - mu[j] * c[j+1]
        b[j] = (a[j+1] - a[j]) / h[j] - h[j] * (c[j+1] + 2*c[j]) / 3
        d[j] = (c[j+1] - c[j]) / (3 * h[j])

    coefficients = []
    for j in range(n):
        coefficients.append((a[j], b[j], c[j], d[j]))

    return coefficients

def clamped_cubic_spline_coefficients(points, FP0, FPN):
    n = len(points) - 1
    a = [point[1] for point in points]
    x = [point[0] for point in points]

    # Step 1
    h = [x[i+1] - x[i] for i in range(n)]

    # Step 2
```

```
        alpha = [(3/h[i]) * (a[i+1] - a[i]) - (3/h[i-1]) * (a[i] - a[i-1]) for i in
                                                range(1, n)]
        alpha.insert(0, (3/h[0]) * (a[1] - a[0]) - 3 * FP0)
        alpha.append(3 * FPN - (3/h[n-1]) * (a[n] - a[n-1]))

        # Step 3
        l = [1] * (n+1)
        mu = [0] * (n+1)
        z = [0] * (n+1)

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

        # Step 5
        l[n] = h[n-1] * (2 - mu[n-1])
        z[n] = (alpha[n-1] - h[n-1]*z[n-1]) / l[n]

        # Step 6
        c = [0] * (n+1)
        b = [0] * (n+1)
        d = [0] * (n+1)

        for j in range(n-1, -1, -1):
            c[j] = z[j] - mu[j] * c[j+1]
            b[j] = (a[j+1] - a[j]) / h[j] - h[j] * (c[j+1] + 2*c[j]) / 3
            d[j] = (c[j+1] - c[j]) / (3 * h[j])

        coefficients = []
        for j in range(n):
            coefficients.append((a[j], b[j], c[j], d[j]))

        return coefficients

    # Define the function for which to interpolate
    def f(x):
        return x**5 - 4*x**4 + 14*x**2 - 17*x + 6

    # Define the interval [a, b] and the number of data points
    a = 0
    b = 3
    n = 3

    # Generate x and y values for plotting
    x_values = np.array([0, 1, 3])
    y_values = np.array([6, 0, 0])

    # Compute the derivatives at the endpoints
    FP0 = 5*a**4 - 16*a**3 + 28*a - 17
    FPN = 5*b**4 - 16*b**3 + 28*b - 17

    # Compute the coefficients for both natural and clamped cubic splines
    natural_coeffs = natural_cubic_spline_coefficients(list(zip(x_values, y_values))
                                                       )
    clamped_coeffs = clamped_cubic_spline_coefficients(list(zip(x_values, y_values))
                                                       , FP0, FPN)
```

```
# Print coefficients
print("Natural Cubic Spline Coefficients:")
for j, coef in enumerate(natural_coeffs):
    print(f"Segment {j+1}: {coef}")

print("\nClamped Cubic Spline Coefficients:")
for j, coef in enumerate(clamped_coeffs):
    print(f"Segment {j+1}: {coef}")

# Plot the original function and both cubic splines
x_interpolated = np.linspace(a, b, 100)  # Interpolated x-values for smooth
                                           curve
y_natural_spline = []  # y-values for natural spline
y_clamped_spline = []  # y-values for clamped spline

# Compute y-values for natural spline and clamped spline
for x in x_interpolated:
    for i, (x0, x1) in enumerate(zip(x_values[:-1], x_values[1:])):
        if x0 <= x <= x1:
            a, b, c, d = natural_coeffs[i]
            y_natural_spline.append(a + b*(x-x0) + c*(x-x0)**2 + d*(x-x0)**3)
            a, b, c, d = clamped_coeffs[i]
            y_clamped_spline.append(a + b*(x-x0) + c*(x-x0)**2 + d*(x-x0)**3)
            break

# Compute y-values for original function
y_original_function = [f(x) for x in x_interpolated]

plt.plot(x_interpolated, y_natural_spline, label='Natural Cubic Spline')
plt.plot(x_interpolated, y_clamped_spline, label='Clamped Cubic Spline')
plt.plot(x_interpolated, y_original_function, label='Original Function')  #
                                           Corrected plotting of original
                                           function
plt.plot(x_values, y_values, 'ro')  # Original function points
plt.legend()
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Cubic Spline Interpolation')
plt.grid(False)
plt.show()
```