Project 1
Cubic Spline Interpolation

Lowell Monis

Mathematics 451 - Numerical Analysis I
Di Liu, Ph.D. (Instructor)
Jason Curtachio (Grader)

Fall 2024
Michigan State University
United States
14 October 2024

Prompt: *Write a code in Python that will fit data using cubic spline interpolation. Illustrate with an example for $n = 4$ and $f(x) = \ln(x)$ on the interval $[1, 5]$.*

One proceeds with constructing a function that takes in the given conditions, and switches between the natural and clamped interpolation methods, to produce the coefficients of the spline interpolant polynomials for each constituent interval within the given interval, while also producing a plot to demonstrate the interpolant polynomial's fit of the data.

Setup One can use Algorithm § 3.4 for Natural Cubic Splines and Algorithm § 3.5 for Clamped Cubic Splines from the text.[1]

To proceed, one can use the following modules:

1. `numpy`: NumPy's vectorization property can be put to good use to perform evaluations on intervals of data between two values.
2. `matplotlib`: To demonstrate a good fit of the data, one needs to generate plots. This module is imported to that end.

One also defines functions for the expression $\ln(x)$ and its derivative. While `sympy` could have been used here, regular Python operators are being used for the sake of simplification.

[1]:
```python
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return np.log(x)
def f_(x):
    return 1/x
```

Function Definition One uses a comprehensive Python user-defined function to compartmentalize the task. The parameters are explained in the function docstring. A boolean operator is used to switch between the natural and clamped methods. The natural method is used as a default, while the flag is activated to `True` whenever one needs to use the clamped method.

The function returns the coefficient matrix of the interpolant spline polynomials, as well as a plot that shows the fit of the spline polynomials to the data.

[2]:
```python
def cubic_spline(start, stop, n, function, derivative_function, clamped = False):
    """
    Computes the cubic spline interpolation for the function values given at␣
 ↪points x.
    Has the ability to use the natural and the clamped methods.

    Parameters:
        start: int, interval start
```

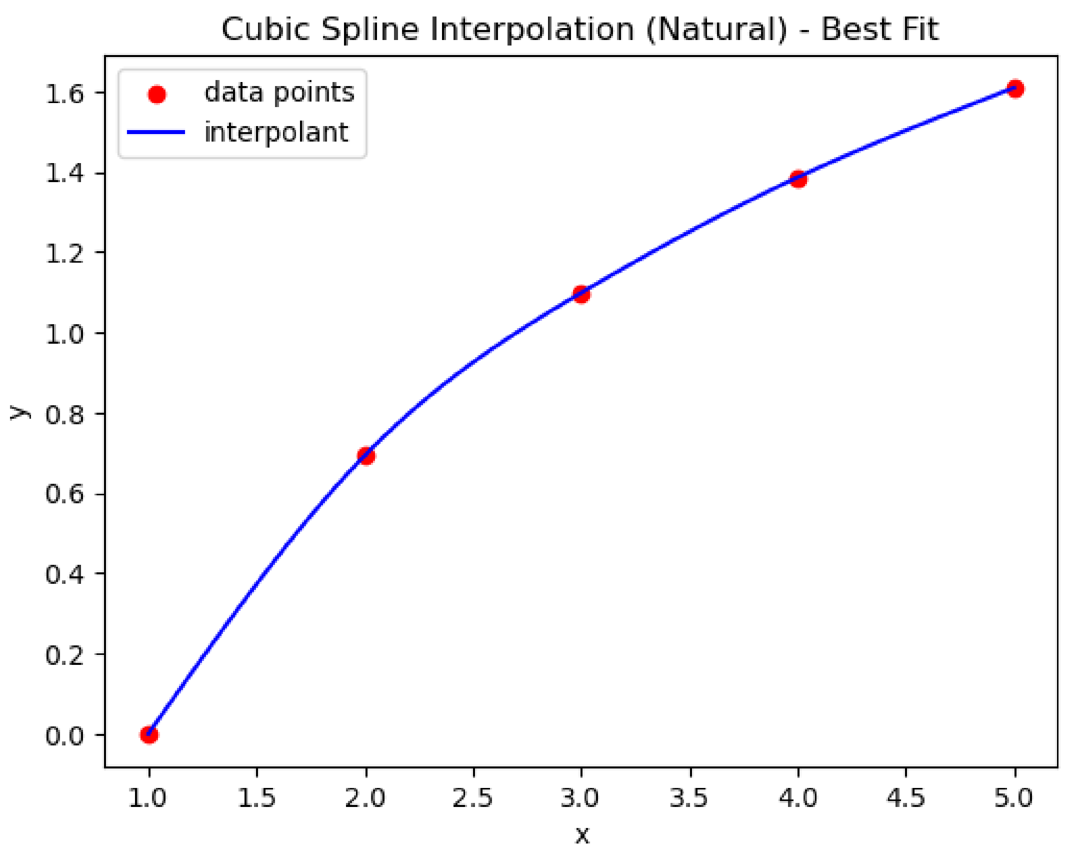
1. Richard L. Burden and J. Douglas Faires, *Numerical Analysis*, 9th ed. (Cengage Learning, 2010).

```
        stop: int, interval stop
        n: int, number of data points
        function: function object, expression of the function being evaluated
        derivative function: function object, expression of the derivative of␣
↪the function being evaluated
        clamped: bool, default False, do you want to evaluate clamped cubic␣
↪spline interpolation?

    Returns:
        coeffs: array of arrays of spline coefficients
        fig: plot figure
    """
    x = np.linspace(start, stop, n+1)  # Number of intervals
    h = np.diff(x)   # h[i] = x[i+1] - x[i]
    a = function(x)

    if clamped: # Clamped CS
        flag = "(Clamped)"
        FP0 = derivative_function(x[0])
        FPN = derivative_function(x[-1])

        # Calculate alpha
        alpha = np.zeros(n+1)
        alpha[0] = 3 * (a[1] - a[0]) / h[0] - (3 * FP0)  # clamped condition at␣
↪x0
        alpha[n] = (3 * FPN) - 3 * (a[n] - a[n-1]) / h[n-1]  # clamped condition␣
↪at xn

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

        # Solve the tridiagonal system
        l = np.ones(n+1)
        mu = np.zeros(n+1)
        z = np.zeros(n+1)

        l[0] = 2 * h[0]
        mu[0] = 0.5
        z[0] = alpha[0] / l[0]

        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] - (h[i-1] * z[i-1])) / l[i]

        l[n] = h[n-1] * (2 - mu[n-1])
        z[n] = (alpha[n] - h[n-1] * z[n-1]) / l[n]
        # Back substitution to solve for c, b, d
        c = np.zeros(n+1)
        b = np.zeros(n)
        d = np.zeros(n)

        c[n] = z[n]

        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])

        # Output: aj, bj, cj, dj for j = 0, ..., n-1
        aj = a
        bj = b
        cj = c
        dj = d

    else: # Natural CS
        flag = "(Natural)"
        # Compute alpha
        alpha = np.zeros(n)
        for i in range(1, n):
            alpha[i] = (3 / h[i]) * (a[i+1] - a[i]) - (3 / h[i-1]) * (a[i] -␣
↪a[i-1])

        # Set up the system
        l = np.ones(n+1)
        mu = np.zeros(n)
        z = np.zeros(n+1)

        # Forward elimination
        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] - h[i-1] * z[i-1]) / l[i]

        # Back substitution
        c = np.zeros(n+1)
        b = np.zeros(n)
        d = np.zeros(n)

        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])
        # Output: aj, bj, cj, dj for j = 0, ..., n-1
        aj = a[:-1]
        bj = b
        cj = c[:-1]
        dj = d

    y_plot = np.array([])
    setup=list(x)[:-1]
    for i in range(len(setup)):
        x_plot = np.arange(setup[i], setup[i]+1, 0.001)
        y_plot = np.append(y_plot, (aj[i] + (bj[i]*(x_plot-setup[i])) +␣
↪(cj[i]*(x_plot-setup[i])**2) + (dj[i]*(x_plot-setup[i])**3)))

    fig = plt.figure()
    plt.scatter(x, a, label = 'data points', color = 'red')
    plt.plot(np.arange(start, stop, 0.001), y_plot, label = 'interpolant', color␣
↪= 'blue')
    plt.title("Cubic Spline Interpolation " + flag + " - Best Fit")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend()

    coeffs = aj, bj, cj, dj

    return coeffs, plt
```

Solving the Prompt using Natural Cubic Spline Interpolation

[3]:
```
coefficients_natural, fig = cubic_spline(1, 5, 4, f, f_)
coefficients_natural
```

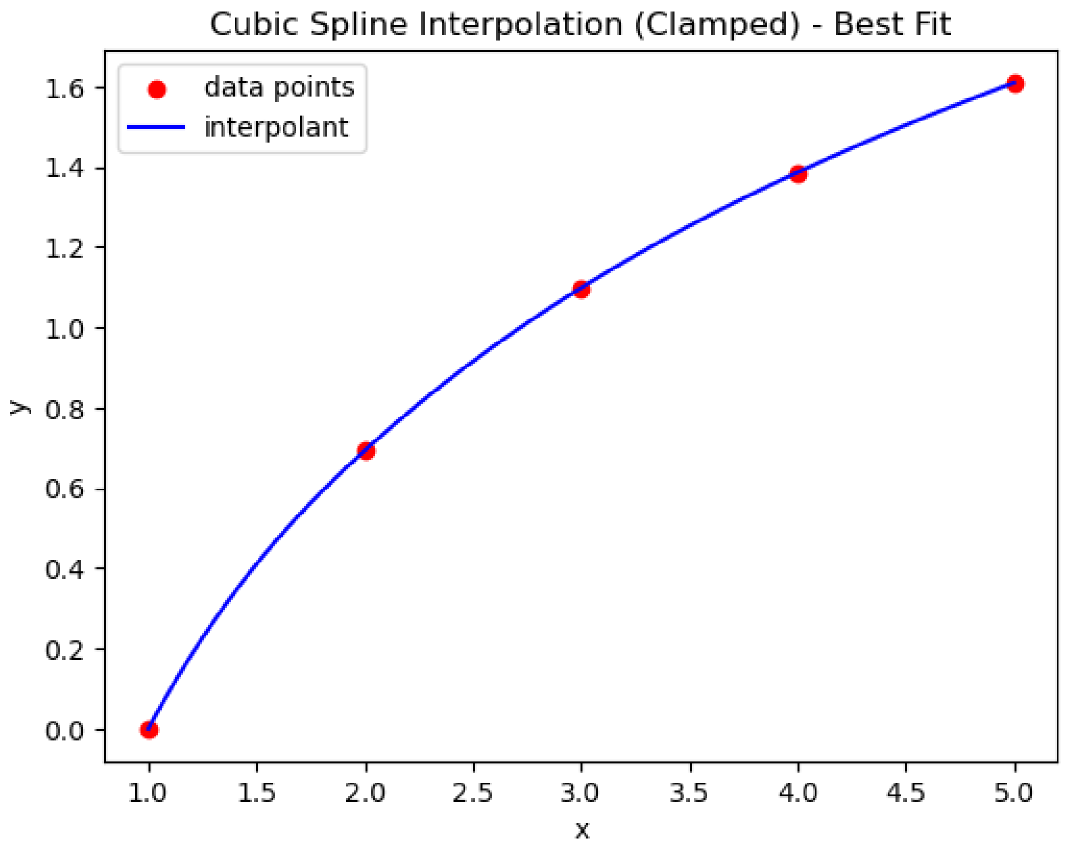
[112]:
```
(array([0.        , 0.69314718, 1.09861229, 1.38629436]),
 array([0.76294428, 0.55355299, 0.31868065, 0.25116597]),
 array([ 0.        , -0.20939129, -0.02548105, -0.04203363]),
 array([-0.0697971 ,  0.06130342, -0.00551753,  0.01401121]))
```



Solving the Prompt using Clamped Cubic Spline Interpolation Essentially, this method 'clamps' the end-points of the interval by taking the derivative of the function at those points.

```
[4]: coefficients_clamped, fig = cubic_spline(1, 5, 4, f, f_, clamped = True)
     coefficients_clamped
```

```
[113]: (array([0.        , 0.69314718, 1.09861229, 1.38629436, 1.60943791]),
        array([1.        , 0.49021899, 0.33496089, 0.249379  ]),
        array([-0.41077745, -0.09900355, -0.05625455, -0.02932734, -0.02005166]),
        array([0.10392463, 0.01424967, 0.00897574, 0.00309189]))
```



References

Burden, Richard L., and J. Douglas Faires. *Numerical Analysis*. 9th ed. Cengage Learning, 2010.