Project Design

Zhehao Chen 3220103172 *

December 8, 2024

Abstract

This project aims to develop a comprehensive program package that implements both piecewise-polynomial splines (pp-Form) and B-splines (B-Form). The package is designed to cover a range of spline functions, including linear and cubic splines under various boundary conditions.

Contents

*Electronic address: 3220103172@zju.edu.cn

1 Introduction

Splines are a fundamental tool in numerical analysis, offering a flexible means to approximate complex functions and data. This project focuses on the implementation of two types of splines: piecewise-polynomial splines and B-splines. The pp-Form splines are particularly useful for their simplicity and ease of computation, while B-splines offer a more robust framework for handling arbitrary knot sequences and higher-order splines. The project is structured to first develop the theoretical foundations and then proceed to the practical implementation of these splines in a computational environment.

2 Design of Core Supporting Classes

The spline interpolation implementation is supported by two fundamental header files that provide essential mathematical abstractions: `function.hpp` and `matrix.hpp`. These headers establish a robust foundation for numerical computations and function representations.

2.1 Function Class Hierarchy

The `function.hpp` header implements an object-oriented approach to function representation through a class hierarchy:

- **Abstract Base Class: Function**
  - Provides a pure virtual `eval()` method for function evaluation
  - Implements numerical differentiation methods using central difference formulas
  - Establishes a common interface for all function types
- **Concrete Class: DiscreteFunction**
  - Represents functions defined at discrete points
  - Stores function values, first derivatives, and second derivatives
  - Implements point-wise evaluation and derivative lookups
  - Ensures data consistency through validation in constructor
- **Concrete Class: Polynomial**
  - Represents polynomial functions through their coefficients
  - Provides efficient evaluation using Horner's method
  - Implements analytical derivative computation
  - Includes string representation for polynomial visualization

The design follows the SOLID principles, particularly the Single Responsibility Principle and the Open-Closed Principle, allowing for easy extension with new function types while maintaining a consistent interface.

2.2 Matrix Operations Framework

The `matrix.hpp` header provides essential linear algebra functionality:

- **Core Matrix Class**
  - Implements dynamic memory management with RAII principles
  - Provides comprehensive move and copy semantics
  - Uses row-major storage for efficient memory access
  - Includes bounds checking for safe element access
- **Linear System Solver**
  - Implements Gaussian elimination with partial pivoting
  - Handles singular matrix detection
  - Provides robust numerical stability through pivot selection
  - Essential for solving spline coefficient systems
- **Specialized ColVector Class**
  - Inherits from Matrix for consistent interface
  - Optimizes for column vector operations
  - Simplifies vector element access

2.3 Design Considerations

Several key design decisions enhance the robustness and usability of these classes:

1. **Exception Safety**
   - Comprehensive error checking in constructors and methods
   - Strong exception guarantees for memory operations
   - Clear error messages for debugging
2. **Memory Management**
   - Efficient use of move semantics for performance
   - Proper cleanup through RAII
   - Prevention of memory leaks in assignment operations
3. **Numerical Stability**
   - Use of appropriate tolerances for floating-point comparisons
   - Implementation of numerically stable algorithms
   - Careful handling of edge cases

These supporting classes provide a solid foundation for implementing various spline interpolation methods, offering both flexibility and reliability while maintaining high performance standards.

3 Framework Overview

The framework is organized into the following key components:

- **Base Classes:** Abstract base classes for splines and B-splines provide a unified interface and encapsulate shared functionality.
- **Derived Classes:** Specific spline types (e.g., linear, quadratic, cubic) inherit from base classes and implement specialized behaviors.
- **Factory Class:** The 'CubicBSplineFactory' class is included to generate cubic B-splines, encapsulating object creation logic.

4 Implementation of pp-Form

4.1 Base Class: `PiecewisePolynomialSpline`

This abstract base class provides a unified interface for all spline types. It includes the following key methods:

- `virtual void setKnots(const std::vector<double>& knots)`: Accepts the knot points for the spline.
- `virtual void setCoefficients(const std::vector<double>& values)`: Accepts the values or derivatives for spline computation.
- `virtual double evaluate(double x) const = 0`: A pure virtual function for evaluating the spline at a given point.

4.2 `LinearSpline`

To construct a linear function for each segment, we can use the relationship:

$$s(x) - y_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i).$$

By rearranging this equation, we can solve for $s(x)$:

$$s(x) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}x + y_i - \frac{y_{i+1} - y_i}{x_{i+1} - x_i}x_i.$$

This formula represents the linear function for the segment between x_i and x_{i+1}.

The `LinearSpline` class implements piecewise linear interpolation. Key methods include:

- `void computeCoefficients()`: Computes the coefficients for each segment.
- `double evaluate(double x) const override`: Evaluates the linear spline at a point.

4.3 `CubicSpline`

The `CubicSpline` class handles cubic spline interpolation. Its design allows flexibility in boundary conditions, supported by derived classes for specific cases.

- `void computeNaturalCoefficients()`: Computes coefficients for natural boundary conditions.
- `void computeClampedCoefficients(double f_prime0, double f_primeN)`: Implements clamped boundary conditions.
- `void computePeriodicSpline()`: Implements periodic boundary conditions.
- `double evaluate(double x) const override`: Evaluates the cubic spline.

A cubic spline segment on $[x_i, x_{i+1}]$ is defined as:

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3.$$

For the three boundary conditions, we have:

4.3.1 `void computeNaturalCoefficients()`

- a_i are set to input function values
- $h_i = x_{i+1} - x_i$ are interval lengths

Constructs and solves a tridiagonal system for second derivatives:

$$\frac{h_i}{6}c_{i-1} + \frac{2(h_i + h_{i+1})}{6}c_i + \frac{h_{i+1}}{6}c_{i+1} = \frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}}.$$

Applies natural boundary conditions:

- Sets $c_0 = c_n = 0$ (second derivatives at endpoints)
- Uses Thomas algorithm for efficient solution

Calculates remaining coefficients:

$$b_i = \frac{y_{i+1} - y_i}{h_i} - \frac{h_i}{3}(c_{i+1} + 2c_i),$$
$$d_i = \frac{c_{i+1} - c_i}{3h_i}.$$

4.3.2 `void computeClampedCoefficients(double f_prime0, double f_primeN)`

- Mostly the same as `void computeNaturalCoefficients()`
- Boundary Conditions:

```
alpha[0] = 3.0 * ((a[1] - a[0]) / h[0] - leftDerivative);
alpha[n] = 3.0 * (rightDerivative - (a[n] - a[n-1]) / h[n-1]);
```

4.3.3 `void computePeriodicSpline()`

Because the matrix equation generated by this boundary condition is not in tridiagonal form and the calculation is more complicated, we use the Eigen library to solve the equation $Ax = \alpha$.

Among them,

$$\mathbf{A} = \begin{bmatrix} 2 & \lambda_0 & 0 & \cdots & 0 & \mu_0 \\ \mu_1 & 2 & \lambda_1 & \cdots & 0 & 0 \\ 0 & \mu_2 & 2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 2 & \lambda_{n-3} \\ \lambda_{n-2} & 0 & 0 & \cdots & \mu_{n-2} & 2 \end{bmatrix},$$

$$\lambda_i = \frac{h_{i+1}}{h_i + h_{i+1}}, \quad \mu_i = \frac{h_i}{h_i + h_{i+1}}, \quad \alpha_i = 6 \cdot \frac{\frac{y_{i+2}-y_{i+1}}{h_{i+1}} - \frac{y_{i+1}-y_i}{h_i}}{h_{i+1} + h_i}.$$

And with special handling for the periodic case at $i = n - 2$:

$$\lambda_{n-2} = \frac{h_0}{h_{n-2} + h_0}, \quad \mu_{n-2} = \frac{h_{n-2}}{h_{n-2} + h_0}, \quad \alpha_{n-2} = 6 \cdot \frac{\frac{y_1-y_0}{h_0} - \frac{y_{n-1}-y_{n-2}}{h_{n-2}}}{h_{n-2} + h_0}.$$

In the end, we set

$$\begin{bmatrix} M_0 \\ M_1 \\ M_2 \\ M_3 \\ \vdots \\ M_{n-2} \\ M_{n-1} \end{bmatrix} = \begin{bmatrix} \alpha_{n-2} \\ \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{n-3} \\ \alpha_{n-2} \end{bmatrix}.$$

Thereby, we can calculate coefficients:

$$\begin{aligned} b_i &= \frac{y_{i+1} - y_i}{h_i} - \frac{h_i(M_{i+1} + 2M_i)}{6}, \\ c_i &= \frac{M_i}{2}, \\ d_i &= \frac{M_{i+1} - M_i}{6h_i}. \end{aligned}$$

4.4 `Degree 4 pp-Spline`

Here we implement only one boundary condition, i.e. $s'''(x_0) = 0, s''(x_0) = 0, s'''(x_{n-1}) = 0$.

4.4.1 Constructor

The constructor of the class, `Degree_4_pp`, takes two arguments: a vector of knots and a vector of values. The constructor checks if the sizes of the knots and values vectors are compatible and if there are at least two knots. It then calls the `computeCoefficients` method to compute the polynomial coefficients for each segment.

4.4.2 Evaluation Method

The spline is evaluated at a given point x using the `evaluate` method. The method first checks if the point x is within the bounds of the spline. It then iterates over the intervals defined by the knots and computes the polynomial value using the degree-4 polynomial form for the appropriate segment.

4.4.3 Printing the Spline Expression

The method `printExpression` prints the mathematical expression of the spline for each segment. Each segment's polynomial is printed in the form:

$$P_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 + e_i(x - x_i)^4$$

where a_i, b_i, c_i, d_i, e_i are the polynomial coefficients for the i-th segment.

4.4.4 Computing the Coefficients

The computation of the polynomial coefficients is performed by solving a system of linear equations. The polynomial is defined piecewise, with each segment represented by a degree-4 polynomial:

$$P_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 + e_i(x - x_i)^4$$

We need to determine the coefficients a_i, b_i, c_i, d_i, e_i for each segment.

The following steps describe the coefficient computation process:

Step 1: Setup the Variables and Knots
The differences between consecutive knots, denoted $h_i = x_{i+1} - x_i$, are computed. These differences are used to form the system of equations.

Step 2: Construct the Matrix System
The matrix system is constructed using boundary conditions for the spline. The system is divided into different parts corresponding to different boundary conditions. The matrix G is constructed as a block matrix:

$$G = \begin{pmatrix} G_1 & G_2 \\ G_3 & G_4 \end{pmatrix}$$

where:

$$G_1 = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ h_0 & h_1 & 0 & \cdots & 0 \\ 0 & h_1 & h_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & h_{n-3} & h_{n-2} \end{pmatrix}_{(n-1)\times(n-1)}$$

$$G_2 = \begin{pmatrix} 0 & 0 & 0 & 0 & \cdots & 0 \\ \frac{5}{4}h_0^2 & \frac{3}{4}h_0^2+\frac{1}{4}h_1^2 & \frac{1}{4}h_1^2 & 0 & \cdots & 0 \\ 0 & \frac{5}{4}h_1^2 & \frac{3}{4}h_1^2+\frac{1}{4}h_2^2 & \frac{1}{4}h_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{5}{4}h_{n-3}^2 & \frac{3}{4}h_{n-3}^2+\frac{1}{4}h_{n-2}^2 & \frac{1}{4}h_{n-2}^2 \end{pmatrix}_{(n-1)\times n}$$

$$G_3 = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ 2 & -2 & 0 & \cdots & 0 \\ 0 & 2 & -2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 2 & -2 \\ 0 & 0 & \cdots & 0 & 0 \end{pmatrix}_{n\times(n-1)}$$

$$G_4 = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ h_0 & h_0 & 0 & \cdots & 0 & 0 \\ 0 & h_1 & h_1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & h_{n-3} & h_{n-3} & 0 \\ 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix}_{n\times n}$$

The right-hand side vector α is constructed as:

$$\alpha = \begin{pmatrix} 0 \\ \frac{y_2-y_1}{h_1} - \frac{y_1-y_0}{h_0} \\ \vdots \\ \frac{y_{n-1}-y_{n-2}}{h_{n-2}} - \frac{y_{n-2}-y_{n-3}}{h_{n-3}} \\ 0 \\ \vdots \\ 0 \end{pmatrix}_{(2n-1)\times 1}$$

Step 3: Solve the System
The system $Gx = \alpha$ is solved using the Eigen library, which applies LU decomposition to compute the solution for x. The solution x contains the coefficients for the spline segments.

Step 4: Extract the Coefficients
After solving the system, the coefficients c_i and d_i are extracted from the solution vector x, and the remaining coefficients b_i and e_i are computed based on the values of h_i, c_i, and d_i:

$$
\begin{aligned}
&c_i = x_i, \quad d_i = x_{i+n-1}, \quad i = 0, \ldots, n-2, \quad d_{n-1} = x_{2n-2}, \\
&b_i = \frac{a_{i+1} - a_i}{h_i} - c_i h_i - \frac{1}{4} d_{i+1} h_i^2 - \frac{3}{4} d_i h_i^2, \quad i = 0, \ldots, n-3, \\
&e_i = \frac{1}{4} \frac{d_{i+1} - d_i}{h_i}, \quad i = 0, \ldots, n-3.
\end{aligned}
$$

The final coefficients for the last segment are computed as:

$$
\begin{aligned}
b_{n-2} &= b_{n-3} + 2c_{n-3}h_{n-3} + 3d_{n-3}h_{n-3}^2 + 4e_{n-3}h_{n-3}^3, \\
e_{n-2} &= \frac{a_{n-1} - a_{n-2} - b_{n-2}h_{n-2} - c_{n-2}h_{n-2}^2 - d_{n-2}h_{n-2}^3}{h_{n-2}^4}.
\end{aligned}
$$

4.5 `Degree 5 pp-Spline`

Here we implement only one boundary condition, i.e. $s'''(x_0) = 0, s^{(4)}(x_0) = 0, s'''(x_{n-1}) = 0, s^{(4)}(x_{n-1}) = 0$.

Since most of the content is the same as `Degree 4 pp-Spline`, let's focus on part of computing the Coefficients

4.5.1 Computing the Coefficients

Here we only give the matrix equation for the autumn coefficients and the general formula for extracting the coefficients.

The matrix G is constructed as a block matrix:

$$
G = \begin{pmatrix} G_1 & G_2 & G_3 \\ G_4 & G_5 & G_6 \\ G_7 & G_8 & G_9 \end{pmatrix}
$$

where:

$$
G_1 = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ h_0 & h_1 & 0 & \cdots & 0 \\ 0 & h_1 & h_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & h_{n-3} & h_{n-2} \end{pmatrix}_{(n-1) \times (n-1)}
$$

$$
G_2 = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ 2h_0^2 & h_1^2 & 0 & \cdots & 0 & 0 \\ 0 & 2h_1^2 & h_2^2 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & 2h_{n-3}^2 & h_{n-2}^2 & 0 \end{pmatrix}_{(n-1) \times n}
$$

$$G_3 = \begin{pmatrix} 0 & 0 & 0 & 0 & \cdots & 0 \\ \frac{11}{5}h_0^3 & \frac{4}{5}h_0^3 + \frac{4}{5}h_1^3 & \frac{1}{5}h_1^3 & 0 & \cdots & 0 \\ 0 & \frac{11}{5}h_1^3 & \frac{4}{5}h_1^3 + \frac{4}{5}h_2^3 & \frac{1}{5}h_2^3 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{11}{5}h_{n-3}^3 & \frac{4}{5}h_{n-3}^3 + \frac{4}{5}h_{n-2}^3 & \frac{1}{5}h_{n-2}^3 \end{pmatrix}_{(n-1)\times n}$$

$$G_4 = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ 2 & -2 & 0 & \cdots & 0 \\ 0 & 2 & -2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 2 & -2 \\ 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 \end{pmatrix}_{(n+1)\times(n-1)}$$

$$G_5 = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 & 0 \\ 6h_0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 6h_1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 6h_{n-3} & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 & 1 \\ 0 & 0 & \cdots & 0 & 0 & 0 \end{pmatrix}_{(n+1)\times n}$$

$$G_6 = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ 8h_0^2 & 4h_0^2 & 0 & \cdots & 0 & 0 \\ 0 & 8h_1^2 & 4h_1^2 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 8h_{n-3}^2 & 4h_{n-3}^2 & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix}_{(n+1)\times n}$$

$$G_7 = \big(O\big)_{(n-1)\times(n-1)}$$

$$G_8 = \begin{pmatrix} 6 & -6 & 0 & \cdots & 0 \\ 0 & 6 & -6 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 6 & -6 \end{pmatrix}_{(n-1)\times n}$$

$$G_9 = \begin{pmatrix} 12h_0 & 12h_0 & 0 & \cdots & 0 \\ 0 & 12h_1 & 12h_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 12h_{n-2} & 12h_{n-2} \end{pmatrix}_{(n-1)\times n}$$

The right-hand side vector α is constructed as:

$$\alpha = \begin{pmatrix} 0 \\ \frac{y_2-y_1}{h_1} - \frac{y_1-y_0}{h_0} \\ \vdots \\ \frac{y_{n-1}-y_{n-2}}{h_{n-2}} - \frac{y_{n-2}-y_{n-3}}{h_{n-3}} \\ 0 \\ \vdots \\ 0 \end{pmatrix}_{(3n-1)\times 1}$$

The system $Gx = \alpha$ is solved using the Eigen library, which applies LU decomposition to compute the solution for x. The solution x contains the coefficients for the spline segments.

After solving the system, the coefficients c_i, d_i and e_i are extracted from the solution vector x, and the remaining coefficients b_i and f_i are computed based on the values of h_i, c_i, d_i and e_i:

$$\begin{aligned}
&c_i = x_i, \quad d_i = x_{i+n-1}, \quad e_i = x_{i+2n-1}, \quad i = 0, \ldots, n-2, \\
&d_{n-1} = x_{2n-2}, \quad e_{n-1} = x_{3n-2}, \\
&b_i = \frac{a_{i+1} - a_i}{h_i} - c_i h_i - d_i h_i^2 - \frac{4}{5} e_i h_i^3 - \frac{1}{5} e_{i+1} h_i^3, \quad i = 0, \ldots, n-3, \\
&f_i = \frac{1}{5} \frac{e_{i+1} - e_i}{h_i}, \quad i = 0, \ldots, n-3.
\end{aligned}$$

The final coefficients for the last segment are computed as:

$$\begin{aligned}
b_{n-2} &= b_{n-3} + 2c_{n-3}h_{n-3} + 3d_{n-3}h_{n-3}^2 + 4e_{n-3}h_{n-3}^3 + 5f_{n-3}h_{n-3}^4, \\
f_{n-2} &= \frac{a_{n-1} - a_{n-2} - b_{n-2}h_{n-2} - c_{n-2}h_{n-2}^2 - d_{n-2}h_{n-2}^3 - e_{n-2}h_{n-2}^4}{h_{n-2}^5}.
\end{aligned}$$

5 Implementation of B-Form

5.1 Base Class: `BSpline`

The `BSpline` class serves as the abstract base class. It includes methods for evaluating basis functions, derivatives, and the spline value at a given point x.

- `basis`: Recursively computes the value of the n-th order basis function $B_{i,n}(x)$ using the Cox-de Boor recursion formula:

$$B_{i,0}(x) = \begin{cases} 1, & \text{if } t_{i-1} \le x < t_i, \\ 0, & \text{otherwise.} \end{cases}$$

$$B_{i,n}(x) = \frac{x - t_{i-1}}{t_{i+n-1} - t_{i-1}} B_{i,n-1}(x) + \frac{t_{i+n} - x}{t_{i+n} - t_i} B_{i+1,n-1}(x),$$

where the support interval of $B_{i,n}(x)$ is $[t_{i-1}, t_{i+n}]$.

- `basisDeriv`: Computes the first derivative of the basis function $B_{i,n}(x)$.
- `basisDeriv2`: Computes the second derivative of the basis function $B_{i,n}(x)$.
- `eval`: Computes the spline value at x as a weighted sum of basis functions:

$$S(x) = \sum_{i=0}^{n} c_i B_{i,n}(x),$$

  where c_i are the spline coefficients.

5.2 LinearBSpline

This class implement first-order B-spline. The class extends `BSpline` and overrides `getBasis` to specify the order of the basis functions.

5.3 QuadraticBSpline

5.3.1 Constructors

- The first constructor takes a vector of points and a corresponding vector of values. It checks if the number of points is at least 2 and if the number of values is 1 bigger than the number of knots.

  Because we made the "middle point" of nodes in the implementation process, the number of nodes and the number of function values are the same in the final operation, so as to simplify the calculation.

```
1     QuadraticBSpline(const std::vector<double>& points, const
      Function& f) {
2         std::vector<double> values;
3         values.push_back(eval(points.front()));
4         for (int i = 0; i < points.size() - 1; i++) {
5             values.push_back(eval((points[i] + points[i+1]) /
      2));
6         }
7         values.push_back(eval(points.back()));
8         *this = QuadraticBSpline(points, values);
9     }
10
```

  It then sets up the knots and computes the coefficients for the B-spline.

- The second constructor takes a vector of points and a function object. It evaluates the function at specific points to generate the values needed for the B-spline and then calls the first constructor to complete the initialization.

5.3.2 Basis Function

The `getBasis` method overrides the base class method to compute the basis function value for a quadratic B-spline.

5.3.3 Private Methods

- `setupKnots` sets up the knots vector based on the input points. It adds two outer knots that are offset by 2 units from the first and last points, respectively.
- `computeCoefficients` constructs and solves a system of linear equations to find the coefficients of the B-spline.

5.3.4 Matrix Equation for Coefficients

The matrix equation for computing the coefficients of the quadratic B-spline is derived from the conditions that the B-spline must satisfy at each point. The system of equations is represented as:

$$A\mathbf{c} = \mathbf{b}$$

where:
- A is a $(size + 1) \times (size + 1)$ matrix.
- $\mathbf{c}$ is a column vector of coefficients.
- $\mathbf{b}$ is a column vector of values at specific points.

The matrix A and vector $\mathbf{b}$ are constructed as follows:

$$A = \begin{bmatrix} B_{1,2}(p_0) & B_{2,2}(p_0) & 0 & \cdots & 0 & \\ 0 & B_{2,2}(p_1) & B_{3,2}(p_1) & \cdots & 0 & \\ \vdots & \vdots & \vdots & \ddots & \vdots & \\ B_{i,2}(p_{i-1}) & B_{i+1,2}(p_i) & B_{i+2,2}(p_i) & \cdots & 0 & \\ \vdots & \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & 0 & \cdots & B_{n,2}(p_{n-1}) & B_{n+1,2}(p_{n-1}) \\ B_{n,2}(p_{n-1}) & B_{n+1,2}(p_n) & 0 & \cdots & 0 & \end{bmatrix}$$

$$\mathbf{b} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{i+1} \\ \vdots \\ y_{n+1} \end{bmatrix}$$

Here, $B_{i,k}(p)$ represents the B-spline basis function of degree k at point p, and y_i are the given values at the points p_i.

5.3.5 Conclusion

The `QuadraticBSpline` class provides an efficient way to construct and evaluate quadratic B-splines. It handles the necessary computations for setting up

the knots and solving the system of equations to find the coefficients. This implementation allows for the creation of quadratic B-splines that interpolate a given set of data points or values.

5.4 `CubicBSpline` and Derived Classes

`CubicBSpline` implements third-order B-splines. Its derived classes represent specific types of cubic splines:

- `NaturalCubicBSpline`: Enforces natural boundary conditions ($S''(t_0) = S''(t_n) = 0$).
- `ClampedCubicBSpline`: Enforces clamped boundary conditions ($S'(t_0) =$ startDeriv, $S'(t_n) =$ endDeriv).
- `PeriodicCubicBSpline`: Ensures periodicity in both function and derivatives.

For each derived class, the coefficients c_i are computed by solving a system of linear equations $A\mathbf{c} = \mathbf{b}$. The matrix A and vector $\mathbf{b}$ are constructed based on interpolation and boundary conditions.

5.4.1 `NaturalCubicBSpline`

For a natural cubic B-spline with n points, we need to solve the matrix equation $A\mathbf{c} = \mathbf{b}$, where: The matrix A is $(n+2) \times (n+2)$ with the following structure:

$$A = \begin{bmatrix} B_1(x_0) & B_2(x_0) & B_3(x_0) & 0 & \cdots & 0 \\ 0 & B_2(x_1) & B_3(x_1) & B_4(x_1) & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & B_{n-1}(x_{n-1}) & B_n(x_{n-1}) & B_{n+1}(x_{n-1}) \\ B_1''(x_0) & B_2''(x_0) & B_3''(x_0) & 0 & \cdots & 0 \\ 0 & \cdots & 0 & B_{n-1}''(x_{n-1}) & B_n''(x_{n-1}) & B_{n+1}''(x_{n-1}) \end{bmatrix},$$

The coefficient vector $\mathbf{c}$ and right-hand side vector $\mathbf{b}$ are:

$$\mathbf{c} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n+1} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \\ 0 \\ 0 \end{bmatrix}$$

where:

- $B_i(x)$ are the cubic B-spline basis functions.
- $B_i''(x)$ are their second derivatives.

- y_i are the input values at points x_i.

The last two equations enforce zero second derivatives at the endpoints.

5.4.2 `ClampedCubicBSpline`

For a clamped cubic B-spline, the matrix equation has the same size but different boundary conditions:

$$A = \begin{bmatrix} B_1(x_0) & B_2(x_0) & B_3(x_0) & 0 & \cdots & 0 \\ 0 & B_2(x_1) & B_3(x_1) & B_4(x_1) & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & B_{n-1}(x_{n-1}) & B_n(x_{n-1}) & B_{n+1}(x_{n-1}) \\ B_1'(x_0) & B_2'(x_0) & B_3'(x_0) & 0 & \cdots & 0 \\ 0 & \cdots & 0 & B_{n-1}'(x_{n-1}) & B_n'(x_{n-1}) & B_{n+1}'(x_{n-1}) \end{bmatrix},$$

$$\mathbf{c} = \begin{bmatrix} c_0 \\ c_2 \\ \vdots \\ c_{n+1} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \\ s_0 \\ s_1 \end{bmatrix}$$

where:

- $B_i'(x)$ are the first derivatives of the basis functions.
- s_0 and s_1 are the specified derivatives at the endpoints.

5.4.3 `PeriodicCubicBSpline`

For a periodic cubic B-spline, the matrix equation enforces continuity of both first and second derivatives across the boundary:

$$A = \begin{bmatrix} B_1(x_0) & B_2(x_0) & B_3(x_0) & 0 & \cdots & 0 \\ 0 & B_2(x_1) & B_3(x_1) & B_4(x_1) & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & B_{n-1}(x_{n-1}) & B_n(x_{n-1}) & B_{n+1}(x_{n-1}) \\ B_1'(x_0) & B_2'(x_0) & B_3'(x_0) & \cdots & -B_n'(x_{n-1}) & -B_{n+1}'(x_{n-1}) \\ B_1''(x_0) & B_2''(x_0) & B_3''(x_0) & \cdots & -B_n''(x_{n-1}) & -B_{n+1}''(x_{n-1}) \end{bmatrix},$$

$$\mathbf{c} = \begin{bmatrix} c_0 \\ c_2 \\ \vdots \\ c_{n+1} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \\ 0 \\ 0 \end{bmatrix}$$

where the last two equations enforce:

- Equality of first derivatives across the boundary (periodicity of first derivative).
- Equality of second derivatives across the boundary (periodicity of second derivative).

5.4.4 `CubicBSplineFactory`

Class Design
The `CubicBSplineFactory` class is designed with a static method `createSpline` that takes several parameters to determine the type of cubic B-spline to create:

- `type`: An integer that specifies the boundary condition (1 for Natural, 2 for Clamped, 3 for Periodic).
- `points`: A vector of double values representing the knots of the B-spline.
- `values`: A vector of double values representing the function values at the knots.
- `startDeriv` and `endDeriv`: Optional parameters for the first derivatives at the start and end of the B-spline, used in Clamped boundary conditions.

Conclusion
The `CubicBSplineFactory` class provides a centralized and efficient way to create cubic B-splines with different boundary conditions. This design enhances the maintainability and scalability of the spline creation process.

6 Convergence Order Analysis of Splines

6.1 Linear Splines

6.1.1 Error Expansion Using Taylor Series

Consider the target function $f(x)$ to be sufficiently smooth, with derivatives up to order two. Let x_i and x_{i+1} denote two consecutive nodes in the partition of the interval, with $h_n = x_{i+1} - x_i$ being the uniform spacing. By Taylor expanding $f(x)$ around x_i, we have:

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{f''(\xi)}{2}(x - x_i)^2, \quad \xi \in [x_i, x_{i+1}].$$

The linear spline $S_n(x)$ is constructed as:

$$S_n(x) = f(x_i) + \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}(x - x_i).$$

Subtracting $S_n(x)$ from $f(x)$, the interpolation error becomes:

$$e_n(x) = f(x) - S_n(x) = \frac{f''(\xi)}{2}(x - x_i)(x_{i+1} - x).$$

6.1.2 Error Bound and Convergence Rate

The maximum error occurs at the midpoint $x = (x_i + x_{i+1})/2$, where:

$$e_n(x) \propto h_n^2 \cdot \max_{\xi \in [x_i, x_{i+1}]} |f''(\xi)|.$$

Since h_n is proportional to $1/n$ for a uniform partition, we conclude:

$$e_n(x) \propto \frac{1}{n^2}.$$

6.1.3 Verification of Q-order Convergence

Using the definition of Q-order convergence, we check whether the sequence of errors satisfies:

$$\lim_{n \to \infty} \frac{|e_{n+1}(x)|}{|e_n(x)|^p} = c > 0.$$

For $p = 2$, observe that doubling the number of subintervals halves the spacing h_n:

$$\frac{|e_{n+1}(x)|}{|e_n(x)|^2} = \frac{h_{n+1}^2}{h_n^4} \propto \frac{(1/(2n))^2}{(1/n^2)^2} = c > 0.$$

Thus, the convergence order of linear piecewise-polynomial splines is $p = 2$.

6.2 Quadratic Splines

6.2.1 Error Expansion Using Taylor Series

Consider the target function $f(x)$ to be sufficiently smooth, with derivatives up to order three. Let x_i, x_{i+1}, and x_{i+2} denote three consecutive nodes in the partition of the interval, with $h_n = x_{i+1} - x_i$ being the uniform spacing. By Taylor expanding $f(x)$ around x_i, we have:

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{f''(x_i)}{2}(x - x_i)^2 + \frac{f^{(3)}(\xi)}{6}(x - x_i)^3, \quad \xi \in [x_i, x_{i+1}].$$

The quadratic spline $S_n(x)$ is constructed to match $f(x)$ at three points, typically x_i, x_{i+1}, and x_{i+2}. Subtracting $S_n(x)$ from $f(x)$, the interpolation error depends on the highest-order term not captured by the quadratic polynomial:

$$e_n(x) = f(x) - S_n(x) \propto \frac{f^{(3)}(\xi)}{6}(x - x_i)(x - x_{i+1})(x - x_{i+2}).$$

6.2.2 Error Bound and Convergence Rate

Assuming uniform spacing, $h_n = x_{i+1} - x_i$, the maximum error over each subinterval scales as:

$$e_n(x) \propto h_n^3 \cdot \max_{\xi \in [x_i, x_{i+2}]} |f^{(3)}(\xi)|.$$

Since h_n is proportional to $1/n$ for a uniform partition, we conclude:

$$e_n(x) \propto \frac{1}{n^3}.$$

6.2.3 Verification of Q-order Convergence

Using the definition of Q-order convergence, we check whether the sequence of errors satisfies:

$$\lim_{n\to\infty} \frac{|e_{n+1}(x)|}{|e_n(x)|^p} = c > 0.$$

For $p = 3$, observe that doubling the number of subintervals halves the spacing h_n:

$$\frac{|e_{n+1}(x)|}{|e_n(x)|^3} = \frac{h_{n+1}^3}{h_n^9} \propto \frac{(1/(2n))^3}{(1/n^3)^3} = c > 0.$$

Thus, the convergence order of quadratic piecewise-polynomial splines is $p = 3$.

6.3 Cubic Splines

6.3.1 Natural Boundary Conditions

The cubic splines typically satisfy three types of boundary conditions:

1. **Natural Boundary Condition**: The second derivative is zero at the endpoints:

$$S''(a) = 0, \quad S''(b) = 0,$$

where $[a, b]$ represents the interpolation domain.

2. These conditions ensure that the cubic spline is well-defined and exhibit smooth transitions across intervals while maintaining natural properties at the endpoints.

6.3.2 Natural Boundary Conditions for Cubic Splines

The cubic splines are commonly defined over intervals using cubic polynomials. To ensure smoothness and stability, we impose **natural boundary conditions**:

$$S''(a) = 0, \quad S''(b) = 0,$$

where $[a, b]$ represents the domain of interest. These conditions mean that at the interval endpoints, the second derivative of the cubic splines is zero. The choice of natural boundary conditions guarantees that the interpolation behaves in a physically reasonable and mathematically stable way near the boundaries.

6.3.3 Error Expansion without Natural Boundary Conditions

Let $f(x)$ be the target function, and let $S_n(x)$ represent the cubic spline interpolant that satisfies the natural boundary conditions. The error is given by:

$$e_n(x) = f(x) - S_n(x).$$

To examine how boundary conditions influence this error, we perform a Taylor series expansion of $f(x)$ near a typical node x_i.

Taylor Expansion of the Function
Using the Taylor expansion of $f(x)$ up to fourth order:

$$f(x) = f(x_i)+f'(x_i)(x-x_i)+\frac{f''(x_i)}{2!}(x-x_i)^2+\frac{f'''(x_i)}{3!}(x-x_i)^3+\frac{f^{(4)}(\xi)}{4!}(x-x_i)^4,$$

where $\xi \in [x_i, x_{i+1}]$, we see that the cubic spline will only capture the cubic behavior up to $f'''(x_i)$, and the remaining error comes from the fourth derivative term.

The local error can be expressed as:

$$e_n(x) \propto \frac{f^{(4)}(\xi)}{24}h_n^4,$$

where h_n is the uniform subinterval size.

However, this expansion is valid in the interior of the intervals. Near the endpoints a and b, the natural boundary conditions impact the construction of the cubic polynomial, which leads to changes in the effective convergence rate.

6.3.4 How Natural Boundary Conditions Affect the Convergence

Key Idea
The natural boundary conditions impose that:

$$S''(a) = 0 \quad \text{and} \quad S''(b) = 0.$$

These constraints lead to modifications in the cubic polynomial coefficients at the boundaries. This adjustment means that the local polynomial behavior near the endpoints can deviate from a simple interior interpolation, which influences the convergence properties.

1. Near the Boundaries

Near the interval endpoints, the imposed natural boundary conditions constrain the system of equations determining the cubic splines. As a result:

- The interpolation error behaves differently compared to the interior of the domain.

- The higher derivatives at the endpoints may not behave in a purely smooth way because the cubic polynomials are forced to satisfy the boundary conditions.

2. Convergence Analysis Near the Endpoints

At or near the endpoints, the Taylor series' higher-order terms affect how the overall error behaves. For instance:

- At interior points, the convergence order is still $p = 4$ (similar to the uniform interval analysis).

- Near the endpoints, this convergence may degrade to lower orders due to the enforced constraints imposed by the boundary conditions.

Error Bound Comparison
Let's compute the error near endpoints versus the interior:

1. **Interior regions**:

$$e_n(x) \propto h_n^4,$$

leading to optimal fourth-order convergence.

2. **Near the endpoints**: The constraints $S''(a) = 0$ and $S''(b) = 0$ cause the cubic spline to behave in a manner that only achieves third-order convergence:

$$e_n(x) \propto h_n^3.$$

Thus, the presence of natural boundary conditions reduces the convergence order near the endpoints from 4 to 3.

6.3.5 Numerical Implications of Boundary Effects

The change in convergence rate implies that:

1. In regions close to the endpoints, interpolation errors will scale as h_n^3.
2. For interior points (away from endpoints), the optimal convergence order of $p = 4$ is maintained.

6.3.6 Conclusion: Summary of Findings

The natural boundary conditions impose constraints that affect the smoothness and behavior of the cubic spline at the interval endpoints. This leads to:

- **Fourth-order convergence** ($p = 4$) in the interior of the domain.
- **Third-order convergence** ($p = 3$) near the endpoints.

This analysis highlights that natural boundary conditions, while stabilizing the cubic spline interpolation, introduce regions of lower convergence rate near the endpoints.

7 Conclusion

The project has provided valuable insights into the practical application of splines in numerical analysis and computer graphics. The implementation of the B-spline formula and the comparison between pp-Form and B-Form splines have been particularly enlightening. The choice of boundary condition should be based on the specific requirements of the application.