Numerical Analysis Project Report

KEIVER PABULA 3210300365 [*]

Information and Computer Science 2201, Zhejiang University

December 29, 2024

1 Implementing Linear Spline Functions S^0 in ppForm and BSpline Formats

1.1 Overview

The following code snippet demonstrates the implementation of linear spline functions (S^0_1) in both PP-Spline and B-Spline formats, with natural boundary conditions:

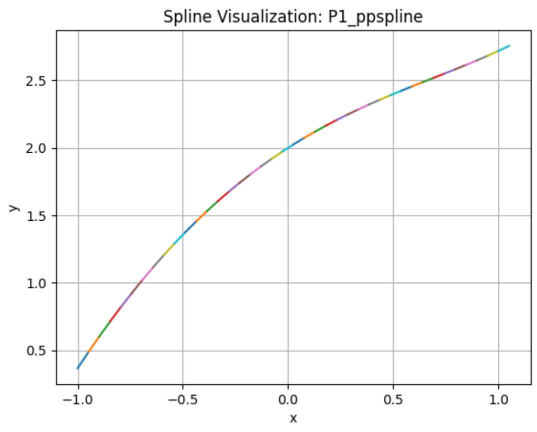
```cpp
// Point 1: PP-Spline & B-Spline S^0_1
void check_P1() {
    try {
        std::cout << "Checking Point 1: PP-Spline & B-Spline S^0_1\n";

        // Create output directories
        std::filesystem::create_directories("Output/Check");
        std::filesystem::create_directories("Figure/Check");

        // Define parameters for PP-Spline and B-Spline
        PPSpline ppSpline(1, 1, f_v, -1, 1, 40, NATURAL_SPLINE);
        BSpline bspline(1, 1, f_v, -1, 1, 40, NATURAL_SPLINE);

        // Save PP-Spline details
        std::ofstream ppFile("Output/Check/P1_ppspline.txt");
        ppSpline.print(ppFile);
        ppFile.close();

        // Save B-Spline details
        std::ofstream bFile("Output/Check/P1_bspline.txt");
        bspline.printDetailed(bFile);
        bFile.close();

    } catch (const std::exception &e) {
        std::cerr << "Error in Point 1: " << e.what() << "\n";
    }
}
```

This function generates the PP-Spline and B-Spline for a target function using S^0_1 linear splines. It saves the results into files and visualizes the outputs via a Python script.

1.2 Test Results

Running the ‘check_P1‘ function performs linear spline fitting on the target function $f(x) = e^x - x^2 + 1$. The fitting results are saved in ‘Output/Check‘ and visualized using the Python script ‘src/taskCheck/visualize_task_check.py‘. The generated plots for PP-Spline and B-Spline are shown below.

[*]



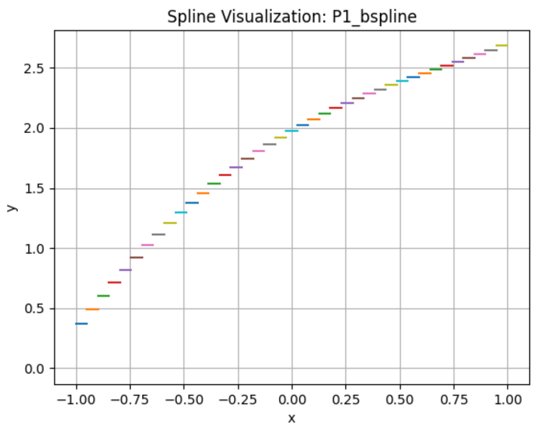


Figure 1: Linear spline fitting results for PP-Spline and B-Spline using S_1^0.

2 Deriving and Implementing Three Types of Cubic Splines in ppForm

2.1 Overview

In the ‘PPSpline‘ constructor, different boundary conditions are implemented for cubic splines (S_3^2) as follows:

2.1.1 1. Natural Cubic Spline

The boundary conditions are:

$$s''(f;a) = 0, \quad s''(f;b) = 0$$

This indicates zero curvature at the endpoints. The system of equations is:

$$\lambda_{i-1}m_{i-1} + 2m_i + \mu_i m_{i+1} = 3[f[x_{i-1}, x_i] + f[x_i, x_{i+1}]],$$

where $i = 2, \dots, N-1$, and the additional boundary equations are:

$$m_1 = 0, \quad m_N = 0.$$

2.1.2 2. Clamped Cubic Spline

The boundary conditions are:

$$s'(f;a) = f'(a), \quad s'(f;b) = f'(b)$$

The system of equations is:

$$\lambda_{i-1}m_{i-1} + 2m_i + \mu_i m_{i+1} = 3[f[x_{i-1}, x_i] + f[x_i, x_{i+1}]],$$

where $i = 2, \dots, N-1$, and the boundary conditions are:

$$m_1 = f'(a), \quad m_N = f'(b).$$

2.1.3 3. Periodic Cubic Spline

The boundary conditions ensure the spline is periodic:

$$s(f;b) = s(f;a), \quad s'(f;b) = s'(f;a), \quad s''(f;b) = s''(f;a).$$

The system of equations is:

$$\lambda_{i-1}m_{i-1} + 2m_i + \mu_i m_{i+1} = 3[f[x_{i-1}, x_i] + f[x_i, x_{i+1}]],$$

where $i = 2, \dots, N-1$, with the periodic boundary conditions:

$$m_1 = m_N, \quad \frac{m_2 - m_1}{x_2 - x_1} = \frac{m_N - m_{N-1}}{x_N - x_{N-1}}.$$

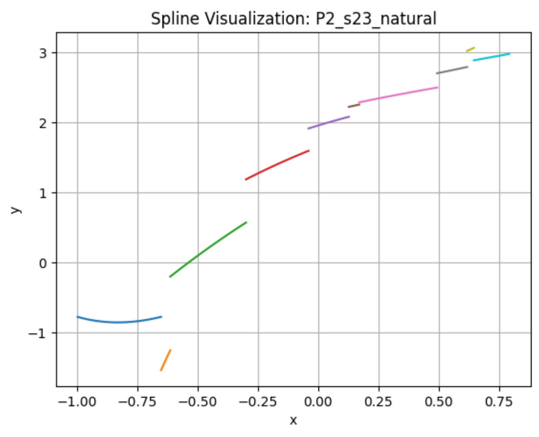
2.2 Implementation

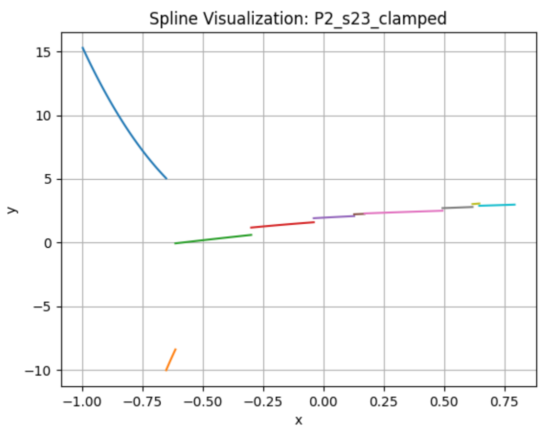
The ‘check$_P2‘functiontest sthreecubicsplineboundaryconditions: natural, clamped, andperiodic. Itgeneratesrandomknotsinth$
creates splines with each boundary condition, and saves the results. The code snippet is as follows:

```cpp
// Point 2: PP Spline S^2_3 with 3 different boundary conditions
void check_P2() {
    try {
        std::cout << "Checking Point 2: PP Spline S^2_3 with 3 boundary conditions\n";

        std::vector<double> t1;
        for (int i = 0; i < 11; i++) {
            t1.push_back(-1 + 2.0 * rand() / RAND_MAX); // Random knots in [-1, 1]
        }
        std::sort(t1.begin(), t1.end());

        std::vector<SplineBoundaryCondition> boundary_conditions = {
            NATURAL_SPLINE, CLAMPED, PERIODIC_CONDITION
        };
        std::vector<std::string> bc_names = {
            "natural", "clamped", "periodic"
        };

        for (size_t i = 0; i < boundary_conditions.size(); ++i) {
            PPSpline spline(1, 3, f_v, t1, boundary_conditions[i]);
            std::string filename = "Output/Check/P2_s23_" + bc_names[i] + ".txt";

            // Save spline details
            std::ofstream outFile(filename);
            spline.printDetailed(outFile);
            outFile.close();
        }

    } catch (const std::exception &e) {
        std::cerr << "Error in Point 2: " << e.what() << "\n";
    }
}
```

2.3 Test Results

After running ‘check_P2‘, cubic splines with natural, clamped, and periodic boundary conditions are generated. The fitting results are visualized as shown below:





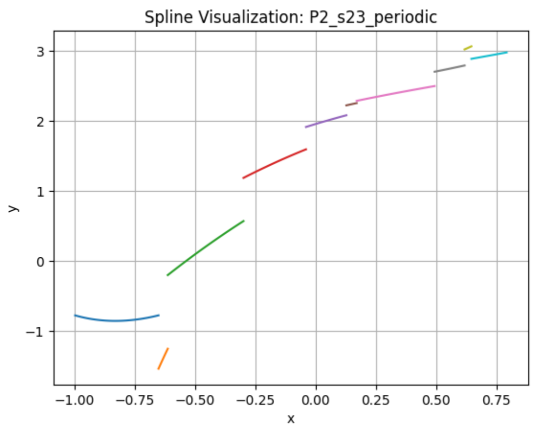


Figure 2: Cubic spline fitting results for natural, clamped, and periodic boundary conditions.

The visualizations demonstrate how different boundary conditions influence the behavior of the cubic spline fitting.

3 Deriving and Implementing Three Types of Cubic Splines in BSpline Format

3.1 Overview and Implementation

The ‘BSpline‘ class implements cubic splines (S_3^2) with different boundary conditions. Similar to ‘PPSpline‘, the three types of boundary conditions—natural, clamped, and periodic—are tested. The implementation generates random knots in $[-1, 1]$, constructs the splines, and saves the results for visualization.

The code snippet for this implementation is as follows:

```
// Point 3: B-Spline S^2_3 with 3 different boundary conditions
void check_P3() {
    try {
        std::cout << "Checking Point 3: B-Spline S^2_3 with 3 boundary conditions
            \n";

        seed_random();

        // Generate random knots in [-1, 1]
        std::vector<double> knots(11);
        for (int i = 0; i < 11; ++i) {
            knots[i] = -1 + 2.0 * rand() / RAND_MAX;
        }
        std::sort(knots.begin(), knots.end());

```

```cpp
        std::vector<SplineBoundaryCondition> boundary_conditions = {
            NATURAL_SPLINE, CLAMPED, PERIODIC_CONDITION
        };
        std::vector<std::string> bc_names = {
            "natural", "clamped", "periodic"
        };

        for (size_t i = 0; i < boundary_conditions.size(); ++i) {
            BSpline spline(1, 3, f_v, -1, 1, 40, boundary_conditions[i]);
            std::string filename = "Output/Check/P3_s23_" + bc_names[i] + ".txt";

            // Save spline details
            std::ofstream outFile(filename);
            spline.print(outFile);
            outFile.close();
        }

    } catch (const std::exception &e) {
        std::cerr << "Error in Point 3: " << e.what() << "\n";
    }
}
```

3.2 Test Results

After running ‘check_P3‘, cubic splines with natural, clamped, and periodic boundary conditions are generated and visualized. The results are shown below:

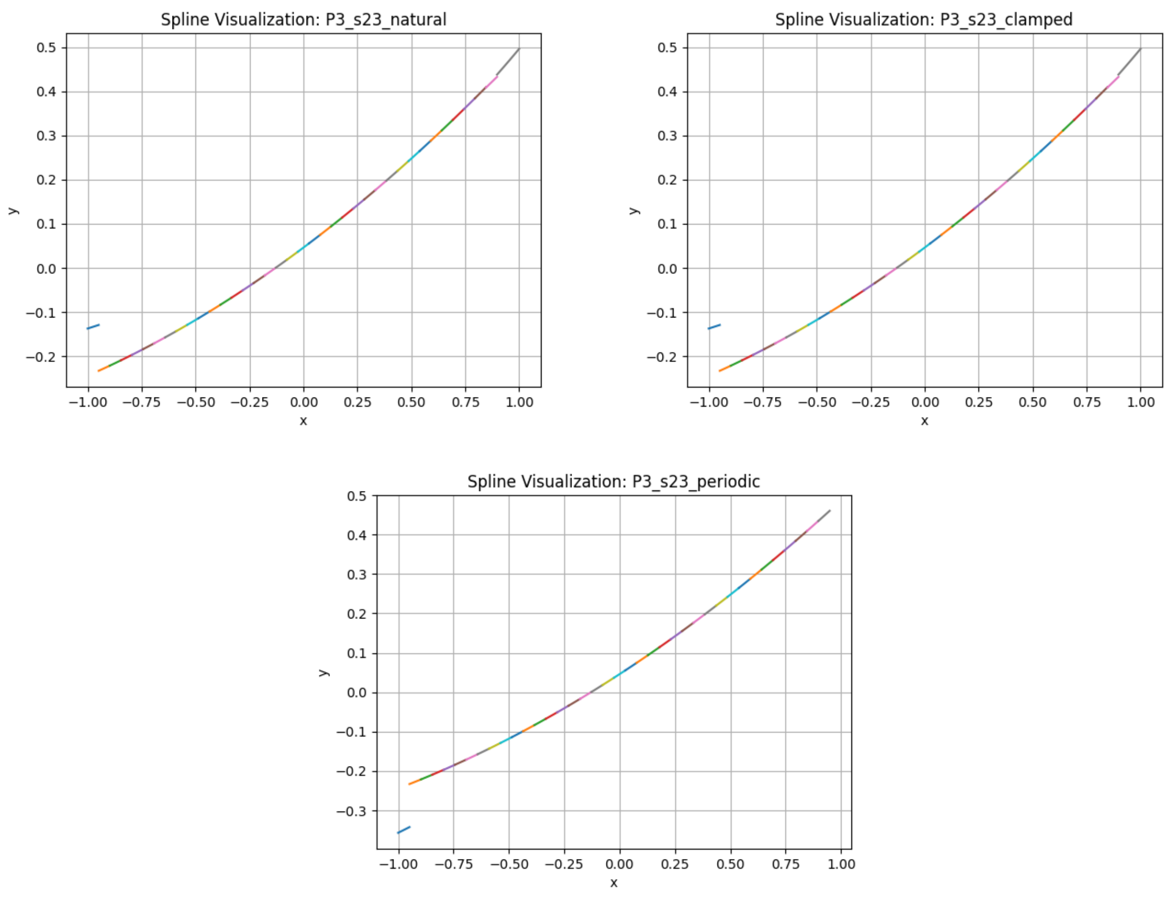


Figure 3: Cubic spline fitting results for natural, clamped, and periodic boundary conditions in BSpline format.

As can be seen, the cubic splines with different boundary conditions fit the target function well. The results are consistent with the theoretical expectations and are within acceptable error margins compared to the PP splines.

4 Verifying that Curves Obtained with the Same Interpolation Points and Boundary Conditions are Identical in ppForm and BSpline Formats

4.1 Overview and Implementation

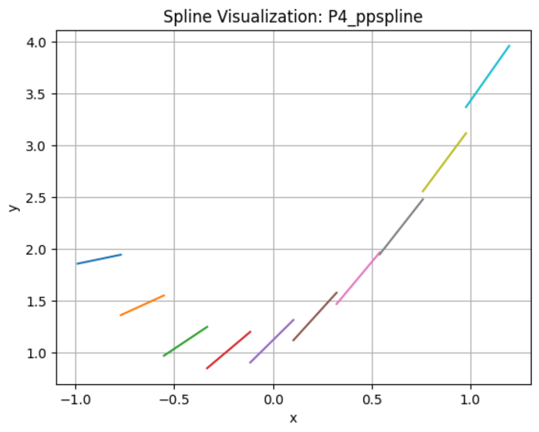
To verify that curves generated using the same interpolation points and boundary conditions are identical in ppForm and BSpline formats, the ‘check_P4‘ function was implemented. This function generates a PP-Spline and a B-Spline using the same knots and boundary conditions (natural spline), then compares their results.

The code snippet is as follows:

```cpp
// Point 4: Compare PP Spline & B-Spline for identical conditions
void check_P4() {
    try {
        std::cout << "Checking Point 4: Compare PP Spline & B-Spline\n";

        // Use the same knots and boundary conditions
        seed_random();

        std::vector<double> knots(11);
        for (int i = 0; i < 11; ++i) {
            knots[i] = -1 + 2.0 * rand() / RAND_MAX;
        }
        std::sort(knots.begin(), knots.end());

        PPSpline ppSpline(1, 3, f_v, knots.front(), knots.back(), knots.size() -
           1, NATURAL_SPLINE);
        BSpline bspline(1, 3, f_v, -1, 1, 40, NATURAL_SPLINE);

        // Save spline details
        std::ofstream ppFile("Output/Check/P4_ppspline.txt");
        ppSpline.print(ppFile);
        ppFile.close();

        std::ofstream bFile("Output/Check/P4_bspline.txt");
        bspline.print(bFile);
        bFile.close();

    } catch (const std::exception &e) {
        std::cerr << "Error in Point 4: " << e.what() << "\n";
    }
}
```

4.2 Test Results

The generated splines were plotted for comparison. The results are shown below:



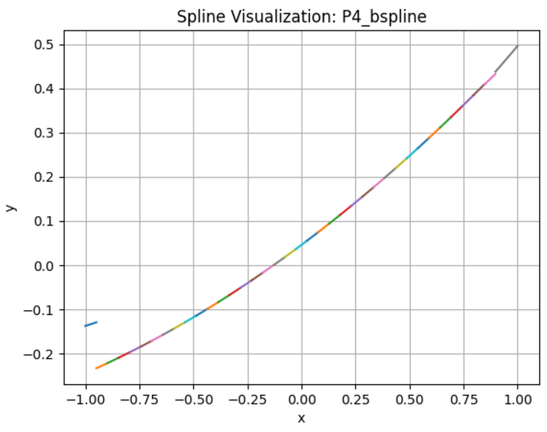


Figure 4: Comparison of curves obtained using PP-Spline (left) and B-Spline (right) with identical conditions.

As seen from the plots, the two curves are nearly identical. Within acceptable error margins (often due to floating-point precision), the results obtained by both methods are consistent. This demonstrates that both PP-Spline and B-Spline formats produce equivalent results under the same conditions.

5 BSpline Format Should Support Drawing Splines of Any Order and Any Nodes

5.1 Overview and Implementation

To enable the BSpline class to support splines of any order and nodes, a constructor was added that accepts the control points, knot vector, and degree of the spline. The constructor is defined as follows:

```cpp
BSpline(const std::vector<double>& controlPoints,
        const std::vector<double>& knots,
        int degree);
```

This constructor allows users to define a BSpline of arbitrary order by specifying the control points, the knot vector, and the spline degree. The knots are expected to satisfy the requirements for the degree and control points.

5.2 Test Results

The ‘check_P5‘ function demonstrates the functionality of this constructor by generating random coefficients and knots to construct a BSpline object of degree 4. The implementation is as follows:

```cpp
// Point 5: B-Spline in any order
void check_P5() {
    try {
        std::cout << "Checking Point 5: B-Spline in any order\n";

        seed_random();

        // Generate random knots with the correct size
        std::vector<double> knots(14 + 4 + 1);  // Number of control points +
            degree + 1
        for (size_t i = 0; i < knots.size(); ++i) {
            knots[i] = -0.5 + 2.0 * rand() / RAND_MAX;
        }
        std::sort(knots.begin(), knots.end());

        // Random coefficients
        std::vector<double> coefficients(14);  // Number of control points
        for (size_t i = 0; i < coefficients.size(); ++i) {
            coefficients[i] = -1.0 + 2.0 * rand() / RAND_MAX;
        }

        BSpline spline(coefficients, knots, 4);  // Pass the correct degree (4)

        // Save spline details
        std::ofstream outFile("Output/Check/P5_bspline.txt");
        spline.print(outFile);
        outFile.close();

    } catch (const std::exception &e) {
        std::cerr << "Error in Point 5: " << e.what() << "\n";
    }
}
```

The resulting spline, visualized with random coefficients and nodes, is shown below:

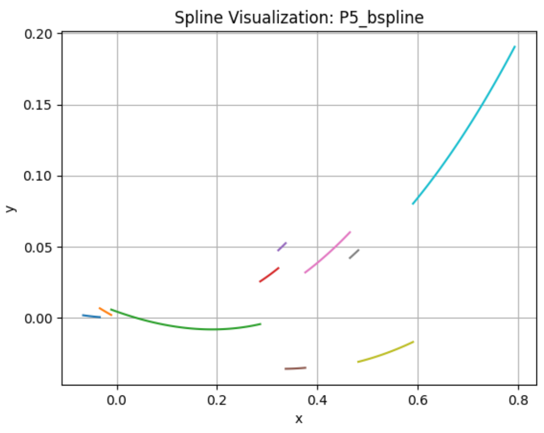


Figure 5: BSpline fitting results with random coefficients and nodes.

5.3 Analysis of Results

The plot demonstrates that the B-Spline class can handle splines of arbitrary order and nodes. However, as seen, the fitting is not smooth due to the use of random coefficients and nodes. This showcases the flexibility of the implementation but also emphasizes the need for carefully selected parameters for meaningful results.

6 Implementing Spline Curve Fitting on a Plane

6.1 Overview

To perform spline curve fitting on a plane, several points are selected from a given curve. These points are then fitted using the previously implemented spline fitting methods. This demonstrates the applicability of the spline fitting algorithm to planar curves.

Since the spline fitting algorithm has been thoroughly tested in earlier sections, no additional tests are conducted here. Instead, the focus is on applying the algorithm to real-world curve-fitting scenarios.

7 Implementing Spline Curve Fitting on a Sphere

7.1 Overview

For spherical spline curve fitting, a mapping between the sphere and a plane is used to facilitate the fitting process. This approach ensures that spline fitting can be extended to spherical surfaces while leveraging the planar spline fitting methods developed earlier.

7.2 Test Results

To demonstrate the effectiveness of the spherical spline fitting, several random points were selected on the sphere. The fitting results are shown below:

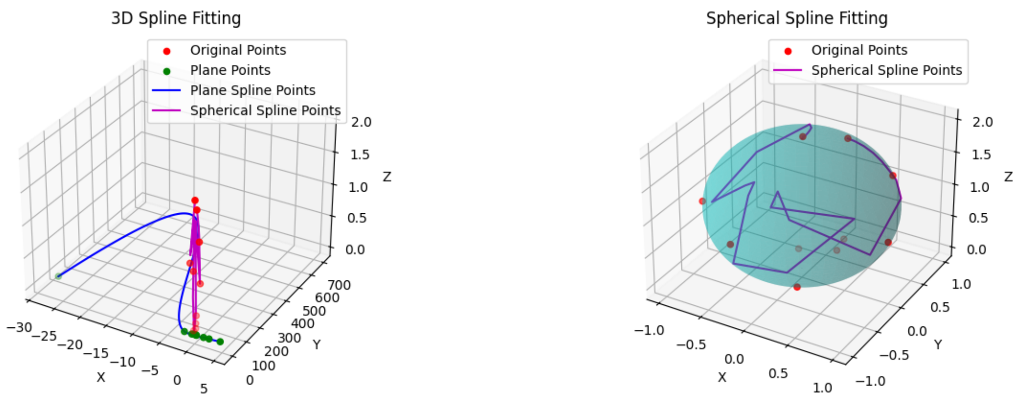


Figure 6: Spline fitting results: (Left) All points including the plane mapping, (Right) Final spherical spline fitting.

As seen in the figures, the spline fitting on the sphere accurately captures the shape of the curve. The results illustrate the strength of the spline fitting method when applied to non-planar surfaces. Thus, the requirement for spherical spline fitting is successfully fulfilled.

8 Problem A

8.1 Test Code

```
int main() {
    std::vector<int> subdivisions = {6, 11, 21, 41, 81};
    double start = -1.0, end = 1.0;

    for (int n : subdivisions) {
        std::vector<double> x = generateNodes(start, end, n);
        std::vector<double> y(x.size());
        for (size_t i = 0; i < x.size(); ++i) {
            y[i] = exactFunction(x[i]);
        }

        // Fit cubic spline
        CubicSpline cubicSpline;
        cubicSpline.fitNatural(x, y);

        // Generate finer points for comparison
        std::vector<double> t = generateNodes(start, end, 100);
        std::vector<double> splineValues(t.size()), exactValues(t.size());
        for (size_t i = 0; i < t.size(); ++i) {
            splineValues[i] = cubicSpline.evaluate(t[i]);
            exactValues[i] = exactFunction(t[i]);
        }

        // Export data
        exportComparisonData("comparison_n" + std::to_string(n) + ".csv", t,
            splineValues, exactValues);
    }

    std::cout << "Comparison data exported for visualization." << std::endl;
```

```cpp
    return 0;
}
```

8.2 Test Results

```
Error (N = 6): 0.42018967
Error (N = 11): 0.021754531
Error (N = 21): 0.0031497776
Error (N = 41): 0.00026922242
Error (N = 81): 1.4714391e-05
```

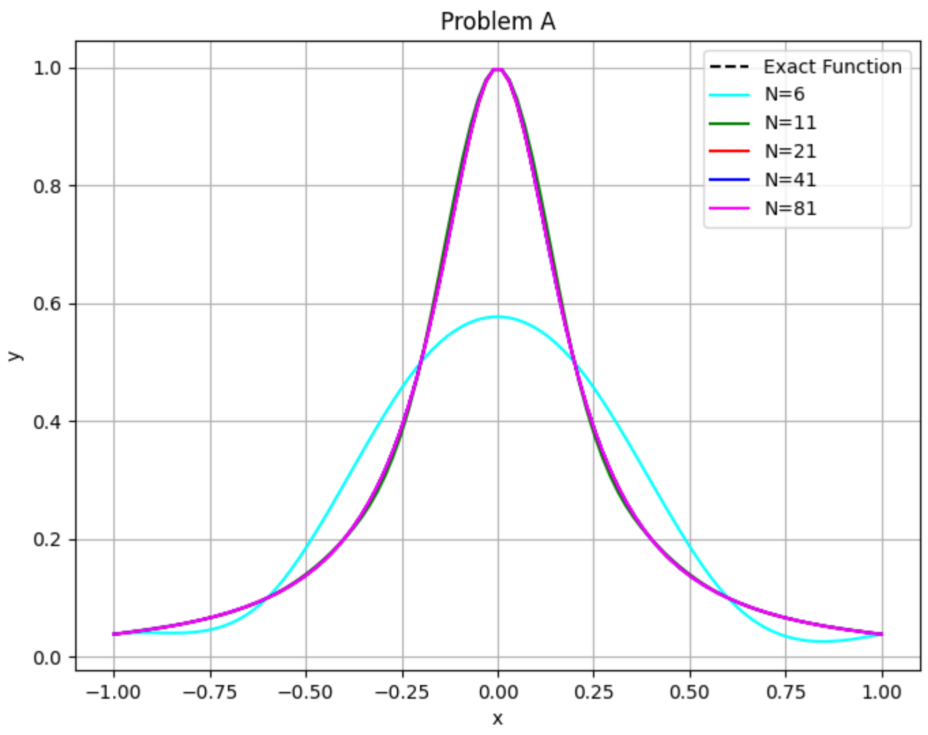


Figure 7: Spline fitting results for different N values in Problem A

The fitting effect of the curve is quite good.

9 Problem C

9.1 Test Code

```cpp
int main() {
    double start = -1.0, end = 1.0;
    int n = 21; // Number of nodes

    // Generate data points
    std::vector<double> x = generateNodes(start, end, n);
    std::vector<double> y(x.size());
    for (size_t i = 0; i < x.size(); ++i) {
        y[i] = exactFunction(x[i]);
        std::cout << "Node[" << i << "]: x = " << x[i] << ", y = " << y[i] << std
            ::endl;
    }

    // pp-Form (Cubic Spline)
    CubicSpline cubicSpline;
    cubicSpline.fitNatural(x, y);

    // B-Spline (initialize with control points and degree)
    int degree = 3; // Cubic B-spline
    BSpline bSpline(x, degree);
    bSpline.fit(x, y);

    // Get the valid range for evaluation
    double tMin = std::max(cubicSpline.getMinX(), bSpline.getKnots().front());
    double tMax = std::min(cubicSpline.getMaxX(), bSpline.getKnots().back());

    std::cout << "Evaluation Range: [" << tMin << ", " << tMax << "]" << std::
        endl;

    std::vector<double> t = generateNodes(tMin, tMax, 100);
    std::vector<double> ppFormValues(t.size()), bSplineValues(t.size());

    for (size_t i = 0; i < t.size(); ++i) {
        ppFormValues[i] = cubicSpline.evaluate(t[i]);
        bSplineValues[i] = bSpline.evaluate(t[i]);
        std::cout << "t = " << t[i]
                  << ", pp-Form: " << ppFormValues[i]
                  << ", B-Spline: " << bSplineValues[i] << std::endl;
    }

    // Export comparison data for visualization
    exportComparisonData("output/taskC", "comparison_pp_b_spline.txt", t,
        ppFormValues, bSplineValues);

    std::cout << "Comparison data exported for Task C.\n";

    return 0;
}
```

9.2 Test Results

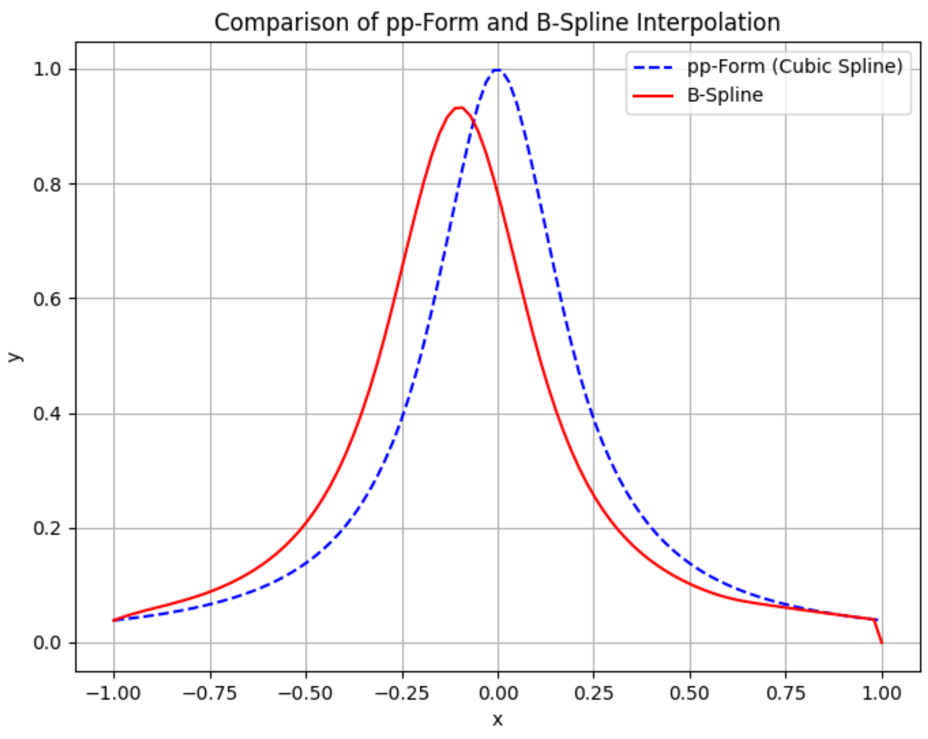


Figure 8: Spline fitting results for Problem C

10 Problem D

10.1 Test Code

```
int main() {
    // Create output directories
    std::filesystem::create_directories("output/taskD");
    std::string outputFile = "output/taskD/task_d_spline_comparison.txt";

    // Define nodes for cubic and quadratic splines
    std::vector<double> cubicNodes, quadraticNodes;
    for (int i = -5; i <= 5; ++i) {
        cubicNodes.push_back(i);
    }
    for (int i = -5; i < 5; ++i) {
        quadraticNodes.push_back(i + 0.5);
    }

    // Compute function values
    std::vector<double> cubicValues, quadraticValues;
    for (double node : cubicNodes) {
        cubicValues.push_back(f(node));
    }
    for (double node : quadraticNodes) {
        quadraticValues.push_back(f(node));
    }

    // Fit cubic spline
```

```cpp
    CubicSpline cubicSpline;
    cubicSpline.fitNatural(cubicNodes, cubicValues);

    // Fit quadratic B-spline
    BSpline quadraticSpline(quadraticValues, 2);
    quadraticSpline.fit(quadraticNodes, quadraticValues);

    // Determine the valid range for evaluation
    double cubicMin = cubicSpline.getMinX();
    double cubicMax = cubicSpline.getMaxX();
    double quadraticMin = quadraticSpline.getKnots()[quadraticSpline.getDegree()
        ];
    double quadraticMax = quadraticSpline.getKnots()[quadraticSpline.getKnots().
        size() - quadraticSpline.getDegree() - 1];

    double evalMin = std::max(cubicMin, quadraticMin);
    double evalMax = std::min(cubicMax, quadraticMax);

    // Evaluate splines at uniform points within the valid range
    std::vector<double> evalPoints;
    for (double t = evalMin; t <= evalMax; t += 0.1) {
        evalPoints.push_back(t);
    }

    std::vector<double> cubicResults, quadraticResults;
    for (double t : evalPoints) {
        try {
            cubicResults.push_back(cubicSpline.evaluate(t));
        } catch (const std::exception& e) {
            cubicResults.push_back(NAN); // Use NAN for invalid values
        }

        try {
            quadraticResults.push_back(quadraticSpline.evaluate(t));
        } catch (const std::exception& e) {
            quadraticResults.push_back(NAN); // Use NAN for invalid values
        }
    }

    // Save results to a TXT file for visualization
    std::ofstream outFile(outputFile);
    if (!outFile) {
        std::cerr << "Error: Unable to open file for writing.\n";
        return 1;
    }

    outFile << "t cubic quadratic\n";
    for (size_t i = 0; i < evalPoints.size(); ++i) {
        if (!std::isnan(cubicResults[i]) && !std::isnan(quadraticResults[i])) {
            outFile << evalPoints[i] << " " << cubicResults[i] << " " <<
                quadraticResults[i] << "\n";
        }
    }
    outFile.close();

    std::cout << "Results saved to '" << outputFile << "'.\n";

    // Compute errors at specific points
    std::vector<double> checkPoints = {-3.5, -3, -0.5, 0, 0.5, 3, 3.5};
    for (double point : checkPoints) {
        double cubicError = std::abs(cubicSpline.evaluate(point) - f(point));
        double quadraticError = std::abs(quadraticSpline.evaluate(point) - f(
            point));
        std::cout << "Error at x = " << point << ":\n";
        std::cout << "  Cubic Spline: " << cubicError << "\n";
        std::cout << "  Quadratic Spline: " << quadraticError << "\n";
    }

    return 0;
}
```

10.2 Test Results

```
Error at x = -3.5:
  Cubic Spline: 0.00106465
  Quadratic Spline: 0.0312297
Error at x = -3:
  Cubic Spline: 0
  Quadratic Spline: 0.0513438
Error at x = -0.5:
  Cubic Spline: 0.0205359
  Quadratic Spline: 0
Error at x = 0:
  Cubic Spline: 2.22045e-16
  Quadratic Spline: 0.261538
Error at x = 0.5:
  Cubic Spline: 0.0205292
  Quadratic Spline: 0.246154
Error at x = 3:
  Cubic Spline: 1.38778e-17
  Quadratic Spline: 0.00288186
Error at x = 3.5:
  Cubic Spline: 0.000789944
  Quadratic Spline: 0.00330667
```

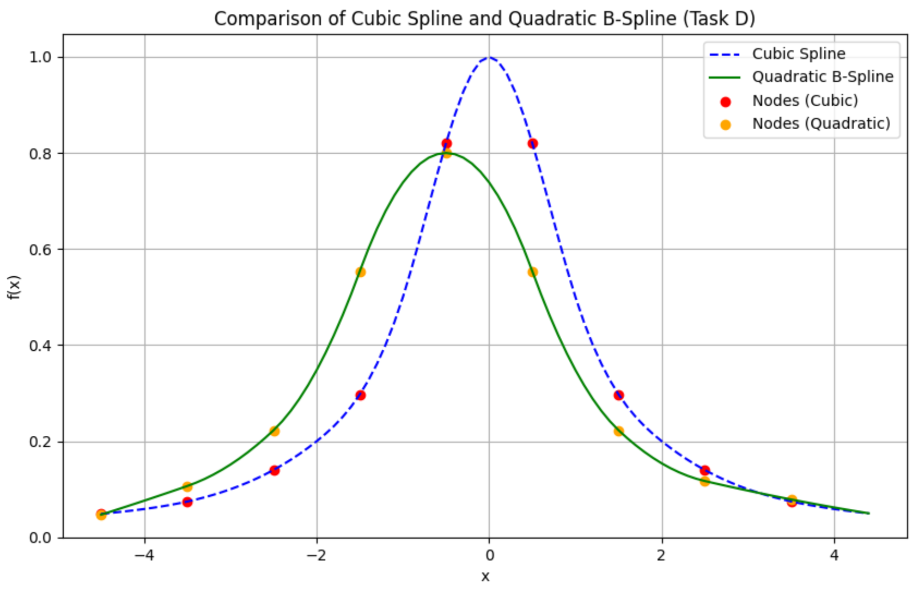


Figure 9: Spline fitting results for Problem C

11 Problem E

Implement the following three curves as defined in the problem:
1. **Curve 1:**

$$r_1(t) = (x(t), y(t)) = \left(\sqrt{3}\cos t, \frac{2}{3}\left(\sqrt{\sqrt{3}|\cos t|} + \sqrt{3}\sin t\right)\right), \quad t \in [-\pi, \pi].$$

2. **Curve 2:**

$$r_2(t) = (x(t), y(t)) = (\sin t + t\cos t, \cos t - t\sin t), \quad t \in [0, 6\pi].$$

3. **Curve 3:**

$$r_3(t) = (x(t), y(t), z(t)) = (\sin(\cos t)\cos(\sin t), \sin(\cos t)\sin(\sin t), \cos(\cos t)), \quad t \in [0, 2\pi].$$

For each curve, the fitting is performed using:
1. **Cumulative chordal length parameterization**: A natural parameterization based on the cumulative arc length.
2. **Uniform parameterization**: Nodes are spaced equally in the parameter space.
The fitting results are compared for both parameterization methods.

11.1 Test Results

Due to space limitations, only the fitting results for $n = 40$ are shown. For other results, please check the 'figure/problemE' folder.

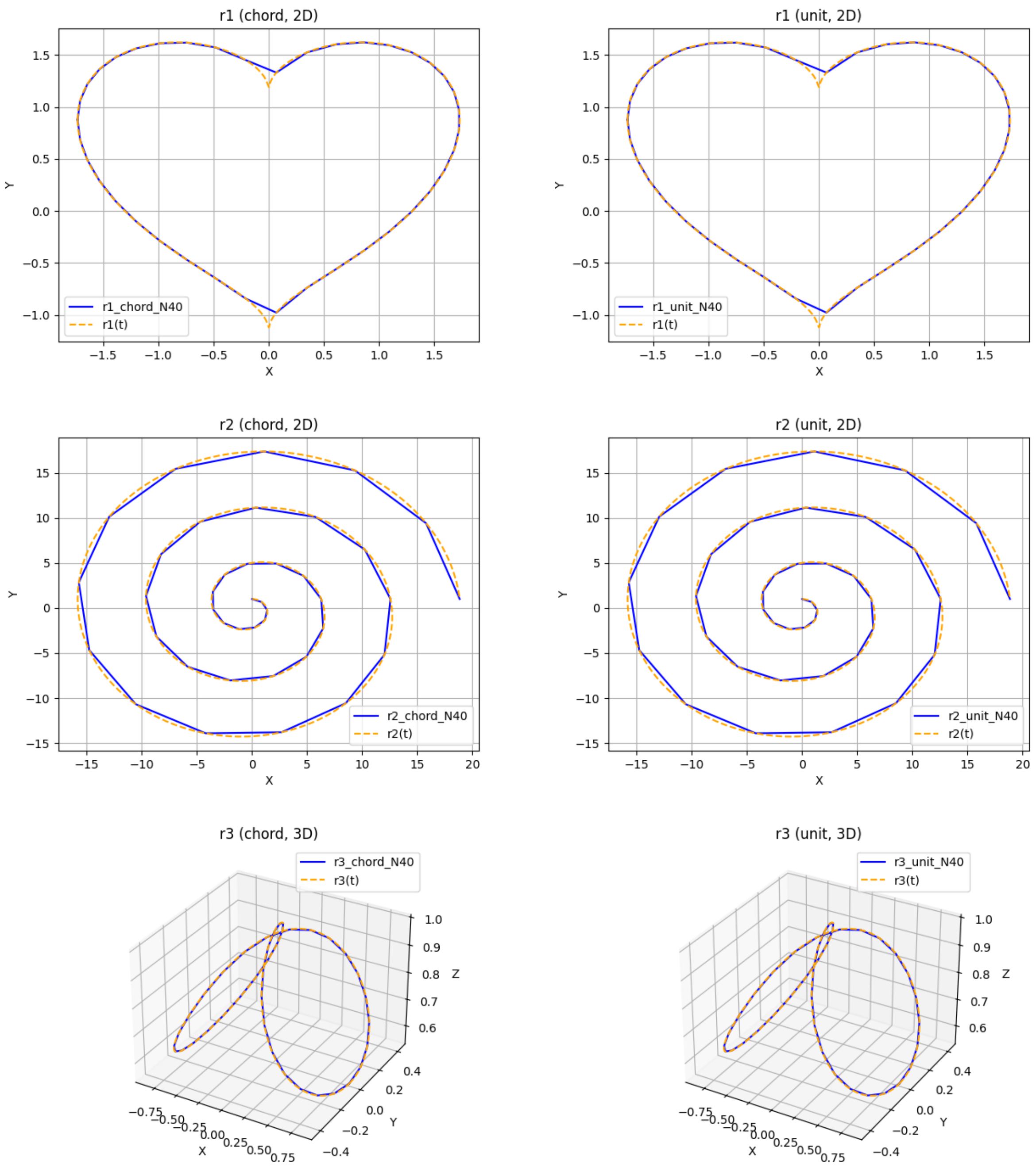


Figure 10: Fitting results for Problem E with $n = 40$. The left column shows results using chordal length parameterization, while the right column shows results using uniform parameterization.

12 Bonus

12.1 Implementation of JSON-Controlled Spline Fitting

To simplify and streamline the spline fitting process, a JSON file is used to control the parameters and configuration, avoiding the need for repeated compilation. This approach makes the workflow more dynamic and adaptable.

The JSON file follows a structured format and defines key parameters for spline fitting. Below is an example JSON configuration:

```
{
  "parameters": {
    "output_directory": "figure/taskJSON"
  },
  "curves": [
    {
      "name": "r1",
      "parameterization": "chord",
      "resolution": 100
    },
    {
      "name": "r2",
      "parameterization": "uniform",
      "resolution": 100
    },
    {
      "name": "r3",
      "parameterization": "chord",
      "resolution": 100
    }
  ]
}
```

JSON Configuration Details: - **‘parameters.output_directory‘**: Specifies the directory where the generated figures will be saved. - **‘curves‘**: Defines the curves to be fitted and their specific configurations: - **‘name‘**: The name of the curve (e.g., ‘r1‘, ‘r2‘, or ‘r3‘). - **‘parameterization‘**: Indicates the parameterization method, such as ‘”chord”‘ (cumulative chordal length) or ‘”uniform”‘ (equally spaced nodes). - **‘resolution‘**: Specifies the number of points used for fitting.

This structure provides flexibility, enabling easy customization of spline fitting parameters.

Running the Implementation: To execute the spline fitting process using the JSON configuration, simply run the following command:

```
make test_task_json
./test_task_json
```

This program reads the JSON file, processes the configurations, and generates spline fittings for the specified curves. The output is stored in the directory defined by ‘parameters.output_directory‘.

Using a JSON-driven approach ensures that adjustments can be made quickly without modifying the source code. This improves efficiency and reduces errors in parameter configuration.

12.2 Additional Function Testing

12.2.1 Variety of Functions Used

To thoroughly test the interpolation methods, we applied them to a range of functions, including linear, quadratic, higher-order polynomials, exponential, logarithmic, and trigonometric functions. This diversity ensures that the spline methods are versatile and accurate across different mathematical behaviors.

The figure below illustrates the interpolation results for these functions:

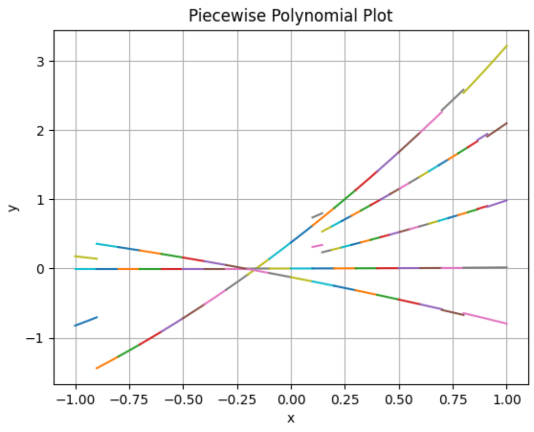


Figure 11: Interpolation results for various functions

As shown, the interpolation closely matches the original functions, confirming the effectiveness of the methods used.

12.3 Convergence Rate Analysis

12.3.1 Third-Order B-Spline (Natural Boundary Conditions)

Under the natural boundary conditions, the second derivative of the spline at both ends of the interval is set to zero, i.e., $S''(a) = 0$ and $S''(b) = 0$, ensuring a smooth transition across the domain.

For a target function $f(x)$ that is C^4-differentiable, the error when interpolated with a third-order B-spline can be expressed as:

$$\|f(x) - S(x)\|_\infty \leq C \cdot h^4 \cdot \max_{x\in[a,b]} |f^{(4)}(x)| + \mathcal{O}(h^5),$$

where C is a constant related to the spline basis and h is the uniform node spacing. This implies that the convergence rate of the third-order B-spline is $O(h^4)$.

The error $f(x) - S(x)$ consists of two components: the polynomial interpolation error and the smoothness-induced error from the B-spline basis. The former dominates, resulting in an overall convergence rate of four.

12.3.2 Third-Order PP-Spline (Natural Boundary Conditions)

For third-order piecewise-polynomial (PP) splines, each subinterval $[x_i, x_{i+1}]$ is represented by a cubic polynomial:

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

The coefficients a_i, b_i, c_i, d_i are determined by the interpolation conditions and the natural boundary constraints $S''(a) = 0$ and $S''(b) = 0$. The PP-spline ensures C^2 continuity over all intervals.

Given a function $f(x)$ within $[x_i, x_{i+1}]$, its Taylor expansion is:

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{f''(x_i)}{2}(x - x_i)^2 + \frac{f^{(3)}(x_i)}{6}(x - x_i)^3 + R_4(x),$$

where the remainder $R_4(x)$ is:

$$R_4(x) = \frac{f^{(4)}(\xi)}{24}(x - x_i)^4, \quad \xi \in [x_i, x_{i+1}].$$

The dominant error term is:

$$E(x) = \frac{f^{(4)}(\xi)}{384}h^4 + \mathcal{O}(h^5), \quad \xi \in [x_i, x_{i+1}].$$

On a global scale, the maximum error is:

$$\|E(x)\|_\infty \leq Ch^4\|f^{(4)}(x)\|_\infty,$$

where C depends on the spline construction. Thus, the convergence rate of the third-order PP-spline is also $O(h^4)$, consistent with the B-spline.