Project Test Report

Zhehao Chen 3220103172 *

December 8, 2024

Abstract

This project report delves into the implementation and testing of piecewise polynomial splines in both the pp-Form and B-Form, focusing on their application in curve fitting and self-intersection detection. The study includes a comprehensive analysis of linear, quadratic, and cubic splines under various boundary conditions, such as natural, clamped, and periodic. The report presents a detailed methodology for generating curve points, calculating distances between line segments, and detecting self-intersections using a threshold-based approach.

Contents

*Electronic address: 3220103172@zju.edu.cn

1 Test of pp-Form

1.1 Test Linear pp-Spline

See `testLinearpp.cpp` for the test code.

1.1.1 Methodology

The interpolation was performed using the following steps:

1. Define the function $f(x) = \sin(\pi x)$.
2. Select a set of knots within the interval $[-1, 1]$.
3. Compute the corresponding function values at these knots.
4. Create a LinearSpline object with the knots and values.

1.1.2 Results

The resulting plot of the linear spline interpolation is shown in Figure 7. The plot displays the original data points and the linear spline that connects them.

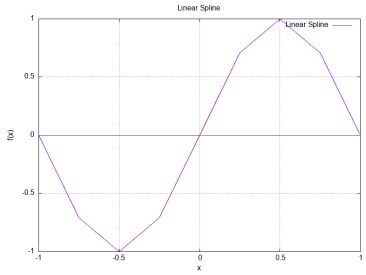

Figure 1: Linear Spline Interpolation of $f(x) = \sin(\pi x)$

1.1.3 Discussion

The plot in Figure 7 illustrates the linear segments that connect the data points. The spline appears to approximate the sine wave well within the given interval. However, the linear nature of the spline segments results in a piecewise linear curve that does not follow the smooth curvature of the sine function. This is an inherent limitation of linear spline interpolation when compared to higher-order splines or other interpolation methods.

1.2 Test Cubic pp-Spline

See `testCubicpp.cpp` for the test code.

1.2.1 Methodology

The interpolation was performed using the following steps:

1. Define the sine function and generate knots within the interval $[-1, 1]$.
2. Compute the function values at these knots.
3. Create cubic splines with natural, clamped, and periodic boundary conditions.

1.2.2 Results

The results of the cubic spline interpolation for the sine function with different boundary conditions are shown in Figures 14, 10, and 15. The plots display the spline and the knots for each boundary condition.

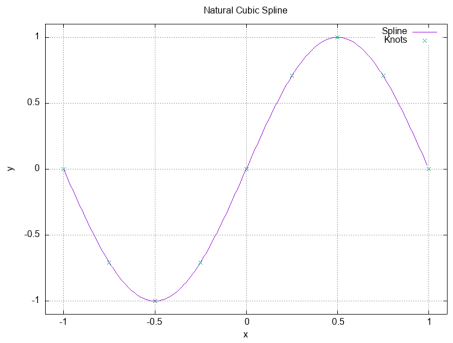

Figure 2: Natural Boundary Condition Spline

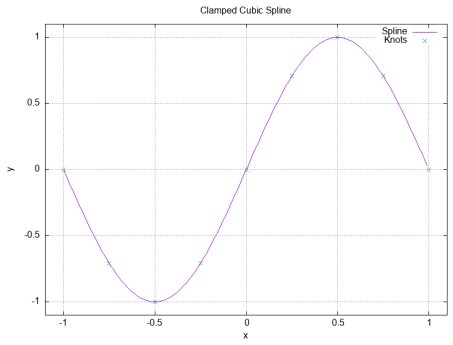

Figure 3: Clamped Boundary Condition Spline

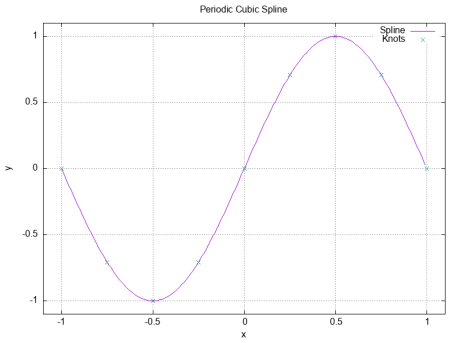

Figure 4: Periodic Boundary Condition Spline

1.2.3 Answer of Problem A

See `ProblemA.cpp` for the test code.

We solve the problem using the following steps:

1. Define the function $f(x) = \frac{1}{1+25x^2}$.
2. Generate evenly spaced nodes within the interval $[-1, 1]$.
3. Compute the function values at these nodes.
4. Create a CubicSpline (Natural Boundary Condition) object with natural boundary conditions.
5. Calculate the maximum and mean squared errors for each node count.

The results of the cubic spline interpolation for different node counts are shown in Figures 5 to 6. The plots display the exact function and the interpolated spline, with the node count increasing from 6 to 81.

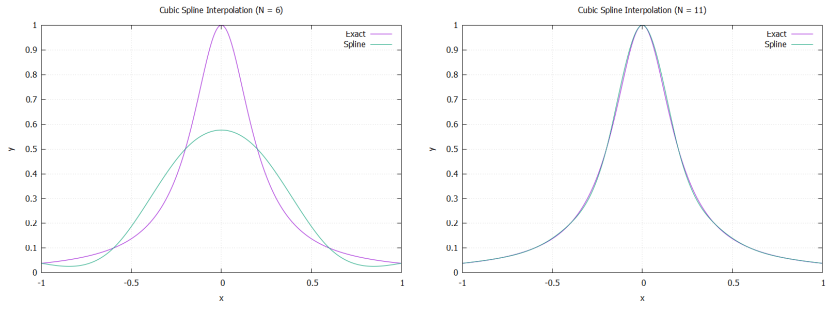

Figure 5: Cubic Spline Interpolation for N = 6 and N = 11

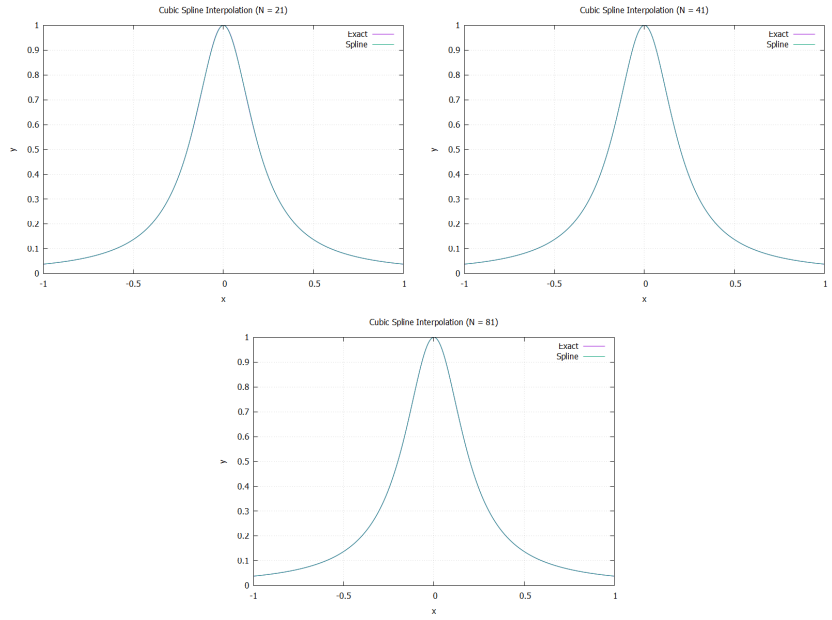

Figure 6: Cubic Spline Interpolation for N = 21, N = 41 and N = 81

The error analysis results for different numbers of nodes are as follows:

Number of Nodes (N)	Maximum Error	Mean Squared Error
6	0.423389	0.0171987
11	0.0219738	5.1134e-005
21	0.00318131	7.71381e-007
41	0.000277477	2.44319e-009
81	1.58168e-005	5.56488e-012

Table 1: Error Analysis for Different Numbers of Nodes

1.2.4 Discussion

The plots illustrate the influence of boundary conditions on the spline interpolation:

- Natural splines have zero second derivatives at the boundaries, resulting in a smooth curve with no overshoot.
- Clamped splines have specified first derivatives at the boundaries, which can lead to overshoots if the derivatives are large.
- Periodic splines are continuous and have the same slope at both ends, which is suitable for cyclic data.

1.2.5 Conclusion

The cubic spline interpolation of the sine function demonstrates how different boundary conditions can affect the interpolation result. Natural splines are smooth without overshoots, clamped splines can have overshoots due to specified derivatives, and periodic splines are ideal for cyclic data. The choice of boundary condition should be based on the specific requirements of the application.

1.3 Test Degree 4 pp-Spline

See `testD4pp.cpp` for the test code.

1.3.1 Methodology

1. Generation of evenly spaced nodes within the interval $[-1, 1]$.
2. Calculation of function values at these nodes using the function $f(x) = \frac{1}{1+25x^2}$.
3. Construction of a Degree-4 spline using these nodes and function values.
4. Evaluation of the spline at a finer set of points to compare with the exact function values.
5. Calculation of interpolation errors.
6. Generation of a plot using Gnuplot to visualize the exact function and the spline interpolation.

1.3.2 Results

The test generated a plot comparing the exact function and the Degree-4 spline interpolation, as shown in Figure 7. The plot illustrates the spline's ability to closely approximate the exact function over the interval.

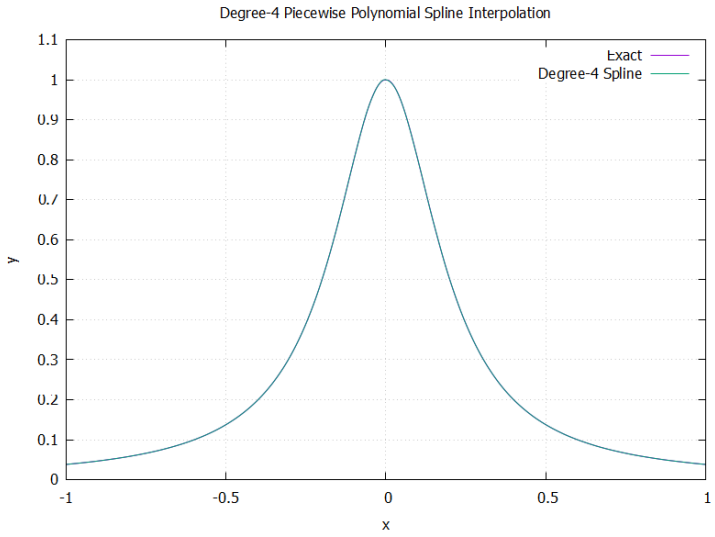

Figure 7: Comparison of the exact function and the Degree-4 spline interpolation.

The test also calculated the maximum error and the mean squared error for the interpolation, providing quantitative measures of the spline's accuracy.

1.3.3 Discussion

The results indicate that the Degree-4 spline provides a good approximation of the exact function, with the plot showing a close match between the spline and the exact function curve.

1.4 Test Degree 5 pp-Spline

See `testD5pp.cpp` for the test code.

1.4.1 Methodology

Most of the steps are the same as "Test Degree 5 pp-Spline", except:

1. Generation of evenly spaced nodes within the interval $[-3, 3]$.
2. Calculation of function values at these nodes using the function $f(x) = \frac{1}{e^{x^2}}$.

1.4.2 Results

The spline interpolation was performed with 11 nodes, and the results are visualized in the provided plot. The plot shows a close match between the exact

function and the interpolated spline, indicating the effectiveness of the interpolation method.

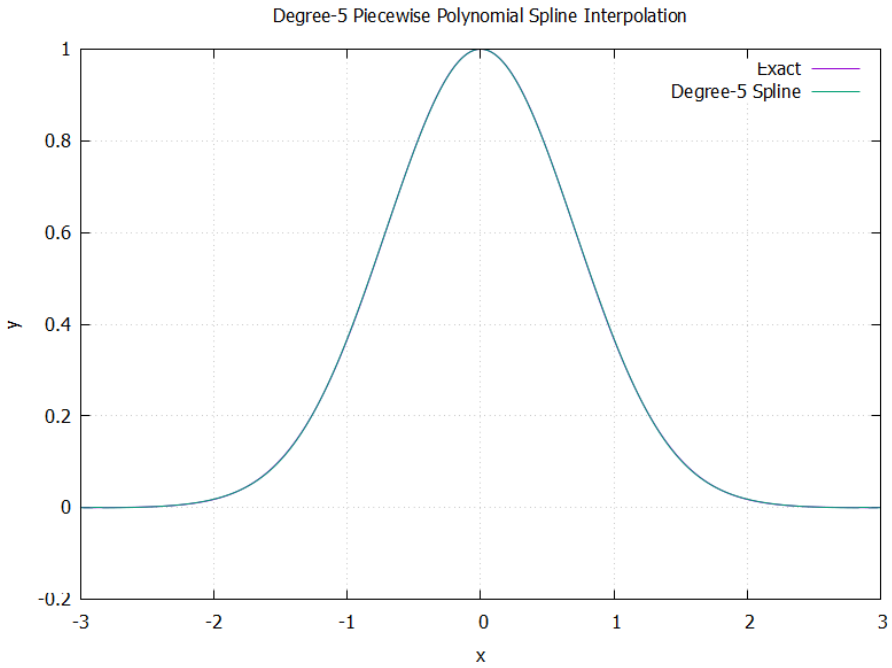

Figure 8: Degree-5 Piecewise Polynomial Spline Interpolation of $f(x) = \frac{1}{e^{x^2}}$

1.4.3 Discussion

The results demonstrate that the degree-5 spline provides a good approximation of the function within the given interval. The plot visually confirms the spline's ability to capture the general shape and behavior of the function, with minor deviations from the exact curve.

1.5 Test Curve-fitting of pp-Form

See `ppheartCurve.cpp` for the test code.

1.5.1 Methodology

The interpolation was performed using the following steps:

1. Define the heart curve parameters and generate t parameter points.
2. Calculate the x and y points for the heart curve.

3. Create cubic splines for x(t) and y(t) with different boundary conditions (natural, clamped, periodic).

4. Generate Gnuplot scripts to plot the heart curve with each boundary condition.

5. Run the Gnuplot scripts to produce the plots.

1.5.2 Results

The results of the cubic spline interpolation for the heart curve with different boundary conditions are shown in Figures 14, 10, and 15. The plots display the heart curve with 10, 40, and 160 points for each boundary condition.

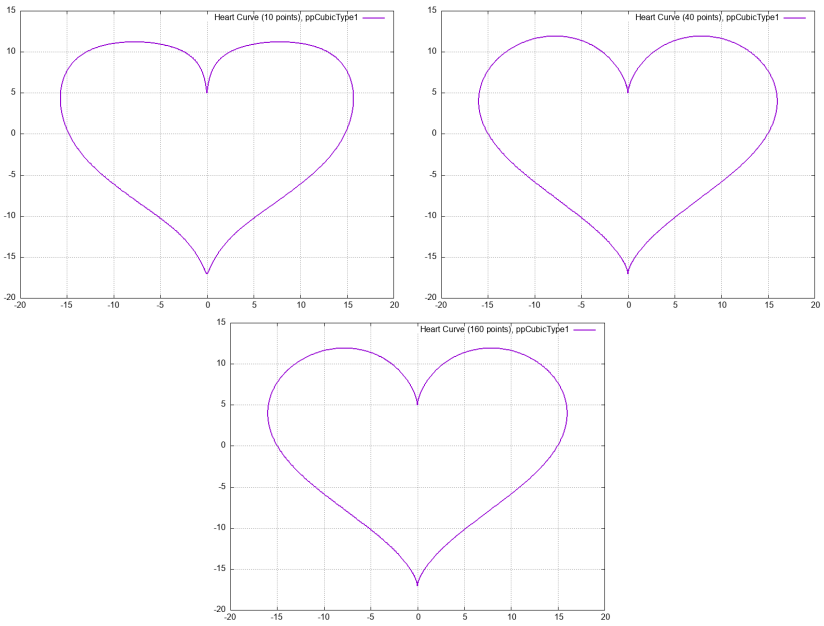

Figure 9: Heart Curve with Natural Boundary: Conditions (1)

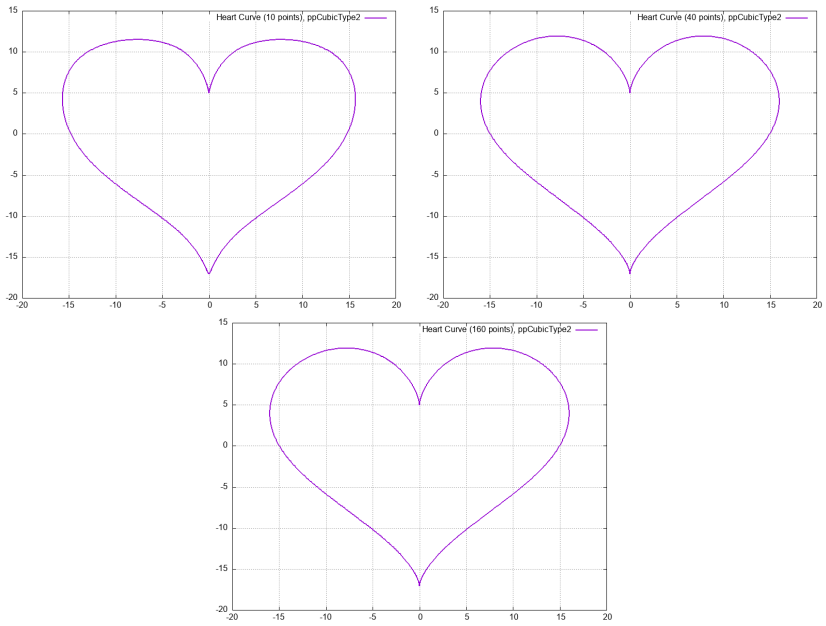

Figure 10: Heart Curve with Clamped Boundary: Conditions (2)

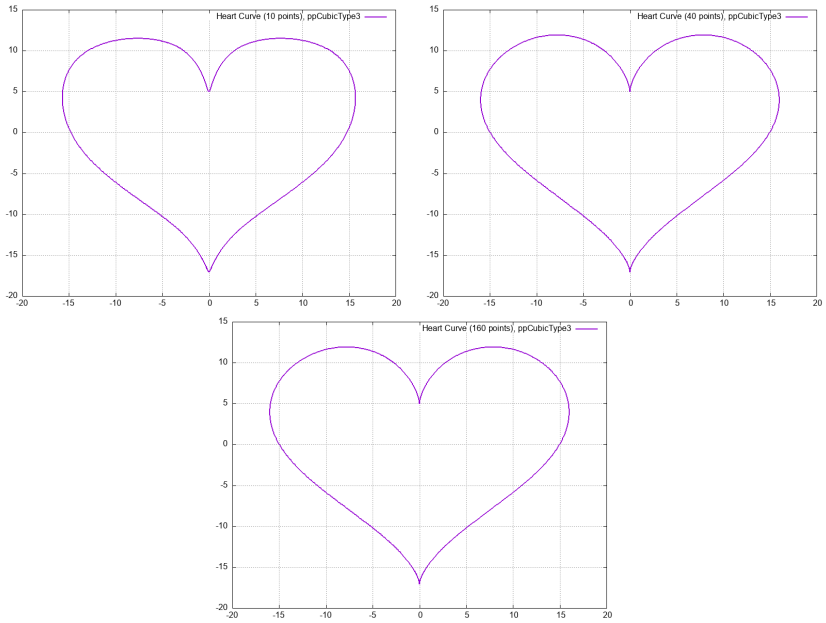

Figure 11: Heart Curve with Periodic Boundary: Conditions (3)

1.5.3 Conclusion

The cubic spline interpolation of the heart curve with different boundary conditions provides a visual demonstration of how boundary conditions influence the interpolation result. For applications requiring different types of boundary behavior, the appropriate cubic spline boundary condition can be selected to achieve the desired curve characteristics.

2 Test of B-Form

2.1 Test Linear B-Spline

See `testLinearB.cpp` for the test code.

2.1.1 Methodology

The interpolation was performed using the following steps:

1. Define the function $f(x) = \frac{1}{1+x^2}$.
2. Select a set of knots within the interval $[-5, 5]$.
3. Compute the function values at these knots.
4. Create a LinearBSpline object with the knots and values.

2.1.2 Results

The result of the linear B-spline interpolation is shown in Figure 18. The plot displays the original function and the interpolated B-spline.

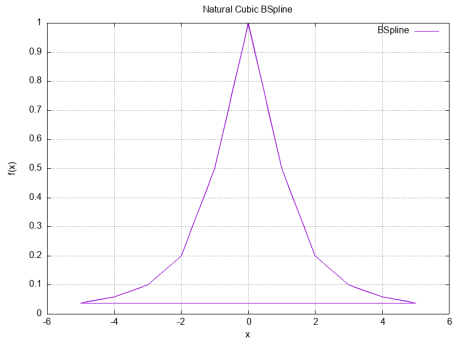

Figure 12: Linear B-Spline Interpolation

2.1.3 Discussion

The plot in Figure 18 illustrates the linear B-spline interpolation of the function. The B-spline closely follows the shape of the original function, demonstrating the effectiveness of this interpolation method. Linear B-splines provide a straightforward and effective way to approximate raw data, suitable for occasions where high-order polynomial interpolation is not required.

2.2 Test Quadratic B-Spline (Problem B&C)

See `testQuadraticB.cpp` for the test code.

2.2.1 Methodology

The interpolation was performed using the following steps:

1. Define the function $f(x) = \frac{1}{1+x^2}$.
2. Select a set of points within the interval $[-5, 5]$.
3. Compute the function values at these points.
4. Create a QuadraticBSpline object with the points and values.

2.2.2 Results

The result of the quadratic B-spline interpolation is shown in Figure 13. The plot displays the original function and the interpolated B-spline.

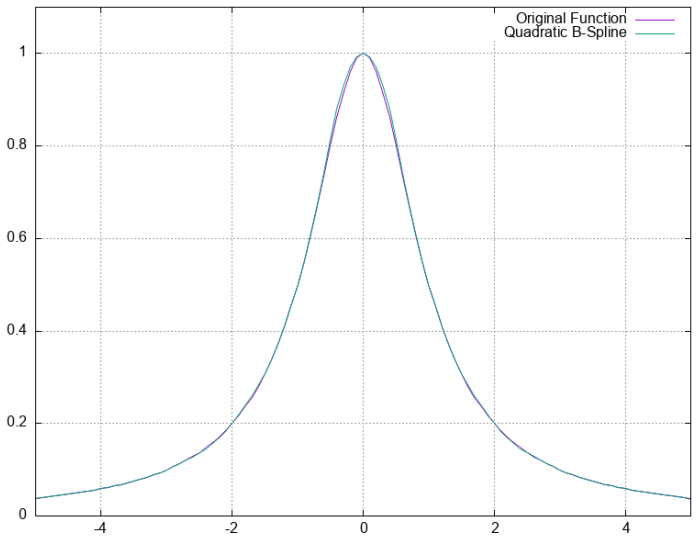

Figure 13: Quadratic B-Spline Interpolation

2.2.3 Discussion

The plot in Figure 13 illustrates the quadratic B-spline interpolation of the function. The B-spline closely follows the shape of the original function, demonstrating the effectiveness of this interpolation method. Quadratic B-splines provide a smooth transition between data points and are particularly useful for their simplicity and efficiency.

2.3 Test Cubic B-Spline (Problem B&C)

See `testCubicB.cpp` for the test code.

2.3.1 Methodology

The interpolation was performed using the following steps:

1. Define the functions $f_1(x) = \frac{1}{1+x^2}$ (for Natural and Clamped) and $f_2(x) = \sin(\pi x)$ (for Periodic).
2. Select a set of nodes for each function.
3. Compute the function values at these nodes.
4. Create cubic B-spline objects with natural, clamped, and periodic boundary conditions.

2.3.2 Results

The results of the cubic B-spline interpolation under different boundary conditions are shown in Figures 14 and 15. The plots display the original functions and the interpolated B-splines.

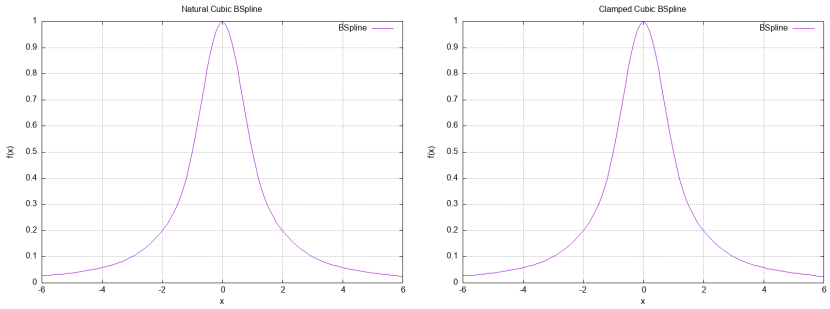

Figure 14: Natural and Clamped Cubic B-Spline Interpolation for $f_1(x)$

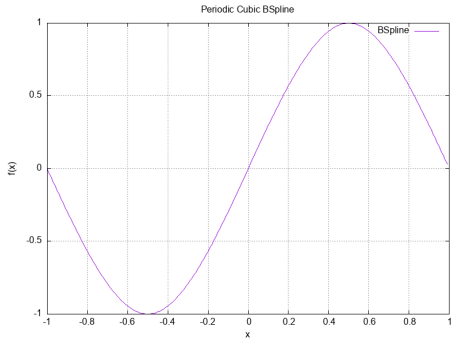

Figure 15: Periodic Cubic B-Spline Interpolation for $f_2(x)$

2.3.3 Discussion

The plots illustrate the influence of boundary conditions on cubic B-spline interpolation:

- Natural splines have zero second derivatives at the boundaries, resulting in a smooth curve with no overshoot.
- Clamped splines have specified first derivatives at the boundaries, which can lead to overshoots if the derivatives are large.
- Periodic splines are continuous and have the same slope at both ends, which is suitable for cyclic data.

2.4 Problem D

See `ProblemD.cpp` for the test code.

2.4.1 Methodology

The interpolation was performed using the following steps:

1. Define the function $f(x) = \frac{1}{1+x^2}$.
2. Select a set of nodes within the interval $[-5, 5]$.
3. Compute the function values at these nodes.
4. Create quadratic and natural cubic B-spline objects.

2.4.2 Results

The results of the quadratic and natural cubic B-spline interpolation are shown in Figures 18 and 17. The plots display the original function, the quadratic B-spline, and the natural cubic B-spline.

Because we made the "midpoint" of the nodes in the implementation process so that the number of nodes and the number of function values are the same in the final operation to simplify the calculation, the error here may be different from the requirements in the book.

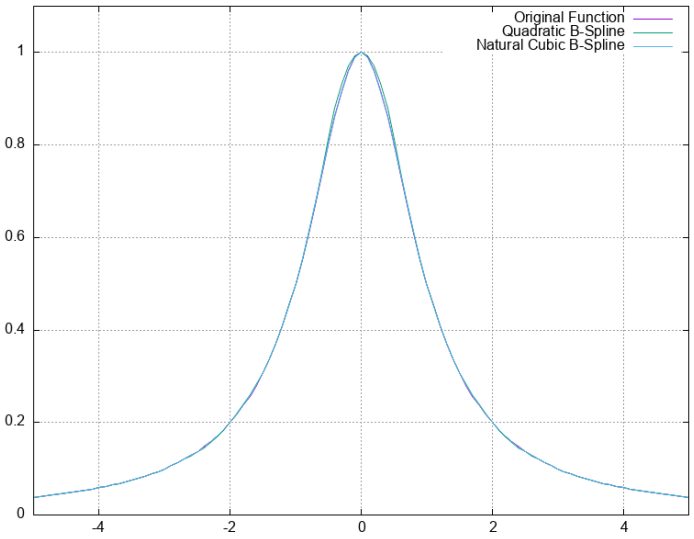

Figure 16: Quadratic and Natural Cubic B-Spline Interpolation

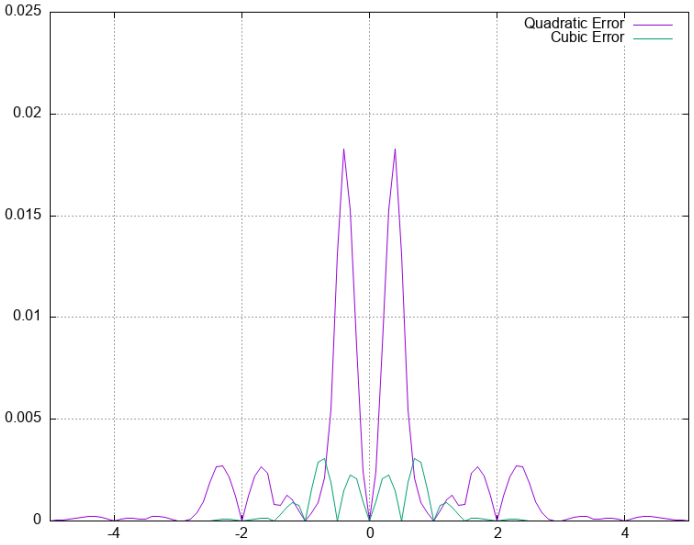

Figure 17: Interpolation Error for Quadratic and Cubic B-Splines

Table 2: Error Analysis for Different Numbers of Nodes

Site	Quadratic Error	Cubic Error
-3.5	7.10258×10^{-5}	2.87991×10^{-18}
-3	3.33053×10^{-17}	8.32803×10^{-18}
-0.5	0.0130736	2.88669×10^{-16}
0	1.11022×10^{-16}	1.11022×10^{-16}
0.5	0.0130736	4.43981×10^{-17}
3	8.32803×10^{-18}	5.54976×10^{-18}
3.5	7.10258×10^{-5}	1.09979×10^{-17}

2.4.3 Discussion

The plots in Figure 17 illustrate the interpolation error for both the quadratic and natural cubic B-splines. The errors close to machine precision are likely due to the high accuracy of the B-spline interpolation at the nodes and the smoothness of the B-spline curves. The cubic B-spline appears to be more accurate overall, as indicated by the smaller error values across the range.

2.5 Any Order and Any Node Splines

See `calculateBSpline.cpp` for the test code.

The B-Spline format should support the drawing of any order and any node splines. That is, to implement the formula:

$$S(t) = \sum_{i=2-n}^{N} a_i B_i^n(t) \in \mathbb{S}_n^{n-1}(t_1, \ldots, t_N), \quad t \in [t_1, t_N]$$

2.5.1 Methodology

The interpolation was performed using the following steps:

1. The degree of the B-spline n was set to 4.
2. The number of knots M was determined to be 20.
3. Knot values were input in non-decreasing order: 1, 2, 3, 5, 9, 10, 11, 12, 15, 16, 18, 19, 21, 22, 24, 30, 31, 32, 35, 37, (At the beginning and end, you are required to enter additional nodes that do not draw curves).
4. Control point coefficients were input for the B-spline: 8, 5, 9, 14, 5, 6, 3, 2, 1, 4, 16, 9, 21, 11, 10.

2.5.2 Results

The result of the B-spline interpolation is shown in Figure 18. The plot displays the B-spline curve generated from the input parameters.

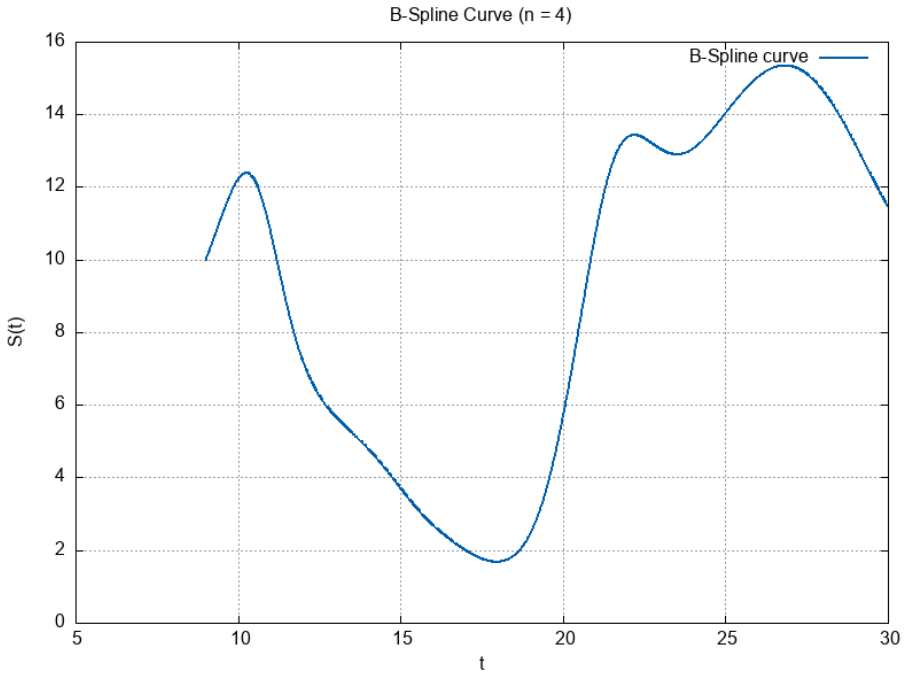

Figure 18: B-Spline Curve (n = 4)

2.5.3 Conclusion

The implementation of the B-spline formula allows us to plot the B-spline curve based on the given node and control point coefficients. This method is useful in computer graphics and numerical analysis because it provides a flexible way to approximate complex curves.

2.6 Test Curve-fitting of B-Form (Problem E)

See `BheartCurve.cpp`, `BheartCurveCCL.cpp`, `Br2Curve.cpp`, `Br2CurveCCL.cpp`, `Br3Curve.cpp`, `Br3CurveCCL.cpp` for the test code.

Here, we implement curve-fitting on:

- heart-shaped curve,
- $r_2(t) = (x(t), y(t)) = (\sin t + t\cos t, \cos t - t\sin t)$, for $t \in [0, 6\pi]$
- $r_3(t) = (x(t), y(t), z(t)) = (\sin(u(t))\cos(v(t)), \sin(u(t))\sin(v(t)), \cos(u(t)))$, for $t \in [0, 2\pi]$, where $u(t) = \cos t$, $v(t) = \sin t$.

The approximation of these three curves should use cumulative chordal length and equidistant nodes respectively, and employ three boundary conditions. Finally, we compare the approximation effects of these two types of nodes.

Below we have taken the heart-shaped curve as an example for specific testing and analysis (the other two curves are similar, and we only give two parameterized fitting results).

The derivatives at the two cusps are undefined, therefore When selecting an interpolation node, choose one of them as the start and end points.

2.6.1 Equidistant Nodes

1. Define the heart curve function

$$x(t) = 16 \cdot \sin^3(t),$$
$$y(t) = 13 \cdot \cos(t) - 5 \cdot \cos(2t) - 2 \cdot \cos(3t) - \cos(4t), \quad t \in [0, 2\pi]$$

2. Select different numbers of nodes and set node values for each condition(10, 40, 160).

3. Create B-spline objects for each boundary condition.

The results of the B-spline interpolation under different boundary conditions (1-Natural, 2-Clamped, 3- Periodic) are shown in the included figures. The plots display the original heart curve and the interpolated B-spline.

We show some of them (Natural of node values 10, 40, 160; Clamped of node values 40; Periodic of node values 40; Natural Error of node values 10, 40, 160; Clamped Error of node values 40; Periodic Error of node values 40) :

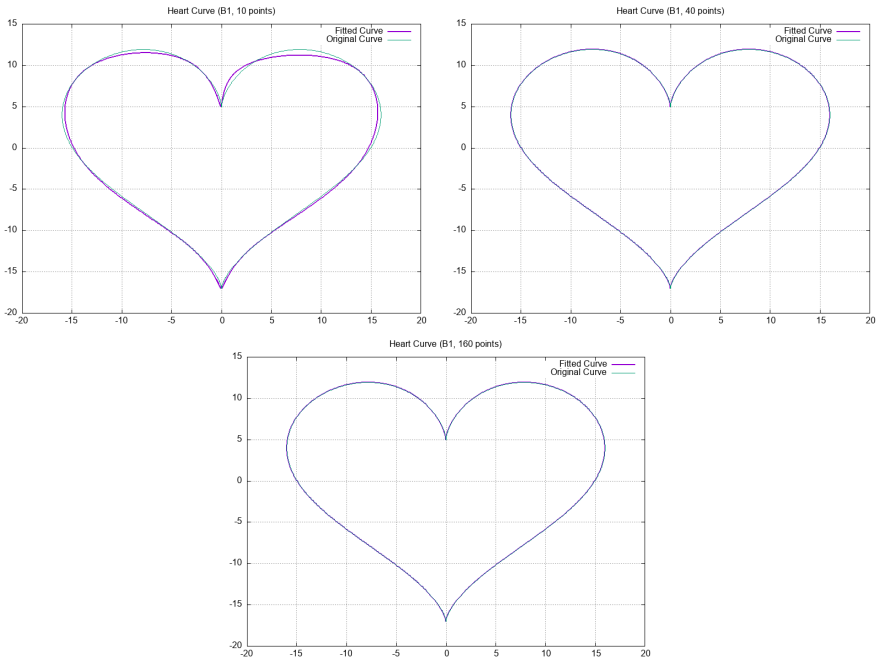

Figure 19: Natural of Heart Curve, Node values 10, 40, 160

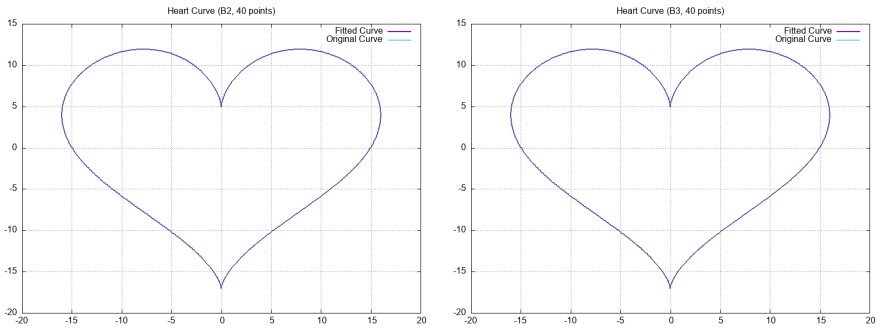

Figure 20: Clamped and Periodic of Heart Curve, Node values 40

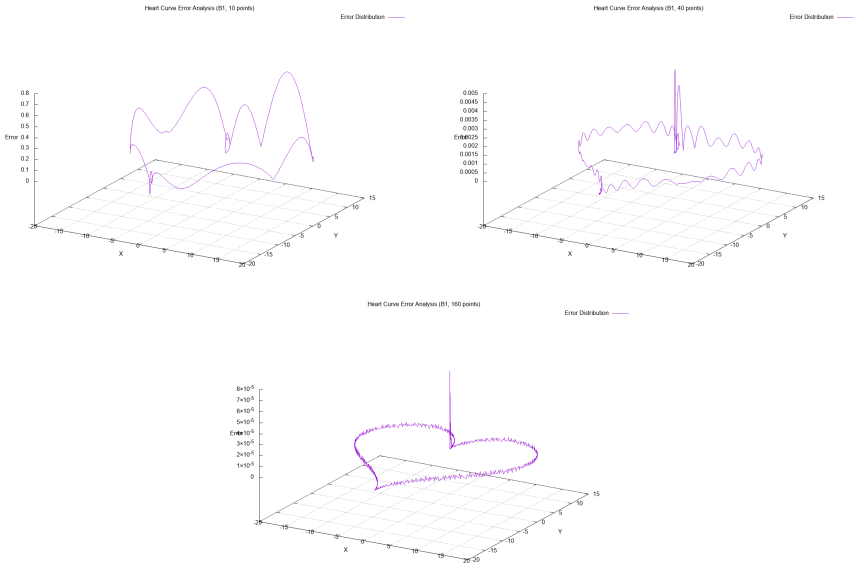

Figure 21: Natural Error of Heart Curve, Node values 10, 40, 160

1. The error plots demonstrate the impact of the number of points on the accuracy of the B-spline interpolation. As the number of points increases, the error decreases significantly, indicating a more accurate interpolation.

2. The relatively large error at the endpoint is due to the fact that the B-spline base does not completely cover the region.

For curve r_2 (do not have periodic boundary conditions) and r_3, we have:

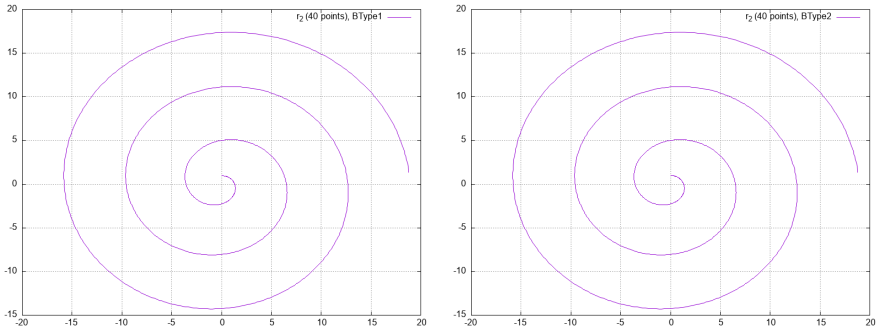

Figure 22: Natural and Clamped of r_2, Node values 40

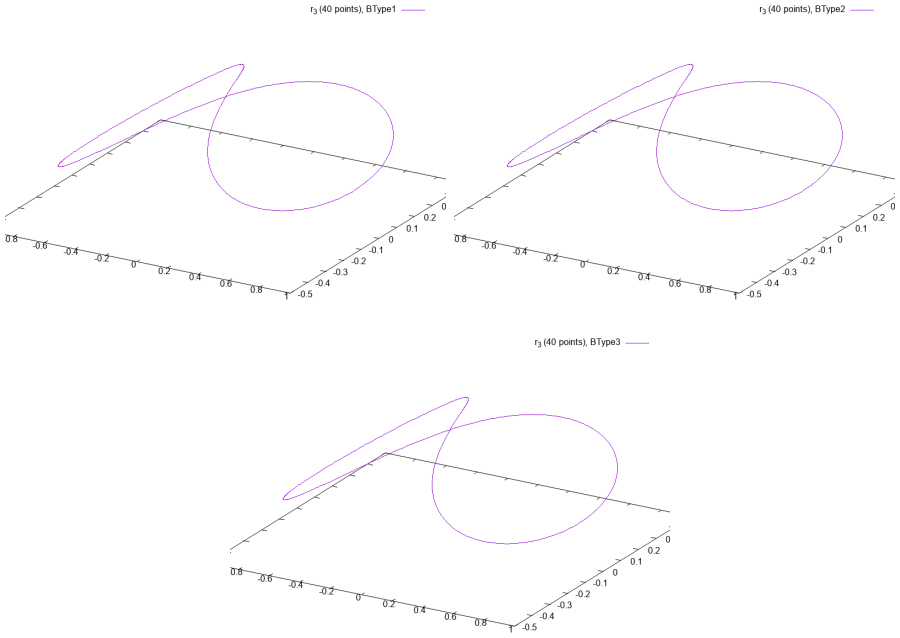

Figure 23: Natural, Clamped and Periodic of r_3, Node values 40

2.6.2 Cumulative Chordal Length Method

The *cumulative chordal lengths* of a point sequence $(x_i)_{i=1}^{N}$ in $\mathbb{R}^D$ form a sequence of N real numbers,

$$t_i = \begin{cases} 0, & \text{if } i = 1; \\ t_{i-1} + \|\mathbf{x}_i - \mathbf{x}_{i-1}\|_2, & \text{otherwise,} \end{cases}$$

where $\|\cdot\|_2$ denotes the Euclidean 2-norm.

So we get the new parameter and use the new parameter for B-spline curve fitting. Here, we show the curve fitting results of the three curves above (Heart Curve, r_2 and r_3 with Natural boundary conditions of node values 40; Error of Heart Curve with Natural boundary conditions of node values 40;) :

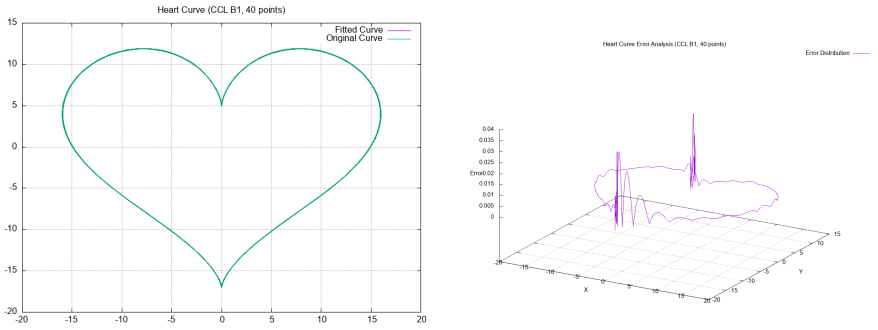

Figure 24: Natural of Heart Curve, Node values 40

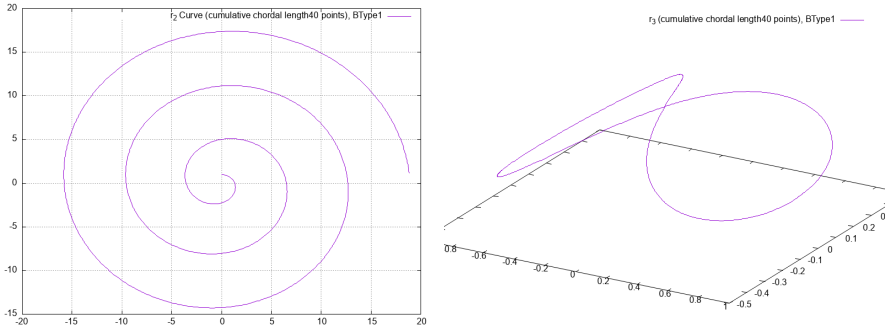

Figure 25: Natural of r_2 and r_3, Node values 40

2.6.3 Compare the Two Parameterizations

Output by the program, we have:

Spline Type	Maximum Error	Average Error
Natural(Uniform)	0.00477674	0.000369366
Clamped(Uniform)	0.000714064	0.000216572
Periodic(Uniform)	0.000714051	0.000216687
Natural(CCL)	0.0354496	0.0032255
Clamped(CCL)	0.0354496	0.00379937
Periodic(CCL)	0.0354496	0.00397928

Table 3: Error Analysis for Different Numbers of Nodes

By comparing the maximum and average errors of the heart curve at N=40 under the three boundary conditions, we can observe the CCL parameterization results in higher errors compared to the Uniform parameterization.

When the data points change sharply (two cusps of the heart curve), the curves parameterized by the cumulative chord length may perform poorly in smoothness. This is because when dealing with fast-changing data points, cumulative chord length parameterization may not transition smoothly when nodes happen to span fast-changing regions, resulting in unnatural curves or fluctuations in the curves of those regions.

2.7 Curve-fitting on the Unit Ball

See `CurveonBall.cpp` for the test code.

Here, we use the curve r_3 above as the test case.

2.7.1 Methodology

The implementation involves the following steps:

1. Generation of parameter points on the sphere.
2. Application of the Riemann mapping to transform the spherical curve into a planar curve.
3. Construction of cubic B-splines for the transformed curve.
4. Inversion of the Riemann mapping to map the planar spline back onto the sphere.

2.7.2 Riemann Mapping and Its Inverse

The Riemann ball realizes a mapping: $\partial B \backslash N \to \mathbb{R}^2$, $N = P(\infty)$.

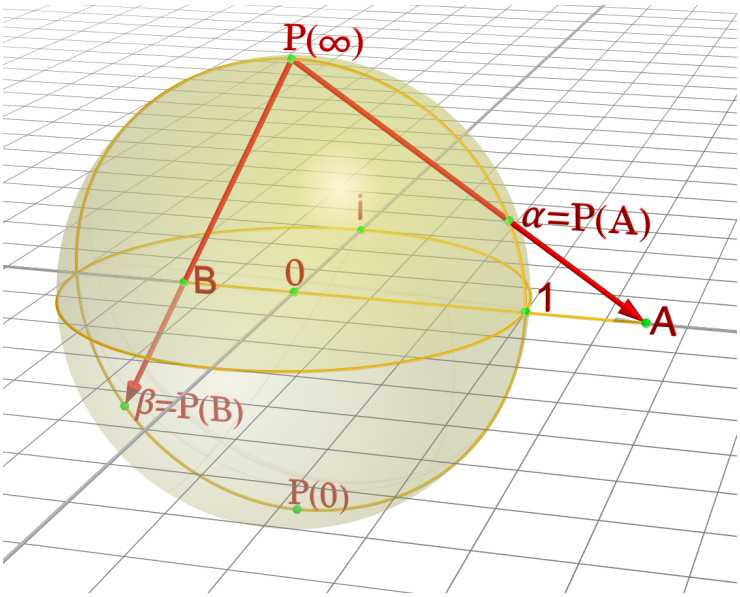

Figure 26: Riemann Mapping.

The Riemann mapping is defined as:

$$x_{\text{Plane}} = \frac{0.5x_{\text{Ball}}}{1 - 0.5(z_{\text{Ball}} + 1)}, \quad y_{\text{Plane}} = \frac{0.5y_{\text{Ball}}}{1 - 0.5(z_{\text{Ball}} + 1)},$$

where t is the parameter along the curve.

The inverse Riemann mapping is given by:

$$\begin{aligned} x_{\text{Ball}} &= \frac{2x_{\text{Plane}}}{x_{\text{Plane}}^2 + y_{\text{Plane}}^2 + 1}, \\ y_{\text{Ball}} &= \frac{2y_{\text{Plane}}}{x_{\text{Plane}}^2 + y_{\text{Plane}}^2 + 1}, \\ z_{\text{Ball}} &= \frac{2(x_{\text{Plane}}^2 + y_{\text{Plane}}^2)}{x_{\text{Plane}}^2 + y_{\text{Plane}}^2 + 1} - 1 \end{aligned}$$

2.7.3 Results

Since the N point is not defined, we do mirror symmetry with respect to the XY plane for r_3.

The test results are visualized in the provided images, showing the spline fitting on the Riemann sphere for different numbers of points (10, 40, 160). The spline closely follows the original curve, demonstrating the effectiveness of the method.

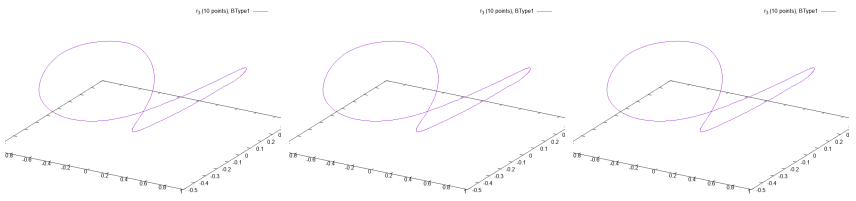

Figure 27: Spline fitting on ball with 10, 40 and 160 points.

2.7.4 Discussion

The results indicate that the spline fitting on the Riemann sphere is robust and accurate, especially as the number of points increases. The use of the Riemann mapping allows for straightforward spline construction in a planar space, which is computationally more manageable.

3 Compare pp-Form with B-Form

See `compareppwithBcurve.cpp` for the test code.

Here, we interpolate the heart-shaped curve with three boundary conditions, pp-Form and B-Form, respectively, to compare the errors of the two fitted curves (number of nodes $N = 40$).

3.1 Methodology

The test was implemented in the following steps:

1. Generation of parameter points for the heart curve.
2. Calculation of x and y points for the heart curve.
3. Creation of Cubic pp-Splines and Cubic B-Splines for x(t) and y(t) with the same set of control points and boundary conditions.
4. Calculation of the error between the pp-Spline and B-Spline curves at a higher resolution.
5. Generation of 3D error distribution plots using Gnuplot.

3.2 Results

The test was conducted for three different spline types (1-Natural, 2-Clamped, 3- Periodic) with 40 control points each. The following 3D error distribution plots were generated:

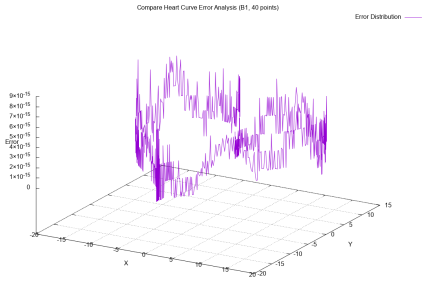

Figure 28: Error distribution for Natural boundary conditions.

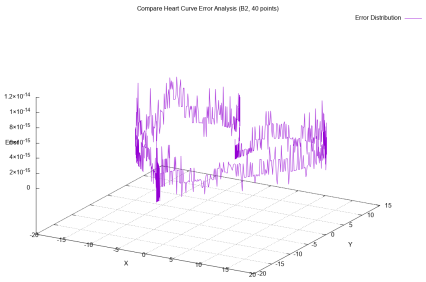

Figure 29: Error distribution for Clamped boundary conditions.

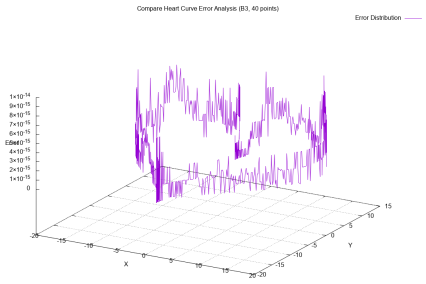

Figure 30: Error distribution for Periodic boundary conditions.

3.3 Discussion

The error distribution plots indicate that the pp-Form and B-Spline methods produce very similar results, with the error being on the order of 10^{-15} to 10^{-14}. This level of error is close to machine error, suggesting that both methods are equivalent in this context.

4 More Function Sample Tests

4.1 $sin\pi x, \quad x \in [-1, 1]$

See the results in part **Test Linear pp-Form** and **Test Cubic pp-Form**.

4.2 $f(x) = \frac{1}{e^{x^2}}, \quad x \in [-3, 3]$

See the result in part **Test Degree 5 pp-Spline**.

4.3 Other Tests of Curve-fitting on the Unit Ball

4.3.1 r_4

The parametric equation for r_4 is:

$$\begin{aligned} &x = sin(cos(2t))cos(t), \\ &y = sin(cos(2t))sin(t), \\ &z = -cos(cos(2t)), \\ &where \quad t \in [0, 2\pi]. \end{aligned}$$

The results are shown below:

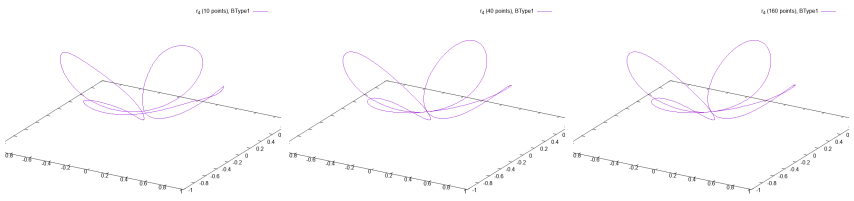

Figure 31: Spline fitting of r_4 on ball with 10, 40 and 160 points.

4.3.2 r_5

The parametric equation for r_5 is:

$$\begin{aligned} &x = sin(cos(t) + 0.5)cos(sin(2t)), \\ &y = sin(cos(t) + 0.5)sin(sin(2t)), \\ &z = -cos(cos(t) + 0.5), \\ &where \quad t \in [0, 2\pi]. \end{aligned}$$

The results are shown below:

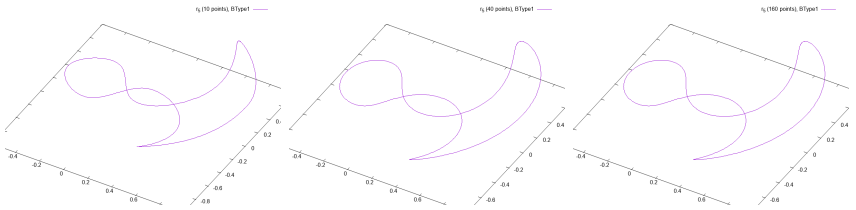

Figure 32: Spline fitting of r_5 on ball with 10, 40 and 160 points.

5 Problem F

See `ProblemF2.cpp, ProblemF2.cpp` for the test code.

5.1 Results

The test generated the following plots for $n = 1$ and $n = 2$:

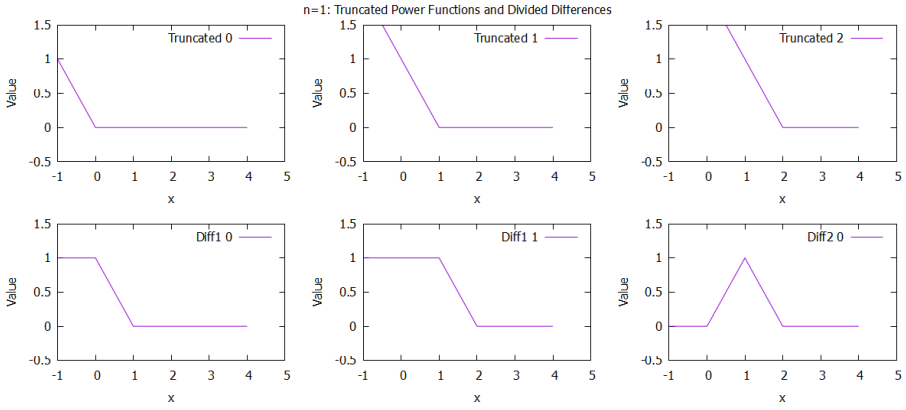

Figure 33: Plots for $n = 1$: Truncated power functions and first and second divided differences.

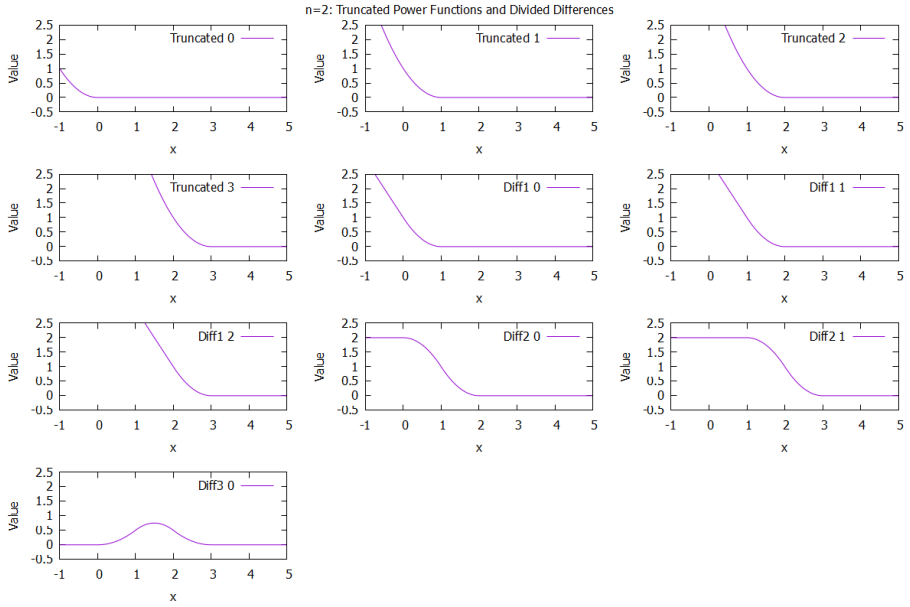

Figure 34: Plots for $n = 2$: Truncated power functions and first, second, and third divided differences.

As shown in the figure, we verify that $n = 1$ and $n = 2$ satisfy:
For any $n \in \mathbb{N}$,

$$B_i^n(x) = (t_{i+n} - t_{i-1}) \cdot [t_{i-1}, \ldots, t_{i+n}](t - x)_+^n.$$

6 Self-Intersection Detection

See `intersection.cpp` for the test code.

The program employs a parametric curve definition and a distance calculation algorithm to identify potential intersections.

6.1 Methodology

The program operates through the following steps:

1. Generation of curve points using a parametric equation.
2. Calculation of the distance between two line segments in 3D space.
3. Detection of self-intersection by comparing segment distances against a threshold.

6.1.1 Curve Definition

Here we use the r_3 mentioned above as the test curve. The curve is parametrically defined as:

$$x = \sin(\cos(t))\cos(\sin(t)),$$
$$y = \sin(\cos(t))\sin(\sin(t)),$$
$$z = \cos(\cos(t)),$$

where t ranges from 0 to 2π.

6.1.2 Distance Calculation

The distance between two line segments $p1p2$ and $p3p4$ is calculated using the algorithm outlined below:

1. Compute vectors $\mathbf{u} = p2 - p1$, $\mathbf{v} = p4 - p3$, and $\mathbf{w} = p3 - p1$.
2. Calculate dot products $a = \mathbf{u} \cdot \mathbf{u}$, $b = \mathbf{u} \cdot \mathbf{v}$, and $c = \mathbf{v} \cdot \mathbf{v}$.
3. Compute the cross product $\mathbf{dP} = \mathbf{w} - (\mathbf{u} \cdot \mathbf{w}/a)\mathbf{u} - (\mathbf{v} \cdot \mathbf{w}/c)\mathbf{v}$.
4. The distance is then $\sqrt{\mathbf{dP} \cdot \mathbf{dP}}$.

6.1.3 Self-Intersection Detection

The program checks for self-intersection by iterating through pairs of curve segments. For each pair, it:

1. Computes the distance between the segments using the line segment distance function.
2. Compares this distance to a predefined threshold. If the distance is less than the threshold, the curve is considered to self-intersect at that point.

6.2 Results

The program execution resulted in the detection of a self-intersection within the curve. With the default parameters set (1000 samples, a threshold of 1×10^{-5}, and a minimum segment distance of 10), the program successfully identified that at least one pair of line segments had a distance less than the predefined threshold, indicating that the curve intersects itself at some point.

6.3 Discussion

The algorithm effectively identifies potential self-intersections by calculating the minimum distance between all possible pairs of line segments. The use of a threshold allows for flexibility in determining what constitutes an intersection based on the application's precision requirements.