Team#5 Writeup

Introduction

The main executable is called main.cpp with most of the other programs for executing curve functions in the source.h header file. The only code we could not add to the main is Revolved_surface program which has to be run separately.

The provided C++ code serves as a menu-driven interface for accessing various geometric modeling functionalities. Upon execution, the main program prompts the user to select an operation from a list, including clamped and unclamped spline interpolation, knot insertion and refinement, generation of a general cylinder, creation of ruled surfaces, Bezier curve generation, Rational Bezier curve generation, and Cubic Spline interpolation. Each operation corresponds to a separate function defined in external source files.

For instance, selecting "clamped_unclamped" invokes a function that allows the user to input control points and knot vectors, then generates an interpolated spline curve based on those inputs. Similarly, choosing "knot_insert_refine" prompts the user to define a degree, control points, and original knot vector, followed by the option to insert or refine knots within the spline curve.

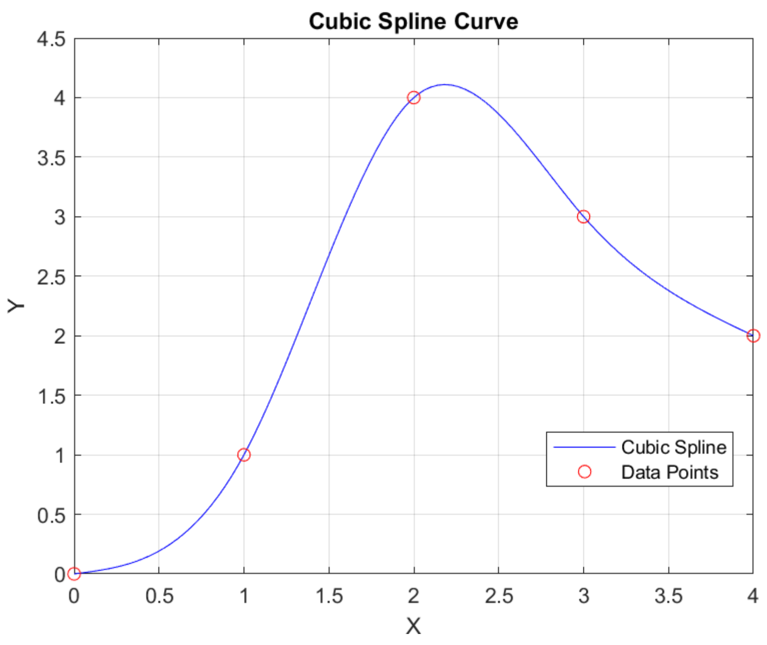
Other operations follow a similar pattern, each encapsulated within its own function and sourced from dedicated external files. These functions leverage mathematical algorithms such as NURBS interpolation, Bezier curve generation, and Cubic Spline interpolation to achieve their respective geometric modelling tasks.

Curves

We have created the individual program for each of the required curves, each section will describe the code features used to generate the plots. We generated the required points for the curves using C++ program that allows the user to supply a set of data points (and potentially slopes at each data point) and then we plot the curves using Matlab.

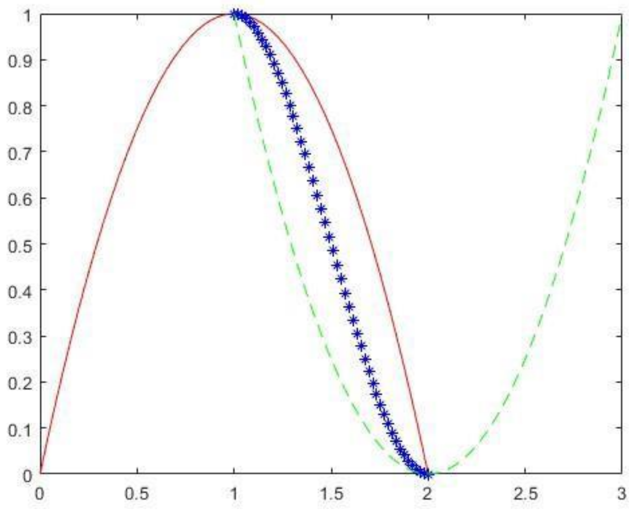
C- spline:

The C++ code that is provided carries out cubic spline interpolation. It defines a structure called ‘Point’ that uses ‘x’ and ‘y’ coordinates to represent data points. The logic for cubic spline interpolation is contained in the ‘CubicSplineInterpolation’ class, which uses the supplied data points to compute spline coefficients at startup. Based on the obtained coefficients, the ‘interpolate()’ method determines the interpolated ‘y’ value for a given ‘x’ value. Users enter data points interactively in the ‘main()’ function, and those points are utilized to generate an instance of the ‘CubicSplineInterpolation’ class. Interpolation is carried out for a range of ‘x’ values by the program, which outputs the interpolated points as ‘x, y’ pairs to the console. With the help of this code, users may easily conduct cubic spline interpolation on their data sets.



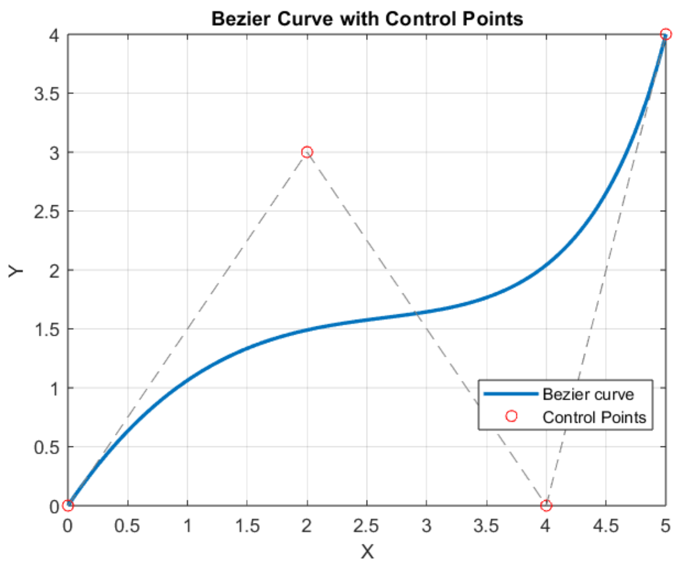
Parabolic blending

This program utilizes the principles of Overhauser curve blending interpolation and Lagrange interpolation. It prompts the user for four data points, generating parabolas based on the first three points and the last three points using Lagrange interpolation. Subsequently, it employs Overhauser parabolic blending to merge the two inner points. The C++ code is only able to do the Lagrange interpolation but we couldn’t get the Overhauser to work in C++. Everything works perfectly in the matlab version of the code.

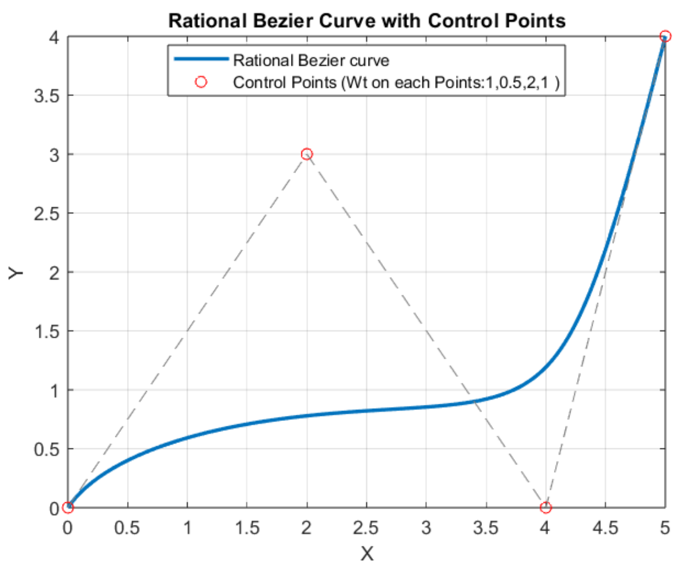


Bezier and Rational Bezier Curves

The code implements algorithms to generate Bézier and Rational Bézier curves in C++. It prompts users to input parameters such as the degree of the curve, control points, and optionally weights for Rational Bézier curves. Using these inputs, the program computes the coordinates of points along the curves and stores the results in text files. Bézier curves are calculated using Bernstein polynomials and binomial coefficients, while Rational Bézier curves incorporate weighted control points to offer greater flexibility in shaping the curves. We plotted the curve exporting the text file to MATLAB.

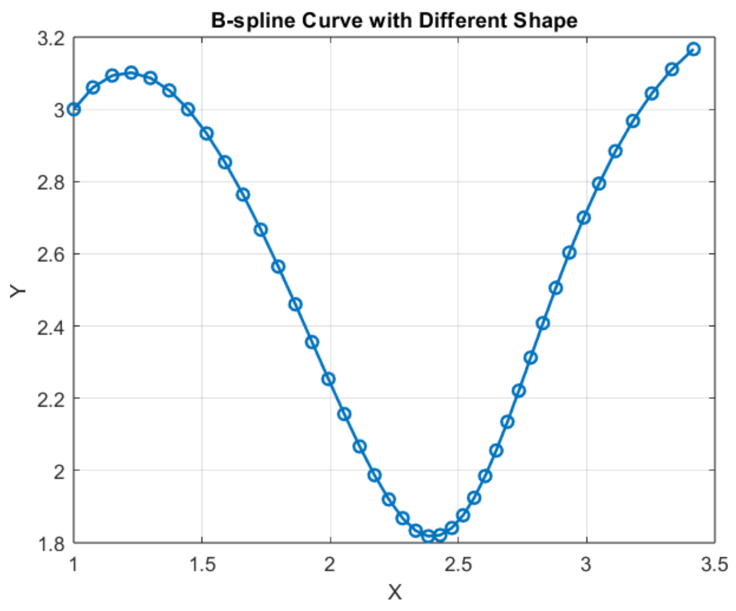


Bezier Curve with Control Points
Y
X
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Bezier curve
Control Points



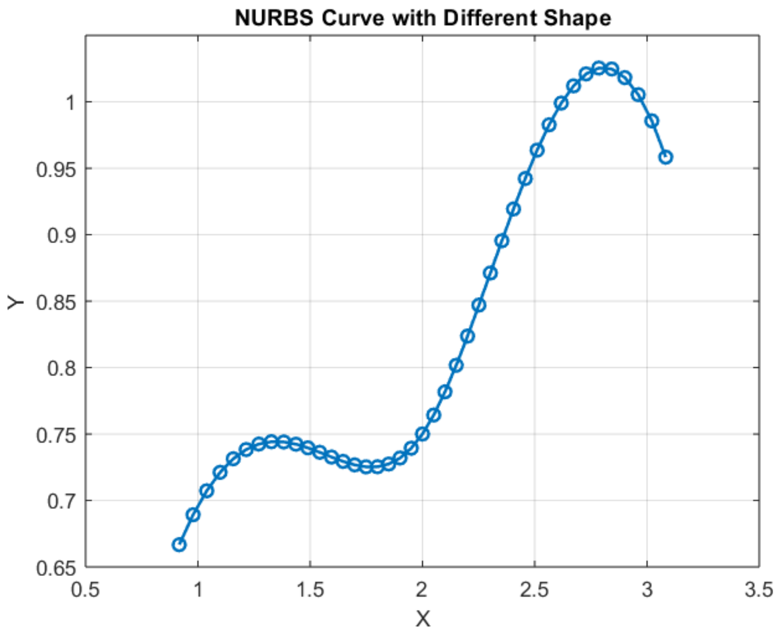
B-spline Curve:

To develop the B-spline curve program, we first delved into the theory behind B-spline curves. B-splines are piecewise polynomial curves defined by control points and a knot vector. We researched the Cox-de Boor recursion formula, which recursively computes B-spline basis functions used to interpolate the curve. With this understanding, we crafted an algorithm to calculate B-spline curves in C++. Our algorithm utilized vectors to store control points and knot vectors, with functions to compute B-spline basis functions and interpolate the curve. The control points and knot vector were provided as input parameters, along with the degree of the curve and the step size for sampling points. We rigorously tested the code to ensure it produced accurate B-spline curves for various shapes. After execution, the program generated a text file containing the points on the B-spline curve. These points were then plotted using MATLAB to visualize the curve's shape and behavior.



NURBS :

For the NURBS curve program, we expanded our knowledge from B-splines to Non-uniform Rational B-splines (NURBS). NURBS curves are an extension of B-splines that incorporate weights for each control point, providing greater flexibility in shape representation. We extended our algorithm to handle NURBS curves by integrating weights into the interpolation process. The Cox-de Boor recursion formula remained the foundation for computing basis functions, now accounting for control point weights. Similar to the B-spline program, the NURBS program accepted control points, knot vectors, degree, and step size as input parameters. The C++ code was designed to compute NURBS curves using the extended algorithm and output the points on the curve to a text file. Upon execution, the program generated a separate text file containing the NURBS curve points. These points were then plotted using MATLAB to visualize the curve's shape and characteristics, allowing for analysis of its behavior and adjustment of input parameters as needed.

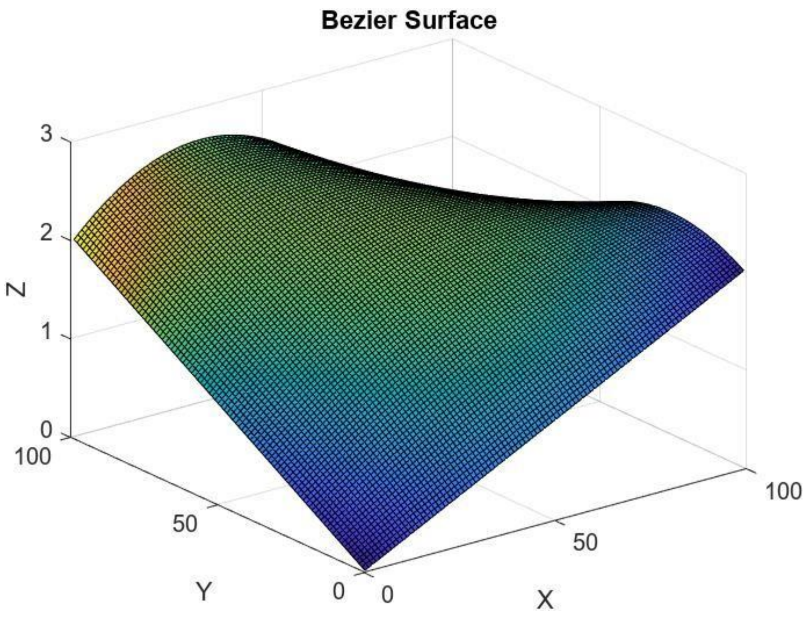


Surfaces

This set of programs demonstrates the generation and visualization of various types of surfaces, including Bezier surfaces, Rational Bezier surfaces, B-spline surfaces, and NURBS surfaces. Developed primarily in C++, these programs prompt users to input parameters such as degrees, control points, knot vectors, and weights to define the surfaces. Subsequently, the programs compute surface points and write them to text files. While the main code is generated in C++, the visualization of the results is facilitated using MATLAB. This combination provides users with a robust toolset for exploring and analyzing these diverse types of surfaces, enabling insights into their properties and behavior.

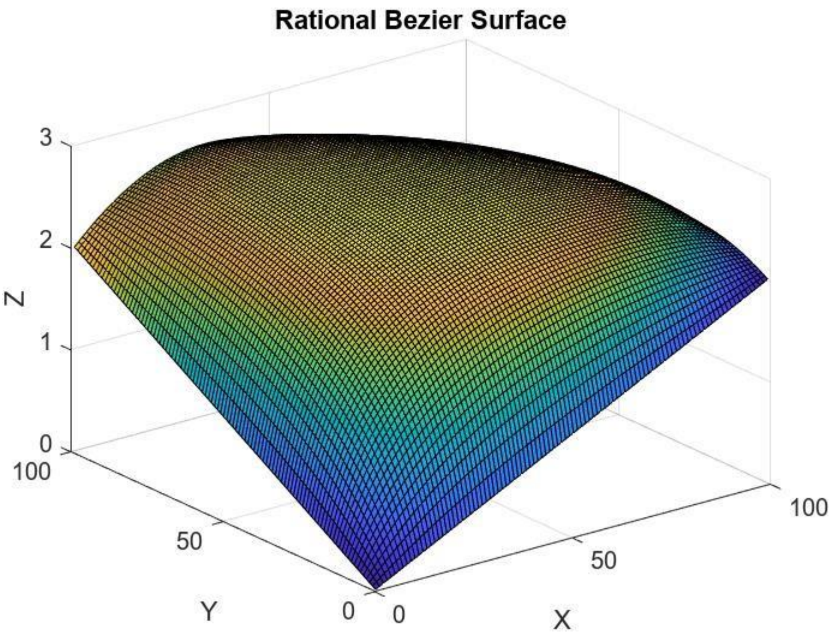
Bezier Surface:

This C++ program allows users to define the degree and control points of a Bezier surface and generates points on it by iterating through **u** and **v** parameters. It calculates the coordinates of each point using the **BezierSurface** function, which utilizes the binomial coefficient calculation from the **binomial** function. The generated points are then written to a text file named "bezier_surface_output.txt". Upon completion, the program provides feedback on the success or failure of writing to the file and waits for user input to keep the terminal open before exiting.



Rational Bezier surface:

This C++ program enables users to define the degree, control points, and weights of a Rational Bezier surface, generating points on it through parameterized iterations of **u** and **v**. The **RationalBezierSurface** function calculates the coordinates of these points by computing the weighted sum of control points' coordinates, normalized by the sum of weights. Upon input completion, the generated points are written to a text file named "rational_bezier_surface_output.txt". The program provides feedback on the success or failure of writing to the file and waits for user input to keep the terminal open before terminating. These output points are suitable for visualizing the Rational Bezier surface in MATLAB, facilitating further analysis and exploration of its properties.



B-spline Surface:

his C++ program facilitates the creation of a B-spline surface based on user-defined control points, knot vectors, and degrees in both U and V directions. It utilizes Cox-de Boor recursion formulas to compute the B-spline basis functions for both U and V directions. The **BSplineSurface** class encapsulates this functionality, offering methods to calculate B-spline surface points and plot the surface by sampling points at regular intervals. In the **main** function, users are prompted to input parameters such as the number of control points, control point coordinates, knot vectors, degrees, and step size for sampling points on the surface. The program then generates the B-spline surface and writes the resulting points to a text file named "bspline_surface.txt". This output can be utilized for further visualization and analysis, providing a flexible tool for exploring B-spline surfaces.

B spline : Example for plotting

Enter the number of rows for the control points matrix: 3

Enter the number of columns for the control points matrix: 3

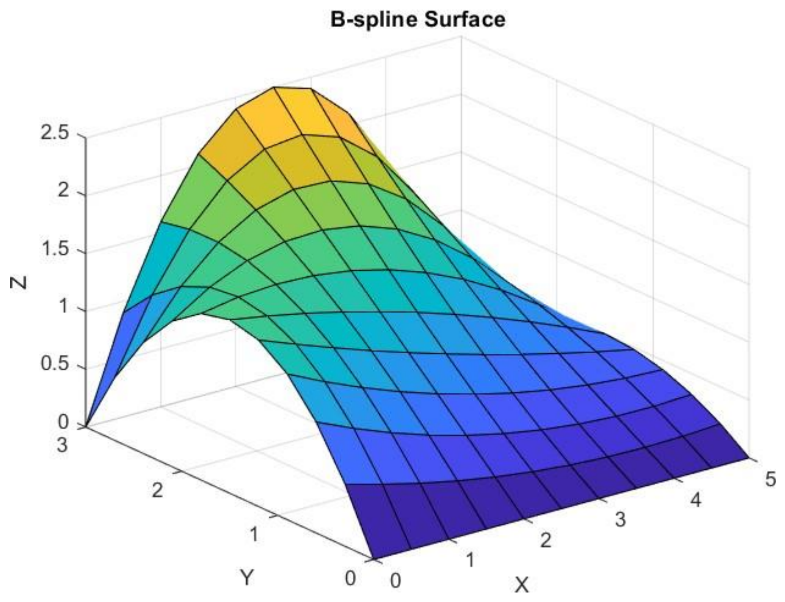
Enter the control points:

Control Point (0, 0): 0 0 0

Control Point (0, 1): 1 0 1

Control Point (0, 2): 2 0 0

```
Control Point (1, 0): 0 1 0
Control Point (1, 1): 1 1 2
Control Point (1, 2): 2 1 0
Control Point (2, 0): 0 2 0
Control Point (2, 1): 1 2 1
Control Point (2, 2): 2 2 0
Enter the knot vector for U direction: 0 0 0 1 1 1
Enter the knot vector for V direction: 0 0 0 1 1 1
Enter the degree of the B-spline surface in U direction: 2
Enter the degree of the B-spline surface in V direction: 2
Enter the step size for sampling points on the surface: 0.1
```



NURBS Surface:

This C++ program allows users to define a NURBS (Non-Uniform Rational B-Spline) surface by specifying control points, knot vectors, degrees in both U and V directions, and weights for each control point. The **NURBSSurface** class encapsulates functionality to compute NURBS basis functions using Cox-de Boor recursion formulas, and to calculate NURBS surface points based on the provided parameters. In the **main** function, users are prompted to input the necessary parameters such as the number of control points, control point coordinates, knot vectors, degrees, and step size for sampling points on the surface. The program then generates the NURBS surface and writes the resulting points to a text file named "nurbs_surface.txt". This output file can be utilized for further visualization and analysis, providing a versatile tool for exploring NURBS surfaces.

Nurbs – Example

```
Enter the number of control points in U direction: 3
Enter the number of control points in V direction: 4
Enter control points:
Control Point (0, 0): 0 0 0
Control Point (0, 1): 1 0 1
Control Point (0, 2): 2 0 0
Control Point (0, 3): 3 0 1
Control Point (1, 0): 0 1 0
Control Point (1, 1): 1 1 2
Control Point (1, 2): 2 1 0
Control Point (1, 3): 3 1 2
Control Point (2, 0): 0 2 0
Control Point (2, 1): 1 2 1
Control Point (2, 2): 2 2 0
Control Point (2, 3): 3 2 1
Enter knot vector in U direction: 0 0 0 1 1 1
Enter knot vector in V direction: 0 0 0 1 1 1 1
Enter weights for control points:
Weight for Control Point (0, 0): 1
Weight for Control Point (0, 1): 2
Weight for Control Point (0, 2): 1
Weight for Control Point (0, 3): 2
Weight for Control Point (1, 0): 1
Weight for Control Point (1, 1): 1
```

Weight for Control Point (1, 2): 1

Weight for Control Point (1, 3): 1

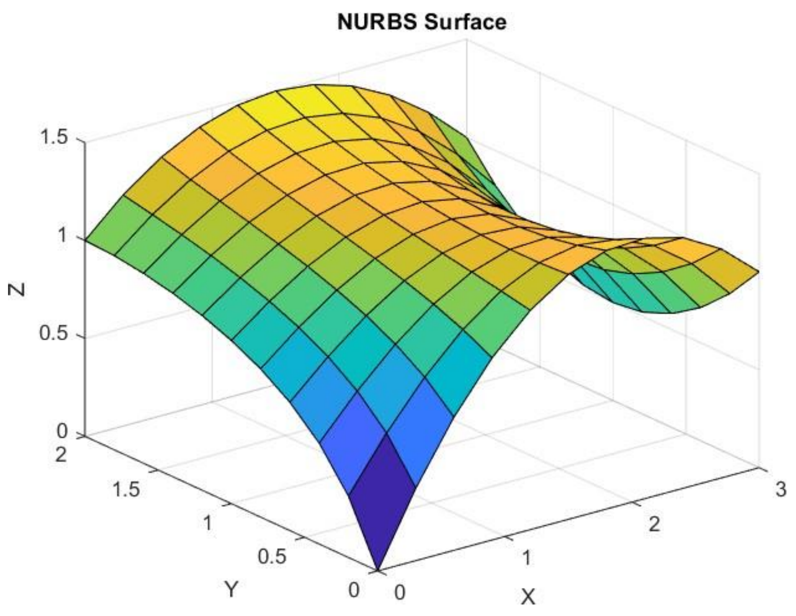
Weight for Control Point (2, 0): 1

Weight for Control Point (2, 1): 1

Weight for Control Point (2, 2): 1

Weight for Control Point (2, 3): 1 Enter degree in U direction: 2

Enter degree in V direction: 2 Enter step size: 0.1



Nurbs Surface Revolution

Introduction

- NURBS (Non-Uniform Rational B-Splines) are vital for creating smooth curves in computer graphics and design.
- The Cox-de Boor recursion formula is essential for NURBS curve generation, offering mathematical precision.

Understanding NURBS Curves

- Defined by control points, weights, and a knot vector.
- Degree of the curve influences how neighboring control points shape the curve.

Cox-de Boor Recursion Formula

- Core algorithm for calculating basis functions that determine control points' influence.
- Basis functions are recursive combinations of lower-degree basis functions, enabling efficient computation.

Calculation Process

- Iterate through control points, calculating their influence using corresponding basis functions.
- Sum contributions across all control points, normalizing by the sum of weights to obtain the final curve position.

Fine Control with Weights

- Control point weights dictate their influence on nearby curve segments.
- Adjusting weights allows designers to refine curvature and smoothness characteristics.

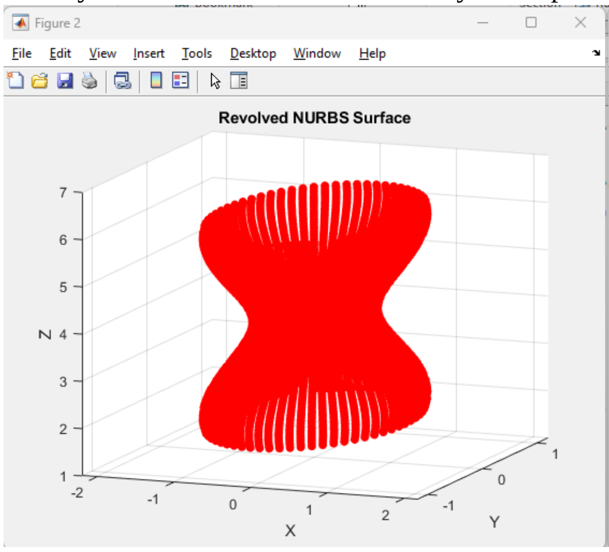
Conclusion

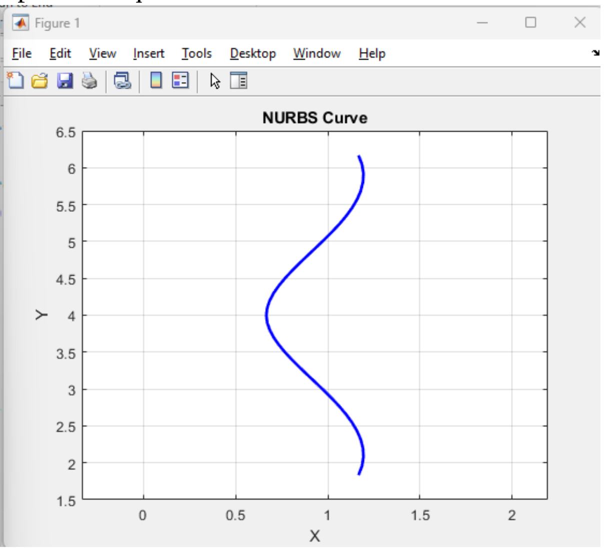
- Cox-de Boor recursion formula is fundamental for precise NURBS curve control.
- Understanding this algorithm empowers designers in various applications, from computer graphics to industrial design.

Running the Code

- Download C++ code and compile it in a compatible IDE.
- Input parameters such as curve degree, control points, weights, knot vector, and step size.
- Generate output files containing plotted NURBS curve and optionally, a 3D surface.

Exploration and Customization

- Experiment with different parameters to observe changes in the generated curve.
- Modify the code to extend functionality or adapt to specific requirements.
- 



Example
Degree of the NURBS curve:
Enter an integer value for the degree of the NURBS curve. For example, you can input: **3**
Number of Control Points:
Enter an integer value for the number of control points you want to define for the curve. For example, you can input: **5**
Coordinates and Weights of Control Points:
For each control point, enter the x and y coordinates followed by its weight. For example:
For control point 1: **2.5 1.8 0.9**
For control point 2: **-3.2 4.1 1.2**
For control point 3: **0.0 0.0 1.0**
For control point 4: **1.7 -2.3 0.8**
For control point 5: **-4.0 -1.5 1.5**
Knot Vector Values:
Enter the knot vector values. For example, if the degree is 3 and there are 5 control points, you might enter:
Knot value 1: **0.0**
Knot value 2: **0.0**
Knot value 3: **0.0**
Knot value 4: **1.0**
Knot value 5: **2.0**
Knot value 6: **3.0**
Knot value 7: **3.0**
Knot value 8**: 3.0**
Knot value 9: **3.0**
Step Size for Plotting the Curve:
Enter a floating-point value for the step size used to plot the curve. For example, you can input: **0.1**
Number of Grid Points:
Enter an integer value for the number of grid points you want to generate along the NURBS curve. For example, you can input: **20**

Knot Insertion and Refinement

1. Input Degree and Control Points: The program prompts the user to enter the "degree" of the spline. This degree influences the number of control points required.It initializes a 2D vector points to store these control points, each being a pair of x and y coordinates.
2. Input Original Knot Vector: The user is asked to input the knot vector. The size of this vector is computed based on the degree and the number of control points.
3. Knot Insertion Instructions: The program explains how to insert new knots for either knot insertion or refinement. The user must ensure that the new knot values are within appropriate bounds to prevent runtime errors.
4. Input New Knot Values: The user inputs new knots which will be inserted into the original spline. For each new knot, the code finds the appropriate span in the original knot vector where the new knot should be inserted. Using the de Boor's algorithm, new control points are calculated and added based on the insertion point and the degree of the spline.
5. Update and Display New Control Points: After inserting all new knots and calculating the new control points, these points are displayed to the user.
6. Update and Display Knot Vector: The knot vector is also updated by inserting the new knots in their correct positions. This updated vector is then displayed.

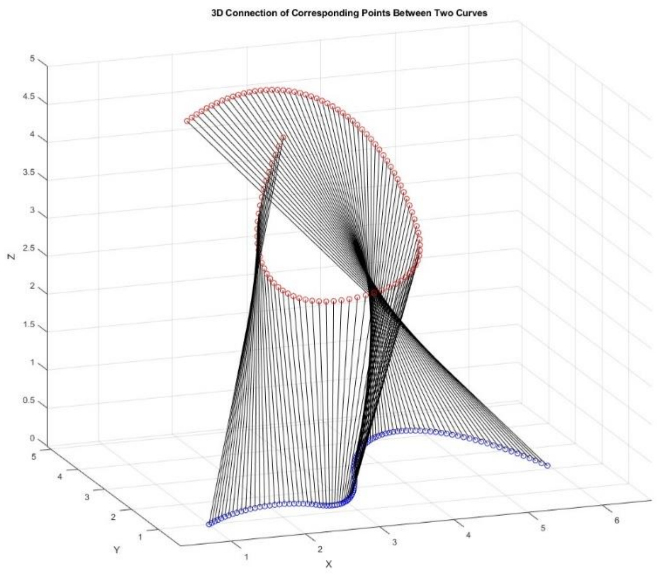
Ruled Surfaces

1. **Interpolation Function:** The **interpolate** function implements spline interpolation, taking parameters such as **t** (parameter along the curve), **degree** (degree of the spline), **points** (control points), **knots** (knot vector), and optional **weights** for control points. It utilizes Cox-de Boor's algorithm to compute interpolated points along the spline curve based on the provided inputs.

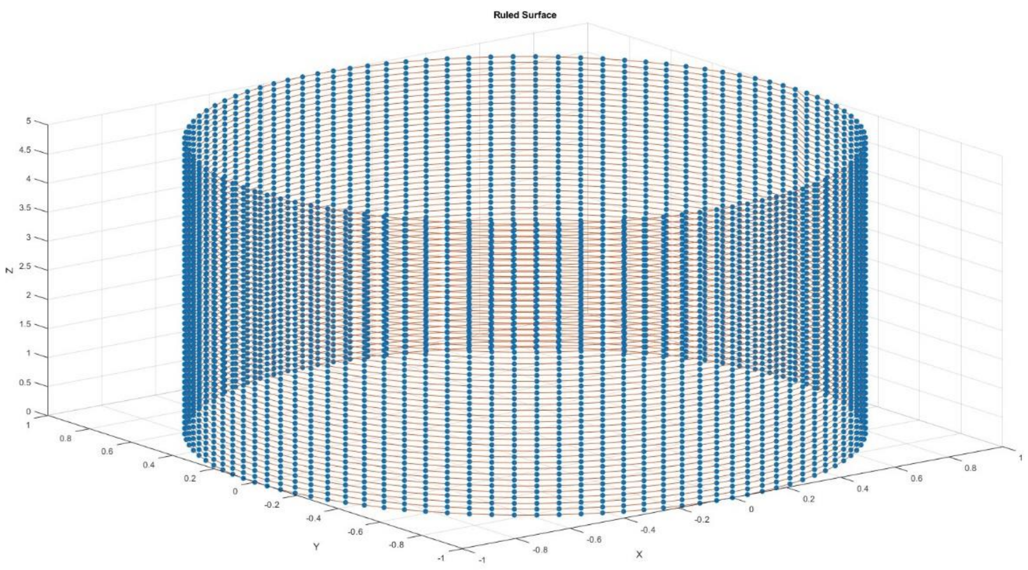
2. **Main Function:** In the **main** function:

- Control points for **curve1** and **curve2** are defined in **points1** and **points2** respectively.
- Knot vectors **knots** and **knots2**, and weights **weights** are provided.
- For each curve, the code iterates over the parameter space from 0 to 1, computing interpolated points using the **interpolate** function.
- The computed points are stored in output files **curve1.txt** and **curve2.txt**, with each point formatted as **(x, y, z)**.

3. **Output and Visualization:** The interpolated points are written to output files, enabling visualization or further analysis. Each point is written in a format suitable for plotting or processing.



Ruled Surface



General Cylinder