

Bézier Curve Transformation into Polynomial Functions Utilizing System of Linear Equations

Nadhif Radityo Nugroho (13523045)^{1,2}

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia

¹13523045@mahasiswa.itb.ac.id, ²nadhifradityo@gmail.com

Abstract—Bézier curves, well-known curves mainly used in computer graphics, have shown their advantages to designers and software developers. Simplicity in their mathematical representation allows adjustable performance based on screen or zoom resolution. However, problem arises when further analyses are required from the curves. A simple sampling does not yield significant result on the information it carries. This paper proposes a method to Bézier curve parameterization utilizing linear algebra principles. The method successfully converts Bézier curves into polynomials function with minimal sampling and errors within acceptable margin. The generated polynomial functions that can be further analyzed offering potential breakthroughs in mathematical analysis, computational simplification, and integration into other systems.

Index Terms—Bézier curves, polynomial functions, Gauss-Jordan elimination

I. INTRODUCTION

Bézier curves have been a breakthrough in computer graphics, especially in photo editing software. Their usage has been proven to ease designers and software developers with their simple mathematical concept and relatively fast rendering. Bézier curves also have dynamic performance since its resolution can be adjusted at runtime depending on the screen or zoom resolution. Despite their simplicity, a usual dilemma with Bézier curves usually comes when determining its curve parameter.

A naive approach to analyzing Bézier curves usually involves Newton's method to find local minima or maxima, followed by sampling near its critical point. However, this approach introduces computational inefficiencies and often lacks flexibility for dynamic rendering needs. Thus, there is a growing demand for alternative methods that optimize Bézier curve parameterization while maintaining computational efficiency, that can offer potential breakthroughs in mathematical analysis, computational simplification, and integration into other systems.

This paper proposes a method to transform Bézier curves into a set of polynomial functions using a systematic approach utilizing the principles of linear algebra. In brief, the method mainly consists of lightly sampling the Bézier curve, converting sampled points into a matrix, and finally using matrix inversion concepts from systems of linear equations to determine the polynomial coefficients. By further analyzing the derivative of the generated polynomial functions, the

solution to determine curve parameter can be revealed. A zero crossing in the derivative on the x-axis indicates a turning point on the curve. Using this description, dynamic optimization and more efficient rendering processes can be achieved.

II. THEORETICAL FOUNDATIONS

A. Bézier Curve

Bézier curves are defined by a collection of points. A single variable that, in the range of 0 through 1, defines a point interpolated from that interval. This method was invented by a French engineer Pierre Bézier. Bézier curves in mathematical terms, are defined this way:

$$\mathbf{B}(t) = \sum_{i=0}^n \binom{n}{i} (1-t)^{n-i} t^i \mathbf{P}_i \quad (1)$$

whereas

- $t \in [0, 1]$ is the parameter.
- \mathbf{P}_i are the control points.
- $\binom{n}{i} = \frac{n!}{i!(n-i)!}$ is the binomial coefficient.

Expanded low-degrees Bézier curves can be expressed as follows:

- Linear Bézier curve:

$$\mathbf{B}(t) = (1-t)\mathbf{P}_0 + t\mathbf{P}_1 \quad (2)$$

- Quadratic Bézier curve:

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2 \quad (3)$$

- Cubic Bézier curve:

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3 \quad (4)$$

This curve is majorly used in computer graphics, primarily photo editing software. Because the step in which the interval variable used is user-defined, the resolution of said curve can be dynamic. This is a huge advantage in computer graphics, as the rastered image can be calculated in real-time, giving the illusion of infinite details. Another advantage is that the Bézier curves allow designers to have more control over their designs with the help of simple control points that they can utilize with freedom.

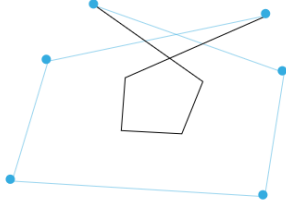


Fig. 1: 6 Control Points Bézier Curve with 5 Sample Points

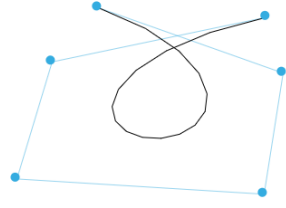


Fig. 2: 6 Control Points Bézier Curve with 17 Sample Points

B. Matrix Inversion Utilizing Gauss-Jordan

Matrices are two dimensional arrays containing numbers that can be arranged in terms of rows and columns. Representing a fundamental structure in linear algebra, matrix describes each of its rows to linear equations and columns to variables. With this concept in mind, an important property of matrix can be derived. The inverse of a matrix is a key to solving system of equations and numerous applications in mathematical modelling and computational algorithms. Inverse of a square matrix M , denoted as M^{-1} , satisfies the equality $M \cdot M^{-1} = I$, where I is the identity matrix with its dimension the same as matrix M .

Matrix inversion property helps convert a set of points sampled from Bézier curves to polynomial functions. In most cases, a single Bézier curve, respecting its complexity, cannot be expressed with single polynomial functions, one might expect performance-efficient algorithm as demand for larger application grows. Using Gauss-Jordan method allows computation of matrix inversion rather efficiently. Gauss-Jordan method starts by augmenting square matrix with its identity matrix, then performing a sequence of row operations, such as row swapping, scaling, and addition, to transform the original matrix into the identity matrix. This algorithm aligns with a systematic and precise approach that integrates seamlessly into this paper topic, making it an optimal choice for tasks requiring efficiency and reliability.

In mathematical terms:

- Form the Augmented Matrix: Combine A and the identity

matrix I to form $[A | I]$

$$[A | I] = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & 1 & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & a_{2n} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & 0 & 0 & \cdots & 1 \end{bmatrix} \quad (5)$$

- Row Operations: Apply elementary row operations to transform the left side into the identity matrix

- Row Addition (Adding a Multiple of One Row to Another):

$$R_i \leftarrow R_i + \alpha R_j \quad (6)$$

where α is a scalar.

- Scaling (Multiplying a Row by a Non-Zero Scalar):

$$R_i \leftarrow \alpha R_i \quad \text{where } \alpha \neq 0 \quad (7)$$

- Row Swapping (Interchange):

$$R_i \leftrightarrow R_j \quad (8)$$

- Transformation: After completing the row operations, the augmented matrix becomes

$$[I | A^{-1}] = \begin{bmatrix} 1 & 0 & \cdots & 0 & b_{11} & b_{12} & \cdots & b_{1n} \\ 0 & 1 & \cdots & 0 & b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix} \quad (9)$$

- Extract the Inverse: The right-hand side of the augmented matrix is the inverse of A

$$A^{-1} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix} \quad (10)$$

III. METHODOLOGY

Algorithmic approach is majorly used in Bézier curves transformation into polynomial functions through a systematic approach. The use of samplings, matrix inversion calculations, error margin caps, and retrials are heavily used. With that said, here are the main outlines of this paper:

- Sampling the Bézier Curve: Introduces a collection of independent x-coordinates and y-coordinates pair of points that will be fed into the next step.
- Constructing the Polynomial Matrix: Converts sequence of x-coordinates and y-coordinates along with its sampling variable into a matrix.
- Computing the Polynomial Coefficients: From the matrix constructed in the previous step, the coefficients are computed by inverting the matrix.
- Error Analysis: Polynomial functions generated are compared against the original Bézier curve. A retrial may occur but with utilizing critical points derived from generated polynomial functions.

IV. IMPLEMENTATION

This paper uses an algorithmic methodology to transform Bézier curves into polynomial functions efficiently. Systemic approach involving sampling, matrix inversion, and iterative refinement to minimize error, are used in this algorithm. Key steps include sampling the Bézier curve to obtain x-coordinate and y-coordinate pairs along with its parametric interval, constructing a polynomial matrix using these sampled points, and computing polynomial coefficients via matrix inversion. Finally, perform an error analysis by comparing the generated polynomial functions against the original Bézier curve. If the errors are outside the margin, retrials are incorporated utilizing critical points from the derived polynomial functions to enhance accuracy.

A. Sampling the Bézier Curve

Sampling involves the step of taking a curve parameter in the range of 0 through 1. At 0, it sits exactly on the first control point. Conversely at 1, it sits exactly on the last control point. The value between 0 through 1 describes the interpolated value between many points.

With no educated guess provided, the algorithm initially samples the curve naively with fixed increment. This initial sampling produces a set of independent x-coordinates and y-coordinates points. These coordinates must be typically the same length, as the curve parameters are the same regardless of independent sampling. These points will later be fed to a matrix accordingly in the next step.

Algorithm 1 Sample Bézier Curve Point
implements: Equation 1

Require: points = $\{p_0, p_1, \dots, p_n\}$, $t \in [0, 1]$
Ensure: (x, y)
 $x \leftarrow 0$
 $y \leftarrow 0$
 $n \leftarrow \text{length of points} - 1$
for $i \leftarrow 0$ to n **do**
 $B(n, i) \leftarrow \frac{n!}{i! \cdot (n-i)!}$
 $\text{basis}(n, i, t) \leftarrow B(n, i) \cdot (1-t)^{n-i} \cdot t^i$
 $x \leftarrow x + \text{basis}(n, i, t) \cdot x_i$
 $y \leftarrow y + \text{basis}(n, i, t) \cdot y_i$
end for
return (x, y)

B. Constructing the Polynomial Matrix

The polynomial square matrix is constructed by evaluating each curve parameters raised to powers decreasing from the highest degree to zero. This matrix construction is commonly known as Vandermonde matrix. In mathematical terms, let n be the number of points, and let the points be $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$. And its form can be expressed as:

$$M = \begin{bmatrix} x_1^{n-1} & x_1^{n-2} & \dots & x_1^0 \\ x_2^{n-1} & x_2^{n-2} & \dots & x_2^0 \\ \vdots & \vdots & \ddots & \vdots \\ x_n^{n-1} & x_n^{n-2} & \dots & x_n^0 \end{bmatrix} \quad (11)$$

C. Computing the Polynomial Coefficients

To determine the coefficients of the polynomial, the matrix from previous step is then inverted and multiplied by a column matrix containing the target sampled values. The sampled values are x-coordinates and y-coordinates which both get computed independently. To solve the polynomial coefficients $a = [a_{n-1} \ a_{n-2} \ \dots \ a_0]$, the equation is:

$$a = M^{-1} \cdot y \quad (12)$$

where M is the Vandermonde matrix from previous step, and y is $[y_1 \ y_2 \ \dots \ y_n]$.

Algorithm 2 Compute Polynomial Coefficients
implements: Equation 12

Require: intervals, sampledPoints
Ensure: $xCoefficients$, $yCoefficients$
 $n \leftarrow \text{length}(\text{intervals})$
 $m \leftarrow \text{length}(\text{sampledPoints})$
 $\text{vandermondeMatrix} \leftarrow \text{Matrix}(n, n)$
for $i \leftarrow 0$ to $n - 1$ **do**
for $j \leftarrow 0$ to $n - 1$ **do**
 $\text{vandermondeMatrix}[i][j] \leftarrow \text{intervals}[i]^{n-1-j}$
end for
end for
 $\text{targetXMatrix} \leftarrow \text{Matrix}(m, 1)$
for $i \leftarrow 0$ to $m - 1$ **do**
 $\text{targetXMatrix}[i][0] \leftarrow \text{sampledPoints}[i][0]$
end for
 $\text{targetYMatrix} \leftarrow \text{Matrix}(m, 1)$
for $i \leftarrow 0$ to $m - 1$ **do**
 $\text{targetYMatrix}[i][0] \leftarrow \text{sampledPoints}[i][1]$
end for
 $\text{inverseVandermondeMatrix} \leftarrow \text{invert}(\text{vandermondeMatrix})$
 $xCoefficients \leftarrow \text{inverseVandermondeMatrix} \cdot \text{targetXMatrix}$
 $yCoefficients \leftarrow \text{inverseVandermondeMatrix} \cdot \text{targetYMatrix}$

D. Error Analysis

The generated polynomial function may deviate from the original Bézier curve. The need of error analysis is important as it allows metrics computation to fit the error within margin.

Algorithm 3 Calculate Error

Require: bezierResolution , $\text{polynomialResolution}$
Ensure: score
 $\text{bezierPoints} \leftarrow \text{new array with size } \text{bezierResolution}$
for $i = 0$ to bezierResolution **do**
 $t_b \leftarrow \frac{i+1}{\text{bezierResolution}}$
 $\text{bezierPoints}[i] \leftarrow \text{sampleBezierCurve}(\text{points}, t_b)$
end for
 $\text{polynomialPoints} \leftarrow \text{new array with size } \text{polynomialResolution}$
for $i = 0$ to $\text{polynomialResolution}$ **do**
 $t_p \leftarrow \frac{i+1}{\text{polynomialResolution}}$
 $x_p \leftarrow \text{computePolynomials}(xCoefficients, t_p)$

```

 $y_p \leftarrow \text{computePolynomials}(y\text{Coefficients}, t_p)$ 
 $polynomialPoints[i] \leftarrow [x_p, y_p]$ 
end for
 $score \leftarrow 0$ 
for  $p \in polynomialPoints$  do
   $minDist \leftarrow \infty$ 
  for  $b \in bezierPoints$  do
     $dist \leftarrow \sqrt{(p[0] - b[0])^2 + (p[1] - b[1])^2}$ 
     $minDist \leftarrow \min(minDist, dist)$ 
  end for
   $score \leftarrow score + minDist$ 
end for

```

E. Retrying Sample

Given in the initial sampling with no educated guess, the generated polynomials may deviate too much from actual Bézier curve. Fortunately within the generated polynomials, a set of helpful hints can be derived from its derivative. This includes the rate of curve parameter contributing to sampled point. Given an arbitrary intervals, a curve parameter may have big distance from the last sampled point. This problem mainly leads to uneven details on the Bézier curve.

Algorithm 4 Distribute Intervals

Require: $coefficients, integrationResolution, intervals$
Ensure: $clumpedIntervals$

```

 $integration \leftarrow 0$ 
for  $i = 0$  to  $integrationResolution - 1$  do
   $x \leftarrow \frac{i}{integrationResolution}$ 
   $polyValue \leftarrow \text{computePolynomials}(coefficients, x)$ 
   $integration \leftarrow integration + \frac{polyValue}{integrationResolution}$ 
end for
 $normalizedCoeff \leftarrow$  array of size
   $length(coefficients)$ 
for  $i = 0$  to  $length(coefficients) - 1$  do
   $coeff \leftarrow coefficients[i]$ 
   $normalizedCoeff[i] \leftarrow \frac{coeff}{integration}$ 
end for
 $integratedNormalizedCoeff \leftarrow$  array of size
   $length(normalizedCoeff) + 1$ 
for  $i = 0$  to  $length(normalizedCoeff) - 1$  do
   $normalizedCoeffValue \leftarrow normalizedCoeff[i]$ 
   $integratedNormalizedCoeff[i] \leftarrow \frac{normalizedCoeffValue}{i+1}$ 
end for
 $integratedNormalizedCoeff$ 
 $[length(normalizedCoeff)] \leftarrow 0$ 
 $cdf \leftarrow x \mapsto \text{computePolynomials}$ 
   $(integratedNormalizedCoeff, x)$ 
 $ppf \leftarrow \text{inverse}(cdf)$ 
 $clumpedIntervals \leftarrow$  new array of size  $intervals$ 
for  $i = 0$  to  $intervals - 1$  do
   $t \leftarrow \frac{i+1}{intervals}$ 
   $clumpedIntervals[i] \leftarrow ppf(t)$ 
end for

```

V. RESULTS AND DISCUSSION

To make the implementation more clear, let's take a look at one example. This example will show how algorithms in previous section work hand-in-hand. Say we have a Bézier curve with these control points:

$$\begin{bmatrix} (394.6955, 256.6955), (305, 328), (504, 341), \\ (625.5372, 179.7772), (370.4139, 177.0606), \\ (251.6408, 326), (447.1015, 324.203), (576, 328), \\ (703.3859, 228.1298), (555.0546, 228.4278), \\ (378.7305, 187.3664), (398.0841, 378.5589) \end{bmatrix} \quad (13)$$

Initially, we sample the Bézier curve using Algorithm 1 with fixed intervals. With $n = 4$, initial intervals and sampled points respectively will be:

$$[0.25, 0.5, 0.75, 1] \quad (14)$$

$$\begin{bmatrix} (455.8376, 263.7338) & (449.6922, 279.2926) \\ (545.1082, 255.4035) & (398.0841, 378.5589) \end{bmatrix} \quad (15)$$

In this step, we get the raw visualization of the Bézier curve:

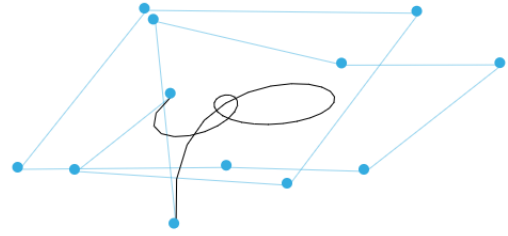


Fig. 3: A Sample Input Bézier Curve

Construct the Vandermonde matrix using Equation 11, then inverse it utilizing Gauss-Jordan elimination.

$$T = \begin{bmatrix} 0.015625 & 0.0625 & 0.25 & 1 & 1 \\ 0.125 & 0.25 & 0.5 & 1 & 1 \\ 0.421875 & 0.5625 & 0.75 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \quad (16)$$

- Write the augmented matrix:

$$\begin{bmatrix} 0.015625 & 0.0625 & 0.25 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0.125 & 0.25 & 0.5 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0.421875 & 0.5625 & 0.75 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

- Find the pivot in the 1st column and swap the 4th and the 1st rows:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0.125 & 0.25 & 0.5 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0.421875 & 0.5625 & 0.75 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0.015625 & 0.0625 & 0.25 & 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

- Eliminate the 1st column:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0.125 & 0.375 & 0.875 & 0 & 1 & 0 & -0.125 & 0 \\ 0 & 0.1406 & 0.3281 & 0.5781 & 0 & 0 & 1 & -0.4218 & 0 \\ 0 & 0.0468 & 0.2343 & 0.9843 & 1 & 0 & 0 & -0.0156 & 0 \end{bmatrix}$$

- Make the pivot in the 2nd column by dividing the 2nd row by 0.125:

$$\left[\begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 3 & 7 & 0 & 8 & 0 & -1 \\ 0 & 0.1406 & 0.3281 & 0.5781 & 0 & 0 & 1 & -0.4218 \\ 0 & 0.0468 & 0.2343 & 0.9843 & 1 & 0 & 0 & -0.0156 \end{array} \right]$$

- Eliminate the 2nd column:

$$\left[\begin{array}{cccc|cccc} 1 & 0 & -2 & -6 & 0 & -8 & 0 & 2 \\ 0 & 1 & 3 & 7 & 0 & 8 & 0 & -1 \\ 0 & 0 & -0.093 & -0.406 & 0 & -1.125 & 1 & -0.281 \\ 0 & 0 & 0.093 & 0.656 & 1 & -0.375 & 0 & 0.031 \end{array} \right]$$

- Make the pivot in the 3rd column by dividing the 3rd row by -0.09375:

$$\left[\begin{array}{cccc|cccc} 1 & 0 & -2 & -6 & 0 & -8 & 0 & 2 \\ 0 & 1 & 3 & 7 & 0 & 8 & 0 & -1 \\ 0 & 0 & 1 & 4.3333 & 0 & 12 & -10.66 & 3 \\ 0 & 0 & 0.093 & 0.656 & 1 & -0.375 & 0 & 0.031 \end{array} \right]$$

- Eliminate the 3rd column:

$$\left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 2.6666 & 0 & 16 & -21.33 & 8 \\ 0 & 1 & 0 & -6 & 0 & -28 & 32 & -10 \\ 0 & 0 & 1 & 4.3333 & 0 & 12 & -10.66 & 3 \\ 0 & 0 & 0 & 0.25 & 1 & -1.5 & 1 & -0.25 \end{array} \right]$$

- Make the pivot in the 4th column by dividing the 4th row by 0.25:

$$\left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 2.6666 & 0 & 16 & -21.33 & 8 \\ 0 & 1 & 0 & -6 & 0 & -28 & 32 & -10 \\ 0 & 0 & 1 & 4.3333 & 0 & 12 & -10.66 & 3 \\ 0 & 0 & 0 & 1 & 4 & -6 & 4 & -1 \end{array} \right]$$

- Eliminate the 4th column:

$$\left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & -10.6667 & 32 & -32 & 10.6667 \\ 0 & 1 & 0 & 0 & 24 & -64 & 56 & -16 \\ 0 & 0 & 1 & 0 & -17.3333 & 38 & -28 & 7.3333 \\ 0 & 0 & 0 & 1 & 4 & -6 & 4 & -1 \end{array} \right]$$

$$T^{-1} = \begin{bmatrix} -10.6667 & 32 & -32 & 10.6667 \\ 24 & -64 & 56 & -16 \\ -17.3333 & 38 & -28 & 7.3333 \\ 4 & -6 & 4 & -1 \end{bmatrix} \quad (17)$$

Finally we compute the polynomial coefficients for both axes using Algorithm 2.

$$\begin{aligned} X_{coeff} &= T^{-1} \cdot \begin{bmatrix} 455.838 \\ 449.692 \\ 545.108 \\ 398.084 \end{bmatrix} = \begin{bmatrix} -3669.350 \\ 6316.517 \\ -3156.629 \\ 907.546 \end{bmatrix} \\ Y_{coeff} &= T^{-1} \cdot \begin{bmatrix} 263.734 \\ 279.293 \\ 255.403 \\ 378.559 \end{bmatrix} = \begin{bmatrix} 1989.254 \\ -3299.464 \\ 1666.535 \\ 22.235 \end{bmatrix} \end{aligned} \quad (18)$$

With the nature of polynomial continuous function property, the generated polynomials are smooth even with small sample points. Fig. 4 shows $X(t)$ as black line representing x-movement and $Y(t)$ as red line representing y-movement. The resulting curve is in the blue line, this sample result is within error margin and visually similar compared to original Bézier curve Fig. 3.

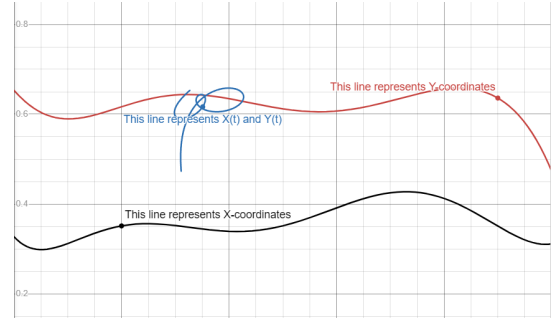


Fig. 4: The Generated Polynomials from Input Fig. 3

Tests in Table I are mainly driven by Bézier curve edge cases. This includes overlapping control points, collinear control points, and self-intersecting curves with multiple control points. These drivers ensure the algorithm proposed by this paper will work under any circumstances.

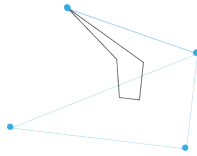

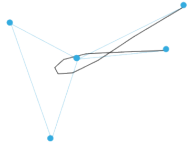
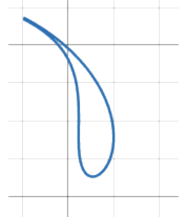

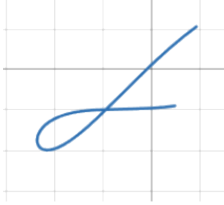
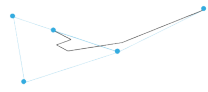

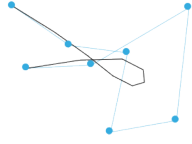
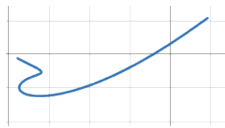


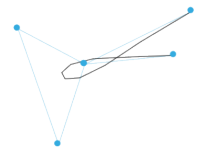
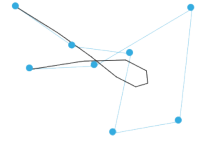




VI. CONCLUSION

This paper presents a method for parameterizing Bézier curves using principles of linear algebra. The proposed approach successfully converts Bézier curves into polynomial functions with minimal sampling and acceptable error margins, addressing the limitations of simple sampling techniques that fail to capture the curves' detailed edges.

The method involves a systematic process, beginning with the sampling of the Bézier curve to gather pairs of x- and y-coordinates. These coordinates are used to construct a polynomial matrix, which is then inverted to compute the polynomial coefficients. Finally, an error analysis step ensures the generated polynomial functions closely approximate the original Bézier curve. If discrepancies are detected, retrials utilizing critical points derived from the polynomial functions further enhance accuracy.

The generated polynomial functions offer a pathway for deeper mathematical analysis, computational simplification, and integration into other systems. This advancement in Bézier curve parameterization has the potential to significantly impact mathematical modeling and computational design, providing a robust tool for designers and developers.

TABLE I: Test Results

	Overlapping Control Points	Collinear Control Points	Self-Intersecting Curve
Overlapping Control Points	 <p>Fig. 5: Input Low-Sampled Bézier Curve for Overlapping-Overlapping Test Case</p>	 <p>Fig. 7: Input Low-Sampled Bézier Curve for Collinear-Overlapping Test Case</p>	 <p>Fig. 9: Input Low-Sampled Bézier Curve for Intersecting-Overlapping Test Case</p>
	 <p>Fig. 6: Output Parametric Polynomial from Overlapping-Overlapping Test Case</p>	 <p>Fig. 8: Output Parametric Polynomial from Collinear-Overlapping Test Case</p>	 <p>Fig. 10: Output Parametric Polynomial from Intersecting-Overlapping Test Case</p>
Collinear Control Points	 <p>Fig. 11: Input Low-Sampled Bézier Curve for Collinear-Overlapping Test Case</p>	 <p>Fig. 13: Input Low-Sampled Bézier Curve for Collinear-Collinear Test Case</p>	 <p>Fig. 15: Input Low-Sampled Bézier Curve for Collinear-Intersect Test Case</p>
	 <p>Fig. 12: Output Parametric Polynomial from Collinear-Overlapping Test Case</p>	 <p>Fig. 14: Output Parametric Polynomial from Collinear-Collinear Test Case</p>	 <p>Fig. 16: Output Parametric Polynomial from Collinear-Intersect Test Case</p>
Self-Intersecting Curve	 <p>Fig. 17: Input Low-Sampled Bézier Curve for Intersecting-Overlapping Test Case</p>	 <p>Fig. 19: Input Low-Sampled Bézier Curve for Collinear-Intersect Test Case</p>	 <p>Fig. 21: Input Low-Sampled Bézier Curve for Intersect-Intersect Test Case</p>
	 <p>Fig. 18: Output Parametric Polynomial from Intersecting-Overlapping Test Case</p>	 <p>Fig. 20: Output Parametric Polynomial from Collinear-Intersect Test Case</p>	 <p>Fig. 22: Output Parametric Polynomial from Intersect-Intersect Test Case</p>

APPENDIX

The code implementation for the methods and experiments discussed in this paper can be found at the following GitHub repository: <https://github.com/NadhifRadityo/poly-bezier-transform>.

This repository contains:

- The source code for the poly-bezier transformation techniques described in the paper.
- Instructions for setting up the development environment and running the code.
- Example datasets and scripts for replicating key results presented in this study.

Please do explore the repository for a deeper understanding of the implementation details and for any potential extensions of the methodology. For issues or questions regarding the repository, please refer to the provided documentation or contact the repository maintainer directly.

ACKNOWLEDGMENT

The author would like to express profound gratitude to Ir. Rila Mandala, M.Eng., Ph.D. along with Dr. Ir. Rinaldi Munir, M.T. for their invaluable guidance and insights as IF2123 Linear Algebra and Geometry course lecturers, which greatly contributed to the development of this paper.

The author also extends apologies for any shortcomings that may remain in this work. It is sincerely hoped that this paper will serve as a useful reference for future studies and research purposes.

REFERENCES

- [1] Baydas, S., Karakas, B. 2019. Defining a curve as a Bézier curve. Journal of Taibah University for Science.
- [2] Matusik, W. 2012. MIT OpenCourseWare: Bézier Curves and Splines. Massachusetts Institute of Technology.
- [3] Melo, M. 2021. Understanding Bézier Curves.
- [4] Munir, R. 2023. Sistem Persamaan Linier (SPL): Metode Eliminasi Gauss-Jordan.
- [5] Strang, G. 2009. Introduction to Linear Algebra.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 31 Desember 2024



Nadhif Radityo N. (13523045)