UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 1187

DESIGN AND VISUALISATION OF MOBILE ROBOT PATHS

Ana Marija Devčić

Zagreb, February 2026

UNIVERSITY OF ZAGREB

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 1187

DESIGN AND VISUALISATION OF MOBILE ROBOT PATHS

Ana Marija Devčić

Zagreb, February 2026

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 06 October 2025

MASTER THESIS ASSIGNMENT No. 1187

Student:	**Ana Marija Devčić (0036524526)**
Study:	Computing
Profile:	Computer Science
Mentor:	prof. Željka Mihajlović, PhD

Title:	**Design and visualisation of mobile robot paths**

Description:

The task is to study the design issues of robotic vehicles that transport pallets with goods in an industrial environment. In particular, study NURBS curves for the purpose of defining robot trajectories. Elaborate on the use of trajectory-defined curves and compare Bezier, B-spline, and NURBS curves for these purposes. Create a programming library of NURBS curves in C++ language. Visualize the planned trajectories of the robot. On several examples, analyze and evaluate the results. Discuss the usability of the solution as well as possible extensions. Create an appropriate software product. The results of the work should be made available via the Internet. Attach algorithms, source codes and results to the paper with the necessary explanations and documentation. Cite the literature used and state the help received.

Submission date: 13 February 2026

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 6. listopada 2025.

DIPLOMSKI ZADATAK br. 1187

Pristupnica:	**Ana Marija Devčić (0036524526)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Dizajn i vizualizacija putanja robotskih vozila**

Opis zadatka:

Proučiti problematiku dizajna putanja robotskih vozila koja prenose palete s robom u industrijskom okruženju. Posebice proučiti NURBS krivulje za potrebe definiranja putanja robota. Razraditi upotrebu krivulja kojima se definira putanja te usporediti Bezier, B-splajn i NURBS krivulje za ove potrebe. Načiniti programsku biblioteku NURBS krivulja u jeziku C++. Načiniti vizualizaciju planiranih putanja robota. Na nizu primjera ostvariti testiranje dobivenih rezultata. Analizirati i ocijeniti postignute rezultate. Diskutirati upotrebljivost rješenja kao i moguća proširenja. Izraditi odgovarajući programski proizvod. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 13. veljače 2026.

I would like to thank my mentor, dr.sc. Željka Mihajlović, for guiding me through my university career, my colleagues at Romb Technologies for supporting me in being a better developer, and my friends and loved ones for always believing in me.

Contents

Introduction

A rapid increase in online commerce has created a need for more automation in logistics. Many manufacturers and warehouses have started relying on fully or almost fully automated mobile robots instead of human labor, largely due to safety concerns and labor shortage, as well as a need for greater efficiency and precision than human labor can provide.

Traditionally, mobile robots, especially ones used for material handling in logistics and manufacturing, have been classified into two categories, automated guided vehicles (AGVs) and autonomous mobile robots (AMRs). AGVs operate on predefined paths while AMRs are free to plan their own path for every new mission. AGV-s often have a controller system which commands and controls all robots in the network. In some cases AMR-s might be the solution that requires less human intervention and input and may be more efficient, however AGV-s allow for more control and predictability. The line separating these two approaches is becoming increasingly blurred as some AGVs can now re-plan their paths around obstacles. This thesis is focused on pre-planned paths and will refer to the vehicles simply as mobile robots.

1 Mobile robot path design

In the past, AGV-s used physical rails, or magnetized or induction tracks, placed in the environment to guide their movement. Modern mobile robots have their paths defined in a controller application which sends commands to dictate their movement. While driving, the robots use various sensors to verify their velocity, position and orientation and correct their path if necessary. Some mobile robots require reflective stickers for navigation, though in other cases no changes need to be made to the environment. In that case the robot uses a combination of LIDAR sensors and computer vision to verify their position in space and their speed.

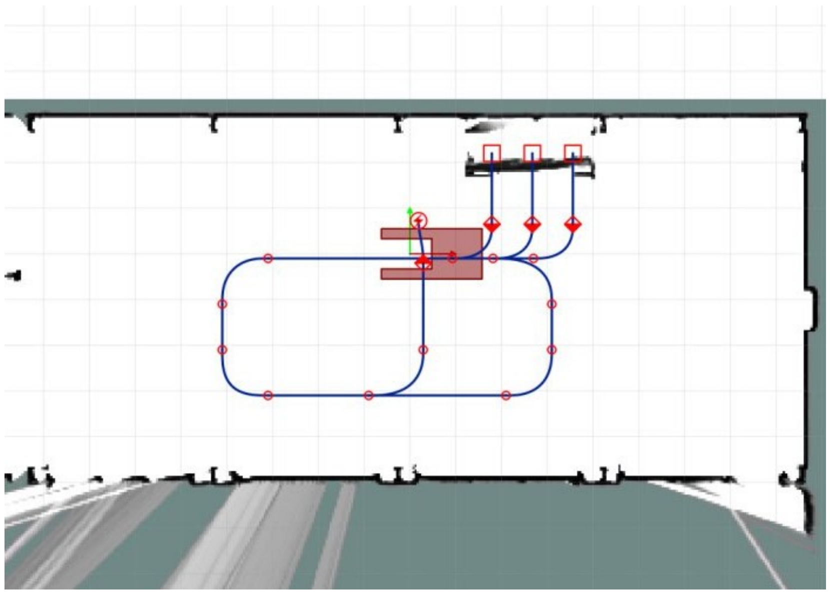


Figure 1.1: Example of a roadmap.

1.1 Path continuity

Two mathematical concepts relevant to mobile robot navigation are geometric continuity and parametric continuity of the robot's path. Parametric continuity of the n-th degree, or C^n, is achieved when the n-th derivative of a function exists and is continuous.

Geometric continuity of the n-th degree, or G^n, is a more relaxed form of parametric continuity, and is achieved if a function can be reparametrized as a C^n continuous function. [1]

A G^0 continuous function is one where all of the points on the curve are continuous, whereas a G^1 continuous function has a continuous tangent direction as well as position. A function with G^2 continuity may be appropriate for most uses. It has continuity in position, tangent direction and curvature. It differs from a C^2 continuous curve in that its derivatives are only continuous in direction, but a C^2 continuous curve would also have a continuous magnitude in speed and acceleration. [2]

For the navigation of multi-wheeled mobile robots it is important for the path to have a continuous curvature derivative. This is only possible using curves with G^3 continuity, as its definition requires the third derivative of the curve, which will be discussed in a later chapter. Following a path with a continuous curvature derivative means that the robot's steering angle is also continuous. If this were not the case, the robot would need to stop and reposition itself whenever the continuity was broken.

The curvature derivative is also important because it indicates how much and how sharply the mobile robot must turn its steering wheels. A high curvature derivative means that the robot needs to rotate its steering wheel a greater angle in a shorter amount of time, which may put strain on the robot and be dangerous for the people around it, especially if carrying heavy or unstable cargo. In addition, robots have a physical limit as to how quickly they can turn based on their size and architecture. [3]

A curve appropriate for mobile robot path design needs to have three attributes: it must have geometric continuity of at least the third order, it must be easy to manipulate to maximize the usage of available space, and it must be able to be shaped in such a way to minimize the derivative of the curvature.

2 Parametric curves and splines

Parametric curves are curves defined in a vector space by a parametric equation, which can be as simple as a parabola or a circle. In an n-dimensional space, these functions are often defined by one input — most commonly t or u, limited to the interval $[0, 1]$ — and n outputs. Each of the n coordinates is its own independent function, and the combination of multiple coordinates creates a curve, plane, or hyperplane.

Splines are a special kind of parametric curve that are defined as piecewise polynomials and are most often used to interpolate between a set of points.

For the purpose of mobile robot path design, the most appropriate types of splines are Bézier curves, B-splines, and NURBS curves. What these three models have in common is that each curve is defined by its degree, a set of control points in n-dimensional space and its basis functions. The position for a given t is a weighted sum of the curve's points, the weights being given by its basis functions. In other words, the curve interpolates between all of its given points.

In literature, the terms *degree* and *order* are both used to describe polynomial curves and splines. While these terms are very closely linked they are not interchangeable. The term *degree* is most commonly used in mathematical literature because it describes the actual degree of the curve's polynomial. The term *order*, on the other hand, is used in CAGD literature and describes the order of continuity along the curve, which is more relevant than the degree of the polynomial in engineering. The degree is always equal to the order minus one. The term *degree* will be used going forward for consistency.

2.1 Bézier

Bézier curves are the simplest of the three. The most commonly used types of Bézier curves are quadratic and cubic curves, defined by three and four points respectively. The degree of a Bézier curve is always equal to the number of control points minus one.

$$C(u) = \sum_{i=0}^{n} B_{i,n}(u) P_i \tag{2.1}$$

A Bézier curve's basis functions, or blending functions, are known as Bernstein polynomials. There are $n+1$ basis functions for a curve of the n-th degree. [4] In other words, there are as many polynomials as there are control points. Bernstein polynomials are defined only by the curve's degree, meaning that all Bézier curves of the same degree have the exact same Bernstein polynomials.

$$B_{i,n}(u) = \frac{n!}{i!(n-i)!} u^i (1-u)^{n-i} \tag{2.2}$$

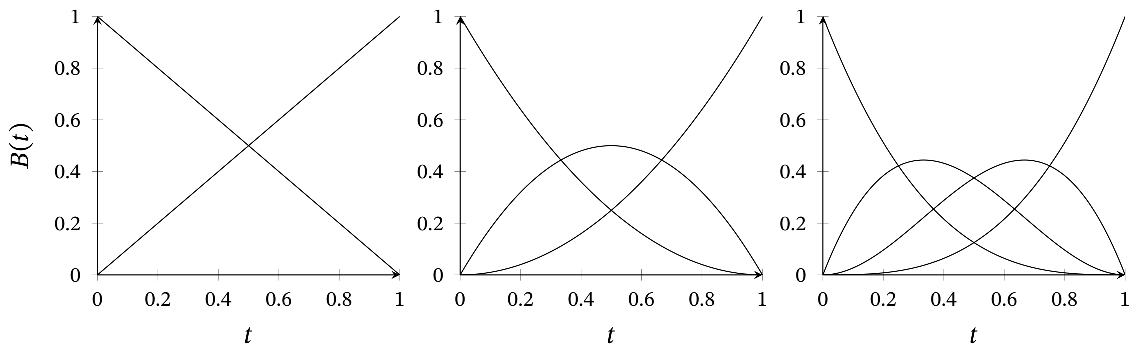


Figure 2.1: Basis functions for a Bézier curve of degree 1, 2 and 3

Figure 2.1 shows three examples of Bernstein polynomials, for a Bézier curve of degrees 1, 2 and 3 respectively. This shows how a given curve has as many Bernstein polynomials as there are control points, and each polynomial represents the influence that a control point has on the curve at a given parameter t. A Bernstein polynomial must be non-zero across the whole parameter range of $[0, 1]$, meaning that all control points have influence over the entire curve. This makes Bézier curves unsuitable for defining long, continuous paths as they do not have *local control*. Local control is defined as the ability to modify a specific part of a curve without affecting the rest of the curve.

2.2 B-spline

B-splines, short for “basis splines,” are a generalization of Bézier curves. Unlike Bézier curves, which have static basis functions dependent on degree, the basis functions of a B-spline can be modified using what is called a *knot vector*. The number of control points on a B-spline is arbitrary and not dependent on its degree, unlike Béziers.

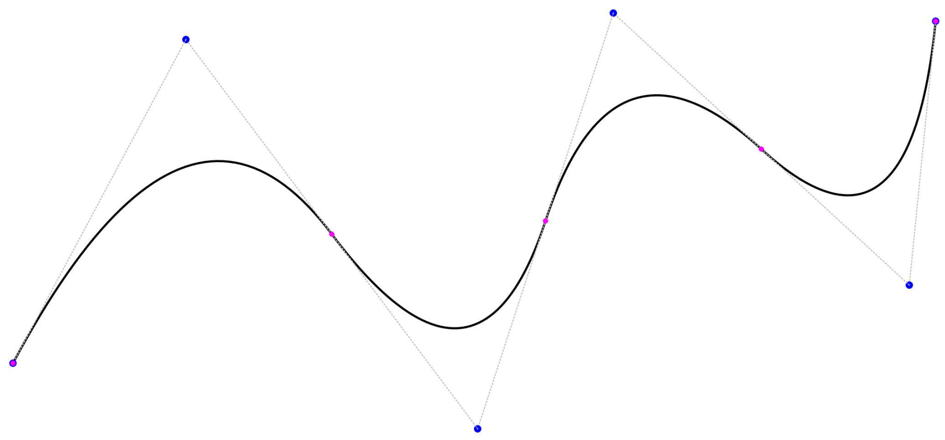


Figure 2.2: A second degree B-spline with six control points.

We call B-splines *piecewise polynomial* splines because they are composed of a series of polynomial segments. This is different from Bézier curves which are, by definition, a single polynomial.

$$C(u) = \sum_{i=0}^{n} B_{i,n}(u) P_i \tag{2.3}$$

The basis functions of a B-spline are not necessarily non-zero in the entire range $[0, 1]$, meaning that a given basis function may not have influence on the entire curve, i.e. a given point on the curve may not be defined by all of its control points. This makes it possible to create complex shapes using a single B-spline.

$$\begin{aligned} B_{i,p}(t) &= \frac{t - t_i}{t_{i+p} - t_i} B_{i,p-1}(t) + \frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}} B_{i+1,p-1}(t) \\ B_{i,0}(t) &= \begin{cases} 1 & t_i \leq t < t_{i+1} \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{2.4}$$

A B-spline of degree p and with N control points has a knot vector of size $N + p + 1$. Each control point affects p knots. By moving a knot's value up or down, we can make a section of the curve denser in control points while making the rest more sparse. Each span between two knots, i.e. between two values of t, can be seen as a separate curve controlled by $p + 1$ points, much like a Bézier. However, this is a much better solution than chaining Bézier curves because the model guarantees geometric continuity between knot spans, while a chain of Béziers needs to be adjusted manually to be continuous.

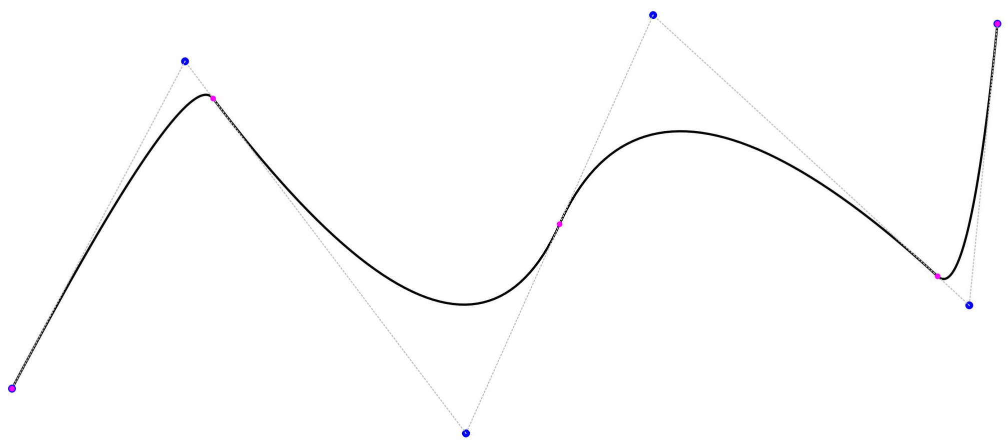


Figure 2.3: A second degree B-spline with six control points and a modified knot vector.

$$\mathbf{U} = \begin{bmatrix} 0 & 0 & 0 & 0.25 & 0.5 & 0.75 & 1 & 1 & 1 \end{bmatrix}$$

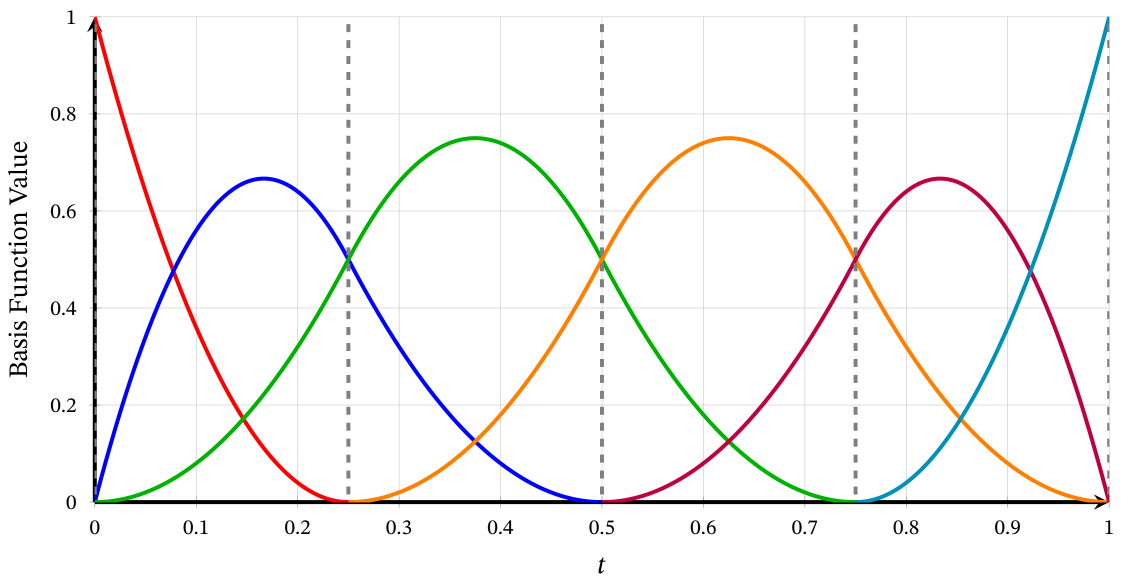


Figure 2.4: Example basis functions for a quadratic B-spline

Figure 2.4 shows the basis functions of a uniform second degree B-spline with six control points. Each basis function is drawn in a different color, and the dashed vertical lines mark the edges between individual knot spans. One thing that can be noticed in the graph is that each knot span has exactly three "active" or non-zero basis functions at a time. Much like a Bézier curve, each knot span of degree p is controlled by $p+1$ points, but B-splines differ from a chain of Bézier curves in the fact that the basis functions are continuous from one span to another. A chain of Bézier curves can only have the desired degree of geometric continuity if its control points are placed in a specific manner, whereas a B-spline will be continuous across its spans by definition.

One important property of B-splines, mentioned earlier, is that Bézier curves are a subset of B-splines. By manipulating the knot vector, we can effectively create a Bézier curve or a chain of Bézier curves from a single B-spline. Figure 2.5 shows the basis functions of a quadratic B-spline with six points and a knot vector of [0 0 0 0.5 0.5 0.5 1 1 1]. This also shows how a knot with a multiplicity of $p+1$ results in a non-differentiable peak.

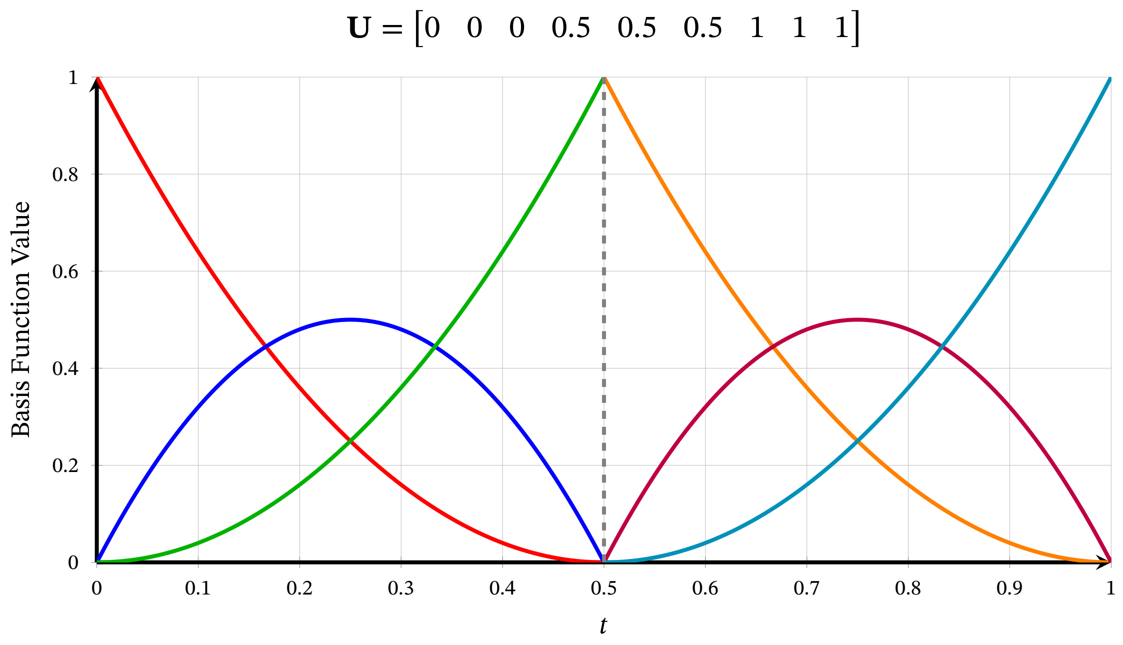


Figure 2.5: Basis functions of a B-spline equivalent to two chained Béziers

This is very close to what we want but it's still difficult to make sharp turns and precise movements without increasing the degree of the curve, making it more expensive to compute in real-time. It is possible to achieve even more control over the shape of our curve using NURBS.

2.3 NURBS

NURBS curves are a further generalization of B-splines. The acronym stands for “non-uniform rational basis spline.” [5]

$$\mathbf{C}(u) = \sum_{i=1}^{n} \frac{B_{i,p}(u)w_i}{\sum_{j=1}^{n} B_{j,p}(u)w_j} \mathbf{P}_i = \sum_{i=1}^{n} R_{i,p}\mathbf{P}_i \tag{2.5}$$

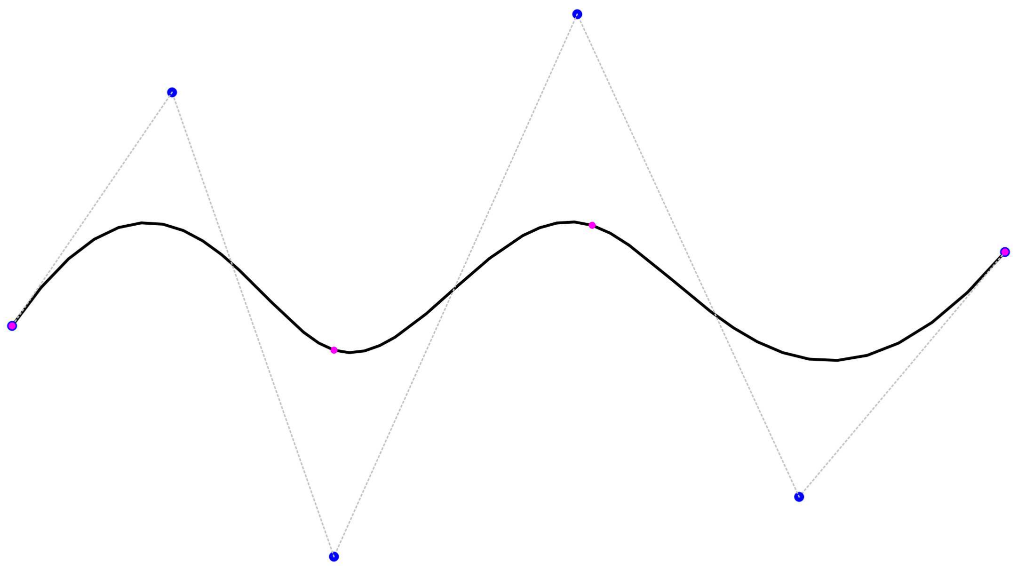


Figure 2.6: A third degree NURBS with six control points.

We describe NURBS as “rational” because they let us define a ratio, i.e., a weight, for each control point. This means that we can define the influence that a control point has on the overall curve. In terms of basis functions, this effectively scales a particular basis function in relation to its neighboring functions, creating a *rational basis function* $R_{i,p}$. Another way to interpret this is that all control points are defined in *homogenous coordinates*, using the three coordinates x, y and w, with the resulting spline being projected onto the plane $w = 1$.

We also describe NURBS as “non-uniform” because its basis functions do not need to be uniformly distributed. This is done using a knot vector which defines the distribution of a curve’s basis functions. B-splines are also non-uniform, as described earlier, though this is not reflected in the name.

Both of the aforementioned parametric curve models are subsets of NURBS curves.

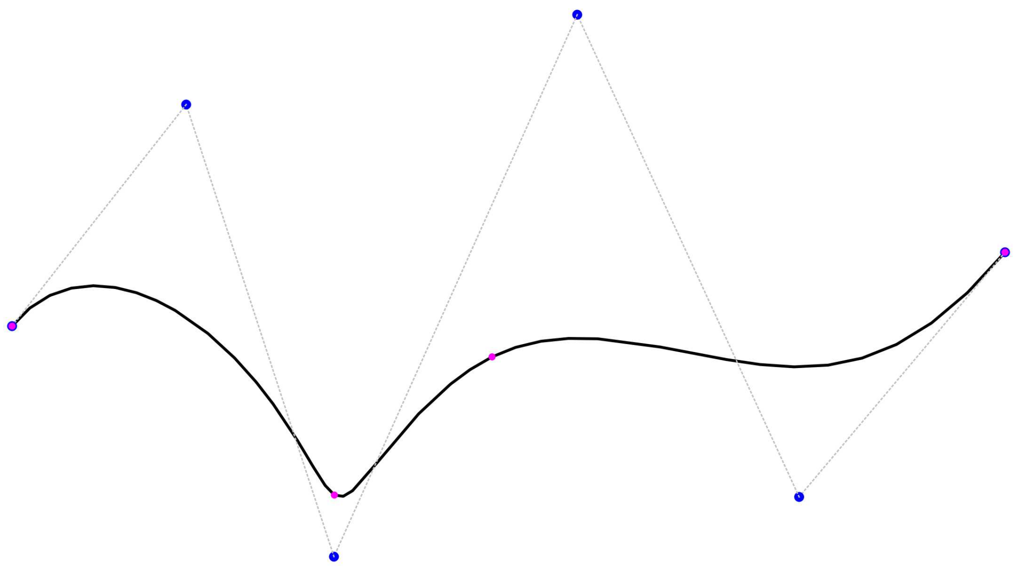


Figure 2.7: A third degree NURBS with six control points and one point weighted.

By limiting the parameters of a NURBS curve we can effectively create a Bézier, a B-spline, or a rational Bézier, among other types of splines.

Much like B-splines, a NURBS curve of degree p and N control points has a knot vector of size $N + p + 1$ and each point affects p knots. Each basis function belongs to one control point. Scaling a particular point's weight also scales its basis function in comparison to its neighboring functions, however it can still only reach the $p - 1$ knot spans before and after it, maintaining local control.

A NURBS curve of degree p always has geometric continuity of the $(p - 1)$-th degree along its length. Changing the knot vector may affect this property if there are multiple identical knots inside the curve. Each duplicate knot reduces the geometric continuity in that point by 1. Having p identical knots in the knot vector results in a sharp peak which only has G^0 continuity. All of this should be avoided, or prohibited to the end-user, if the goal is to maintain geometric continuity. Adjusting the weights cannot affect geometric continuity in any case.

2.3.1 Conic sections

A conic section in two-dimensional space is defined as the intersection of a three-dimensional cone's surface onto a plane. One property of NURBS that is not applicable to Bézier curves and B-splines is that they can represent conic sections, i.e., circles, ellipses, hy-

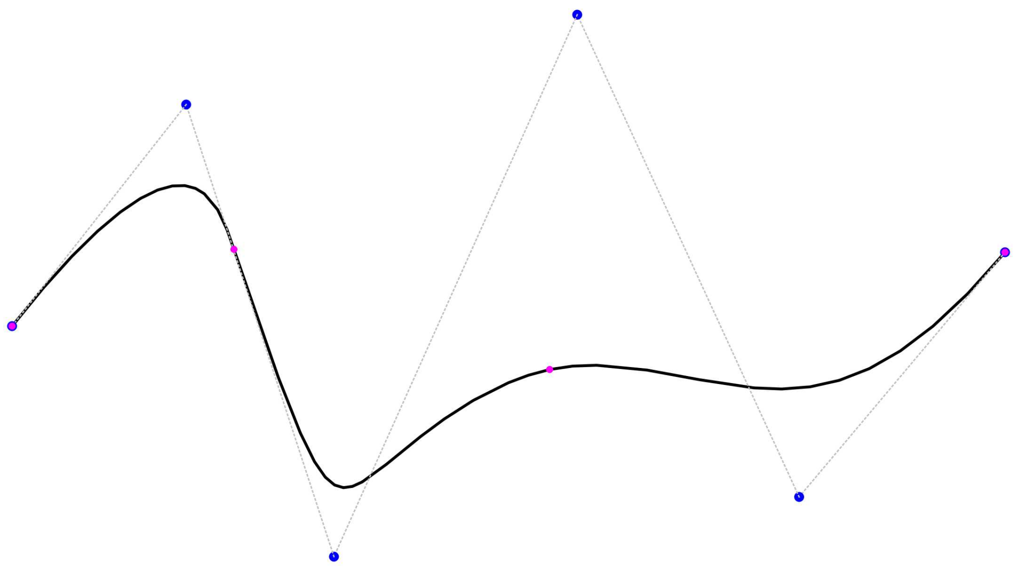


Figure 2.8: A third degree NURBS with six control points, one point weighted, and a modified knot span.

perbolas and parabolas. This is made possible by the ability to weigh specific points. [6] A second degree NURBS is sufficient to define a conic section.

2.3.2 Matrix representation of NURBS

The general formula for NURBS evaluation can be expressed in the form of a matrix making it more appropriate for programmatic computation. [7]

The basis matrix of a spline is defined recursively using Equation (2.6). This matrix is applicable to both NURBS and B-splines due to the fact that it depends only on the knot vector, which is a feature of both NURBS and B-splines, and does not depend on weights which are exclusive to NURBS. Equation (2.6) calculates the i-th basis matrix, with i being the index of a particular knot span. The elements on the main and first diagonal are calculated using Equation (2.7), with t_i being the i-th element of the knot vector. For a knot span of degree p, computing its basis matrix requires p iterations with constant-time operations, resulting in $O(p)$ time complexity.

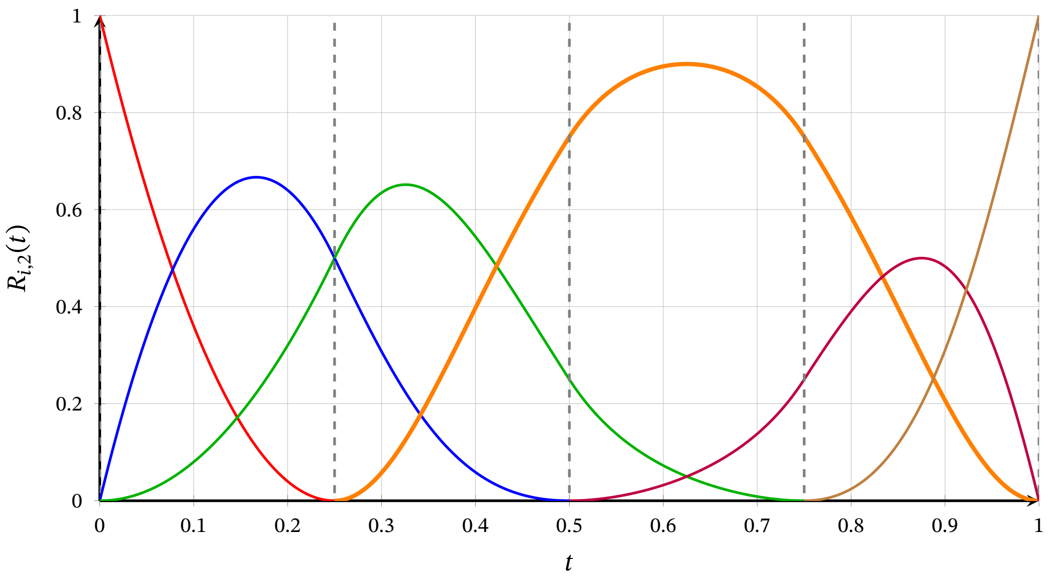


Figure 2.9: Example basis functions for a quadratic NURBS, with one point weighted (in orange)

$$\begin{aligned}
\mathbf{M}^k(i) &= \begin{bmatrix} \mathbf{M}^{k-1}(i) \\ \mathbf{0} \end{bmatrix} \begin{bmatrix} 1-d_{0,i-k+2} & d_{0,i-k+2} & & 0 \\ & 1-d_{0,i-k+3} & d_{0,i-k+3} & \\ & \ddots & \ddots & \\ 0 & & 1-d_{0,i} & d_{0,i} \end{bmatrix} \\
&+ \begin{bmatrix} \mathbf{0} \\ \mathbf{M}^{k-1}(i) \end{bmatrix} \begin{bmatrix} -d_{1,i-k+2} & d_{1,i-k+2} & & 0 \\ & -d_{1,i-k+3} & d_{1,i-k+3} & \\ & \ddots & \ddots & \\ 0 & & -d_{1,i} & d_{1,i} \end{bmatrix} \\
\mathbf{M}^1(i) &= \begin{bmatrix} 1 \end{bmatrix}
\end{aligned} \tag{2.6}$$

$$\begin{aligned}
d_{0,j} &= \frac{t_i - t_j}{t_{j+k-1} - t_j} \\
d_{1,j} &= \frac{t_{i+1} - t_i}{t_{j+k-1} - t_j}
\end{aligned} \tag{2.7}$$

A point on a knot span at a given parameter value u is evaluated using Equation (2.8). Equation (2.9) describes the power series of the parameter value u, while Equation (2.10) describes the weighted control points and the weights affecting the knot span. Given that the vector $\mathbf{U}^k$ has a size of $1 \times k$, the matrix $\mathbf{M}^k$ a size of $k \times k$ and $\mathbf{P}^k$ a size of $k \times n$, we can tell that the result is a point in n-dimensional space. The denominator is always a scalar.

$$\begin{aligned} \mathbf{c}_{i-k+1}(u) &= \frac{\mathbf{U}^k\mathbf{M}^k(i)\mathbf{P}^k(i)}{\mathbf{U}^k\mathbf{M}^k(i)\mathbf{W}^k(i)} \\ &= \frac{\mathbf{R}(u)}{S(u)} \end{aligned} \tag{2.8}$$

$$\mathbf{U}^k = \begin{bmatrix} 1 & u & u^2 & \dots & u^{k-1} \end{bmatrix} \tag{2.9}$$

$$\mathbf{P}^k(i) = \begin{bmatrix} w_{i-k+1}\mathbf{V}_{i-k+1} \\ w_{i-k+2}\mathbf{V}_{i-k+2} \\ \vdots \\ w_i\mathbf{V}_i \end{bmatrix}, \mathbf{W}^k(i) = \begin{bmatrix} w_{i-k+1} \\ w_{i-k+2} \\ \vdots \\ w_i \end{bmatrix} \tag{2.10}$$

With the basis function, control points and weights all being known constants, evaluating a point on a knot span only requires computing the vector $\mathbf{U}^k$ and performing the necessary matrix multiplication instead of the iterative sum given by the original definition of NURBS.

2.3.3 Derivative calculation

Knowing that the matrices $\mathbf{M}$, $\mathbf{P}$ and $\mathbf{W}$ are independent of $\mathbf{U}$, we can calculate the derivative of Equation (2.8) as given in equation (2.11).

$$\frac{d^n}{dt^n}\mathbf{c}_{i-k+j}(u) = \sum_{j=0}^{n} \binom{n}{j} \frac{d^j}{dt^j}\{\mathbf{R}(u)\} \cdot \frac{d^{n-j}}{dt^{n-j}}\{1/S(u)\} \tag{2.11}$$

We know from Equation 2.8 that $\mathbf{R}(u)$ is equal to $\mathbf{U^k M^k}(i)\mathbf{P^k}(i)$. Differentiating $\mathbf{R}(u)$ involves only differentiating the elements of the vector $\mathbf{U}^k$ by u, since the other elements are constants independent of u. Differentiating $1/S(u)$ requires the quotient rule.

2.3.4 Curvature and curvature derivative

Equation 2.12 defines the curvature of a parametric curve at a parameter t. This value represents the inverse of the radius of a circle that can be inscribed inside the curve; a straight line has a curvature of infinity. This formula is not specific to NURBS but can be applied to any twice-differentiable parametric curve, or thrice-differentiable for the derivative defined in Equation 2.13.

$$\kappa(t) = \frac{\mathbf{c}'(t) \times \mathbf{c}''(t)}{||\mathbf{c}'(t)||^3} \tag{2.12}$$

$$\frac{d}{dt}\kappa(t) = \frac{\mathbf{c}'(t) \times \mathbf{c}'''(t)}{||\mathbf{c}'(t)||^3} - 3\left(\mathbf{c}'(t) \cdot \mathbf{c}''(t)\frac{\mathbf{c}'(t) \times \mathbf{c}''(t)}{||\mathbf{c}'(t)||^5}\right) \tag{2.13}$$

2.3.5 Curve splitting

Splitting a NURBS curve at a given parameter t amounts to splitting whichever one of its knot spans contains the point t. Using matrix notation, splitting a knot span into two separate spans can be calculated in a single operation. Equation (2.14) defines a matrix that, when multiplied by a span's basis matrix, results in the basis matrix of a subsection of the span from 0 to t. Similarly, Equation (2.15) defines a matrix that produces a subsection from t to 1.0. [5] In most use cases the parameter t is equal to 0.5, meaning that these matrices can be precomputed.

$$\mathbf{L}_p(t) = \begin{bmatrix} 1 & & & & 0 \\ & t & & & \\ & & t^2 & & \\ & & & \ddots & \\ 0 & & & & t^p \end{bmatrix} \tag{2.14}$$

$$\mathbf{R}_p(t) = \begin{bmatrix} z_R[0,0] & z_R[0,1] & \cdots & z_R[0,p] \\ & z_R[1,1] & & \\ & & z_R[2,2] & \\ & & \ddots & \\ 0 & & & z_R[p,p] \end{bmatrix}, z_R[i,j] = \begin{cases} \binom{j}{i} t^{j-i}(1-t)^i & i \leq j \\ 0 & i > j \end{cases} \tag{2.15}$$

2.4 Comparison of NURBS, B-splines and Bézier curves

Most software using parametric curves primarily uses Bézier curves, however they are limited in their capabilities. Most commonly they are limited to a degree of three or lower; for example, the SVG standard is limited to only quadratic and cubic curves. [8] In some fields such as graphic design this is not considered a problem, as their simplicity, portability and computation speed are more valuable than highly precise shapes. Other fields, such as engineering or robotics, require highly optimized and precise paths, for which Bézier curves are often not sufficient. Software meant for engineering and CAGD, such as AutoCAD and Rhino, use NURBS for this reason.

Out of the three curve models presented, NURBS are the most flexible and allow for the most control, though they might be computationally expensive. However, having modifiable weights gives us a level of flexibility above B-splines without increasing the degree, which makes them potentially more efficient than B-splines if used wisely.

Because all B-splines can be expressed as a NURBS curve with all weights set to 1.0, and all Bézier curves can be expressed as a B-spline with knot multiplicity, it can be concluded that NURBS are a superset of both B-splines and Bézier curves. By implementing NURBS it is possible to cover all use cases of the other two parametric curve types and conic sections, as well as providing additional functionality.

3 NURBS curve implementation

For the purpose of path planning for autonomous mobile robots, a C++ library was created for the implementation and manipulation of NURBS curves. The library utilizes the Eigen library for its linear algebra functions.

The library implements all necessary functions for creating and using NURBS curves. Elementary transformations, such as translation and rotation, are not implemented. It is important to note one of the properties of NURBS curves which is that all affine transformations applied to the set of control points, as well as perspective projections, are equivalent to transformations on the curve itself, meaning that the end-user may implement these functions themselves if necessary [5]. The library only provides the minimum required functions to cover all operations exclusive to NURBS curves.

3.1 The Eigen library

Eigen is an open-source C++ template library for linear algebra [9]. It is commonly used in the fields of computer graphics and robotics. The library provides class and function templates that allow for compile-time optimization, as well as *SIMD* ("single instruction, multiple data") vectorization which drastically speeds up operations on vectors and matrices. Using *Eigen's* built-in vector and matrix classes allows for interoperability with other libraries and end-user programs that use *Eigen*, which is often the case in robotics.

Most of Eigen's functionality revolves around its `DenseBase` class which serves as a parent class to all dense matrices and vectors. Two subclasses which will be relevant for the implementation of NURBS are the `Matrix` and `Vector` classes, which are further subclassed to describe matrices and vectors of specific dimensions. Some examples of these subclasses are `Matrix2Xd`, which describes a matrix of width 2 and an arbitrary height, and `RowVectorXd` which describes a vector of height one and an arbitrary width. Specifying these types at compile-time instead of using the generic parent classes allows the compiler to optimize certain operations for constant-size matrices instead of ones with a size only known at runtime.

Normally, matrix multiplication is an expensive operation, with $n \times n$ matrix multiplication having $O(n^3)$ time complexity, however Eigen's vectorization and SIMD capabilities allow this to be much faster than implementing it manually.

3.2 Class structure

The main data structure that is available to the user is the `Curve` class. A curve is divided into knot spans, or segments of the curve between two adjacent knots, which are represented by the `Span` utility class. A `Curve` object manages its own `Span` objects which are not visible outside of the class, nor is the Span class itself visible to the end user.

3.2.1 Curve

The `Curve` class represents a complete NURBS curve, i.e. sequence of continuous knot spans. It stores some of the basic data of a particular curve, including its control points, knot vector, and order. Internally it is represented by an array of `Span` objects containing the data required to evaluate many NURBS operations, though they are not accessible to the end-user.

Methods that operate on a single span, rather than the entire curve, involve evaluating which spans are being affected, then propagating the function call to the `Span` objects themselves. The `getKnotSpanIndex(t)` method returns the index of the knot span containing the value t. The result is always a single span and never a set of spans. If the value t is itself a knot with a multiplicity greater than one, the function returns the first span after the multiple knot, and never the zero-length span containing the knot t.

3.2.2 Span

The `Span` class defines a segment of a curve between two knots. It shares its order with its parent curve and the number of control points is limited by its order. A single span does not have a knot vector. The shape of a span is defined by its basis function, which is in turn defined by the parent curve's knot vector. This basis function is defined in the code as a $(p + 1)$ by $(p + 1)$ matrix, where p represents the span's order, and is stored in each span.

For the sake of performance and memory efficiency, the `Span` class does not store its control points, but instead caches the dot product of the weights with the basis function, as well as the weighted control points with the basis function, as required by Equation (2.8) for the evaluation of a point on the curve.

Whenever the curve's knot vector is modified, the affected spans use the `update` method to recompute their basis functions as defined in Equation (2.6). Afterwards they recompute their cached control point and weight matrices using the `updateControlPoints` method.

3.3 Key method implementations

The `Curve` class has several constructors, however the main intended method of instantiation is using a sequence of control points. The curve's order defaults to three but can be specified. The knot vector is optional, and if it is not provided a uniformly spaced knot vector is generated. If weights are not specified they are all set to 1.0. As per the definition of NURBS, the number of control points for a curve of degree p must be equal or greater than $p+1$. Given a sequence of N control points, an order p and a knot vector, the constructor generates $N - p$ `Span` objects.

The `Span` constructor takes a subset of control points, a subset of a knot vector and the span's order. The first step is to generate the span's basis function based on the given knot vector segment and order. The basis function is computed according to Equation (2.6) and stored in the span. Using the basis function and the weighted control point segment the span computes the product of the basis function with the control point matrix, and the basis function with the weight vector, named `cached_vbf_` and `cached_wbf_` respectively. After calculating and caching these data members, the span is returned and stored in the curve object.

The `valueAt` method directly implements Equation (2.8) using matrix notation. The helper function `_powSeries(t, p)` computes the powers of t up to p in accordance with Equation (2.9).

```
Point Span::valueAt(double u) const { Eigen::RowVectorXd pw =
 ↪  _powSeries(u,
p_); return (pw * cached_vbf_) / pw.dot(cached_wbf_); }
```

The `length` method approximates the total length of a curve. Exact calculation of the length of any polynomial parametric curve of the third degree or higher in closed form is not possible. The method instead uses Chebyshev approximation to calculate the length of the span with a negligible error `epsilon`, equal to the margin of error in C++ doubles.

The `boundingBox` method creates an axis-aligned bounding box (AABB) around the curve. It does this using the `extrema` method to find all local maxima and minima on the x and y axis, and finding the ones with the highest and lowest values.

The `extrema` method and the `roots` method both use Eigen's polynomial solver module to find the roots of a polynomial, in this case the derivative and the span itself respectively.

3.3.1 Piecewise linear representation

The `polyline` method creates a sequence of points suitable for drawing or other purposes when only an approximation of the curve is needed. One possible way, and the simplest way to implement this would be to iterate over a fixed number of uniformly spaced t values. However, this can cause problems as it does not take into account the scale of the curve and its curvature; a curve could have the same number of points as another curve ten times its length, and a curve that is equal to a straight line could have the same number of points as one with many sharp twists.

The `polyline` method takes a `flatness` argument which defines the tolerance for the error between the linear approximation and the actual curve. The implementation uses iterative subdivision to approximate the curve. For each knot span of the curve, the method checks $p-1$ equally spaced internal points between the beginning and end of the span, and finds the maximum distance from the chord created by the segment's beginning and end point. The points on the curve are projected onto the line segment.

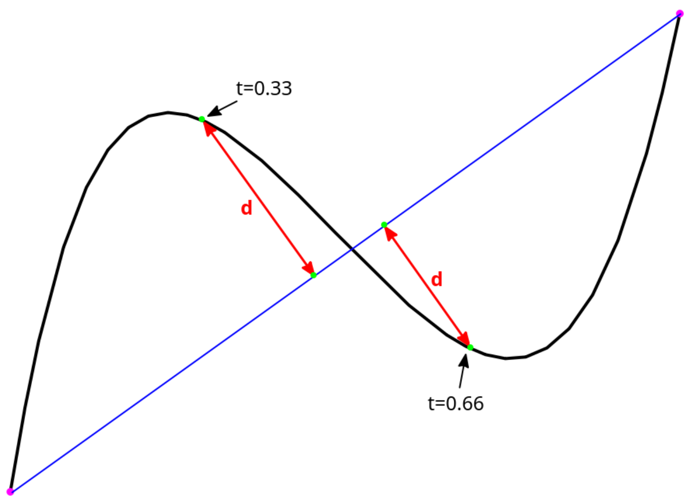


Figure 3.1: Illustration of one step of the polyline algorithm. $p = 3$

If the distance from the line segment to the actual curve is within the threshold defined as `flatness`, the segment is added to the final line. Otherwise, it is subdivided further at the point with maximum deviation. This is illustrated in Figure 3.1, where the span would be subdivided at $t = 0.33$.

3.3.2 Piecewise Bézier representation

As discussed in Chapter 2.4, it is possible to represent Bézier curves as NURBS. Similarly, a NURBS curve can be converted into a series of Bézier curves. Converting a NURBS with modified weights results in rational Bézier curves, whereas an unweighted NURBS (i.e. a B-spline) will produce regular Bézier curves.

3.3.3 Derivative evaluation

The `derivativeAt` method computes the n-th derivative of the curve on a given point t. The order of the derivative is not limited, however higher values for n can result in slow performance. The helper method `_powSeriesDerivative` calculates the derivative of the power series of t, which amounts to differentiating each of the elements individually.

$$\begin{aligned}
\mathbf{U}_k(t) &= \left[1, t, t^2, t^3, \dots, t^k\right] \\
\frac{d}{dt}\mathbf{U}_k(t) &= \left[0, 1, 2t, 3t^2, \dots, kt^{k-1}\right] \\
\frac{d^2}{dt^2}\mathbf{U}_k(t) &= \left[0, 0, 2, 6t, \dots, k(k-1)t^{k-2}\right] \\
&\dots
\end{aligned}$$

The method then iteratively calculates the derivatives from the first to the n-th according to Equation (2.11), using the previous derivative in each consecutive calculation.

The `extrema` method calculates the positions of all extrema of the curve on the x and y axis. One use case of this is calculating the bounding box of the curve or span, as done in the `boundingBox` method in the `Span` class. The `boundingBox` method in the `Curve` class, on the other hand, gets the bounding boxes of all its spans and finds their minimum and maximum.

3.3.4 Curve and span splitting

the `_splittingCoeffs` helper function creates a pair of matrices which, when multiplied with a span's basis function, splits the span into two new spans at a given point t. The matrices are created according to Equations (2.14) and (2.15).

This function is used in methods such as `polyline` and `intersections`, but also in the `Curve` class for literal curve splitting.

3.3.5 Intersections between curves

Finding all intersections between two spans is known to be impossible. Instead, an iterative method is used to approximate the intersections with an error of `epsilon`.

The `intersections` method uses a similar algorithm to the `polyline` method to determine if a span is approximately a straight line. Instead of a queue of subcurves, it keeps a queue of *pairs* of subcurves to be checked against each other. The queue is populated with both whole spans in the beginning. For each pair in the queue, it first checks their bounding boxes to see if they possibly don't overlap at all. If they do, the method then checks if the diagonals of the bounding boxes are of `epsilon` length, or if the subcurves are an approximately straight line. If not, both subcurves are split at $t = 0.5$ using previously cached splitting matrices, and all combinations of the four curves are added into the queue. As the subcurves are split, they eventually reach the exit condition, in which case their intersection is calculated as the intersection of straight line approximations of the two subdivided parts.

4 Robot path visualization

4.1 Qt Framework

Qt is a C++ framework for the development of native desktop applications. For the purpose of debugging the NURBS library and demonstrating its functionalities, a program was created using Qt and C++. Visualizations of curves in earlier chapters were also created using this application.

The central element in a Qt program is the `QMainWindow` class, which the user extends and modifies to their own needs. This window can be configured with *widgets* and custom behaviour. In this case, a `MainWindow` class was created with methods that connect the user interface to methods from the NURBS library using *signals* and *slots*, two mechanisms which the Qt framework heavily relies on. Listing 1 shows part of the initialization of the `MainWindow` class, in which two default curves are defined and constructed. The rest of the method connects the signals and slots of various UI elements.

Signals and slots are Qt's way of implementing the listener design pattern. A class defines signals that it may emit when a specific event happens; for example, a `QButton` will have a public signal `pressed` that emits whenever the button is pressed. A slot can be any member function and can be connected to a signal. The slot function gets called when the signal emits. The signal may or may not have arguments that it passes to whichever slot is connected to it. This could be a numerical value in a number input, or an index in a set of radio buttons. A signal can be connected to any slot with matching arguments. A signal can be connected to any number of slots, and a slot can be triggered by any number of signals. This is the main way to achieve interaction between elements in Qt.

```
MainWindow::MainWindow(QWidget* parent) : QMainWindow(parent),
 ↪  ui(new Ui::MainWindow), scene(new CustomScene)
    {
    ui->setupUi(this);
    ui->graphicsView->setScene(scene);
    new QGraphicsViewZoom(ui->graphicsView);

    Eigen::MatrixX2d cp1, cp2;
    cp1.resize(4, 2);
    cp2.resize(5, 2);

    cp1 << 84, 162, 246, 30, 48, 236, 180, 110;
    cp2 << 180, 110, 175, 160, 60, 48, 164, 165, 124, 134;

    NURBS::Curve curve1(cp1 * 5);
    NURBS::Curve curve2(cp2 * 5);

    addCurveToScene(curve1);
    addCurveToScene(curve2);
    ...
    }
```

Listing 1: `MainWindow` constructor method and initialization.

4.1.1 Graphics view

The actual visualization of NURBS paths was done using the `QGraphicsView` module. This module implements various drawing functions including those for drawing line segments and points.

A graphics view widget must be initialized with a `QGraphicsScene` which manages objects to be drawn as well as handling user input. Each object in a scene must inherit from `QGraphicsObject`. A `CustomScene` class was created to handle drawing multiple curves on the screen.

The class `qCurve`, which inherits from both `QGraphicsObject` and `Curve`, was created as a wrapper to provide Qt drawing functionality to a NURBS curve. It overrides some methods of `QGraphicsObject`, such as `paint`, `type` and `boundingRect`, to specify drawing and interaction operations unique to NURBS curves while staying compatible with the Qt API.

Overloading the `paint` function of `QGraphicsObject` allows us to define how we

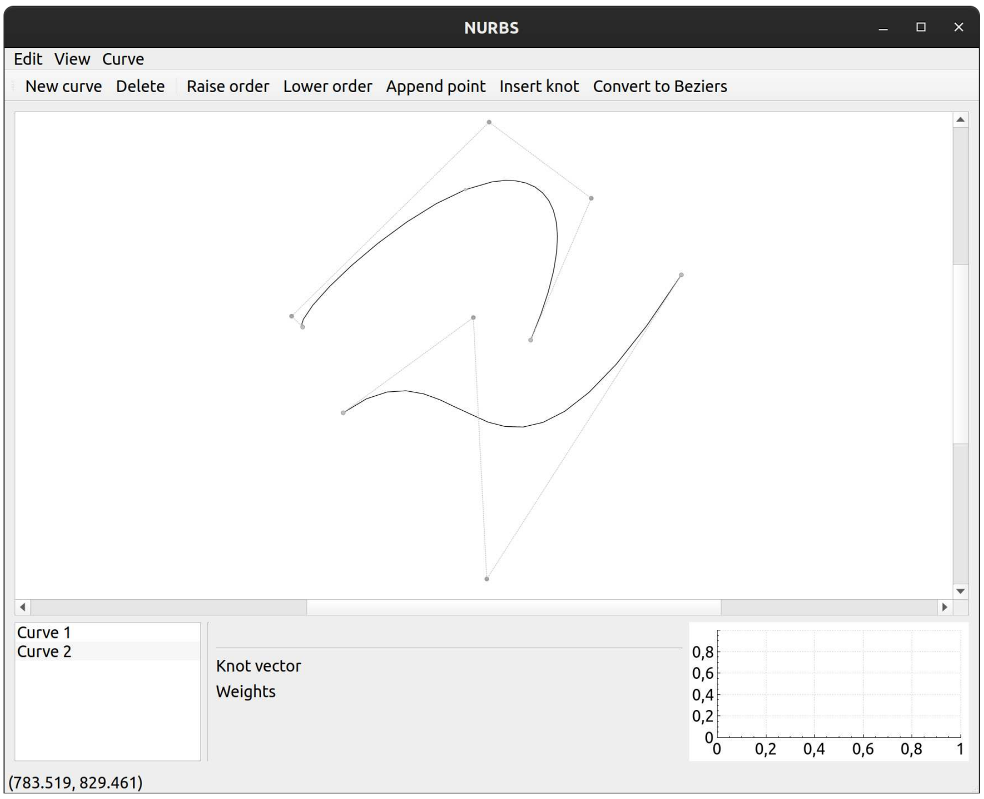


Figure 4.1: Interface of the NURBS example application.

want our object to be drawn in a `QGraphicsView` widget. In the case of NURBS curves, this means that it is possible to draw not just the curve itself, but other data associated with it, such as its control points, knots, or its bounding box. The `polyline` method defined in the NURBS library creates a sequence of points for the express purpose of quick and simple visualization. On every scene update each curve's polyline is recomputed and drawn using the `drawLine` method of the `QPainter` class. A scene is only updated if something inside it was modified, such as the control points on a curve. If another curve in a scene was not modified and has its polyline cached, it will not recompute its polyline.

4.1.2 User interaction

Several methods have been implemented for the user to interact with a NURBS curve in the custom scene, both for debugging and to make sure the curves are intuitive to work with and manipulate.

The `qCurve` class draws some additional information along with the curve itself. It draws the control points which can be dragged to modify the curve. Along the length of the curve it draws the edges of the knot spans. It can draw the bounding box around the curve, and display the curvature as a series of perpendicular lines. Any of these functions can be enabled and disabled using keyboard shortcuts.

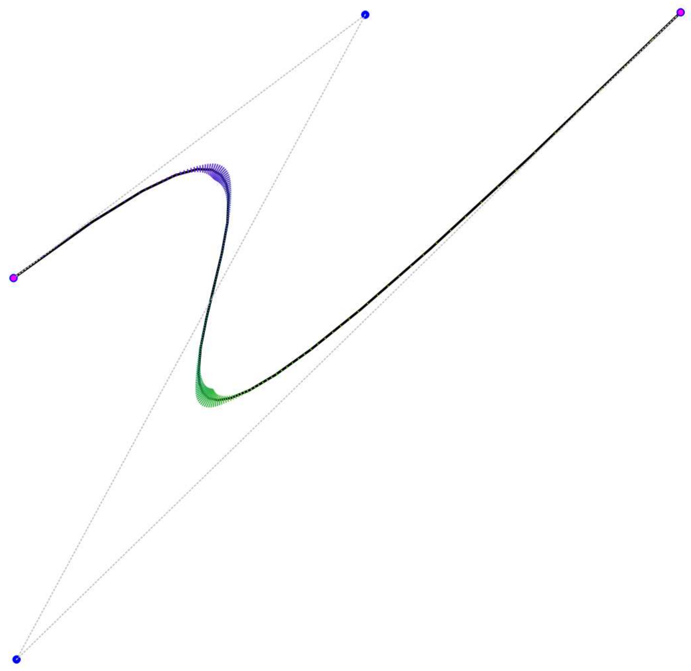


Figure 4.2: A NURBS curve with its curvature displayed. The color represents the value of t.

The `CustomScene` class overrides several methods of `QGraphicsScene` in order to implement the functionality required to interact with the curves. This is done through the `mousePressEvent`, `mouseMoveEvent` and `mouseReleaseEvent` methods.

The `mousePressEvent` method handles any mouse presses inside the scene. When the left mouse button is pressed, it checks for collision with each curve and control point in the scene. If a curve was clicked, it becomes selected. Control points can be dragged and moved around the scene. Holding the right mouse button projects a line onto the selected curve and draws its tangent.

Underneath the graphics view are a few other widgets for editing the currently selected curve, as can be seen in Figure 4.5. The `qVectorField` widget was created to edit fixed-length arrays of data, namely for knot vectors and weights for the active curve. On the left side is a `QListWidget` with a custom item type, `qCurveListItem`, which makes a curve active upon being selected in the list.

The open-source Qt widget *QCustomPlot* is used to visualize the basis functions of the

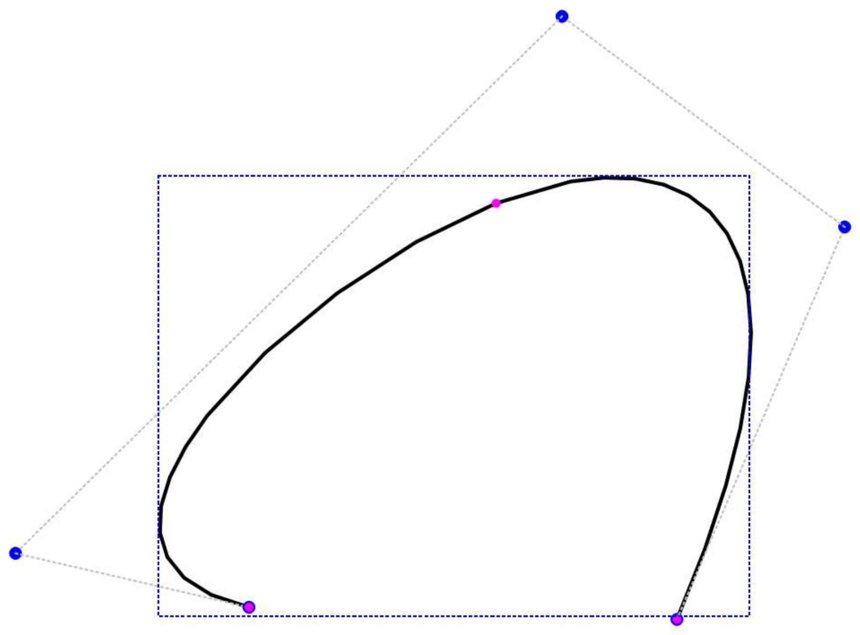


Figure 4.3: A NURBS curve with visible control points, knots and bounding box.

selected curve[1]. The plotting is done by taking each span's basis matrix and sampling it over an interval of $[0, 1]$, adjusted to the knot span's actual beginning and end in the knot vector.

4.2 RViz

For testing in a simulation environment and designing paths for mobile robots a plugin was created for the RViz tool often used for visualization of robots.

4.2.1 Robot Operating System

The *Robot Operating System*, or ROS for short, provides a standardized interface between user programs and robot architecture, and is commonly used in the field of robotics.

ROS is mainly built upon the concept of *publishers* and *subscribers* sending and receiving data among each other. A publisher sends *messages*, which can contain various different types of data, to a *topic*. These will often contain user commands, sensor data or other real-time information that a robot might use. A subscriber subscribes to a topic

[1]Available at `https://www.qcustomplot.com/`

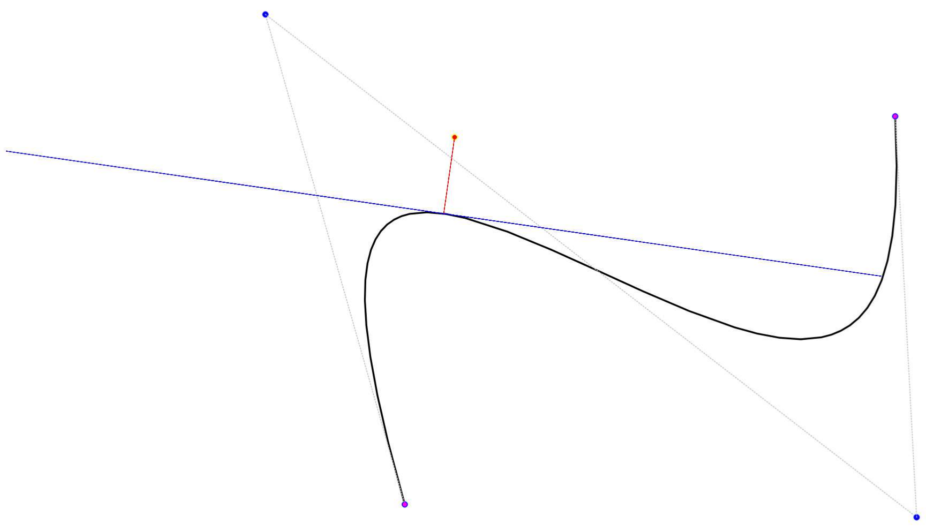
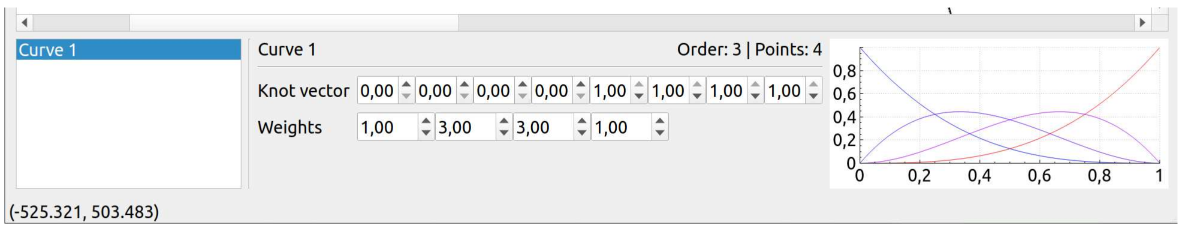


Figure 4.4: A NURBS curve with a point (in red) being projected onto a curve, and its tangent in the resulting point.



Curve 1

Order: 3 | Points: 4

Knot vector 0,00 0,00 0,00 0,00 1,00 1,00 1,00 1,00

Weights 1,00 3,00 3,00 1,00

(-525.321, 503.483)

Figure 4.5: Closeup of the controls for a NURBS curve.

and receives all messages sent to that topic. Subscribers may receive commands for a robot, but also collect data from a robot's sensors to show to the user.

RViz works by subscribing to topics in a system and visualizing their data. Because a topic can have many subscribers at a time, and messages are broadcast to all of its subscribers at once, it can be inserted into an existing system to visualize and debug without disrupting any other subscribers.

4.2.2 PathStamped message

The message type used to describe a mobile robot's path is defined as `PathStamped`. The message contains information relevant to the robot and the user such as the path's start and end points and their types. The section that unambiguously describes the NURBS curve contains its degree, its weighted control points and its knot vector. This information is used by the plugin to construct a NURBS object and use its functions. This means that the sender does not need to use the NURBS library; the message may be predefined

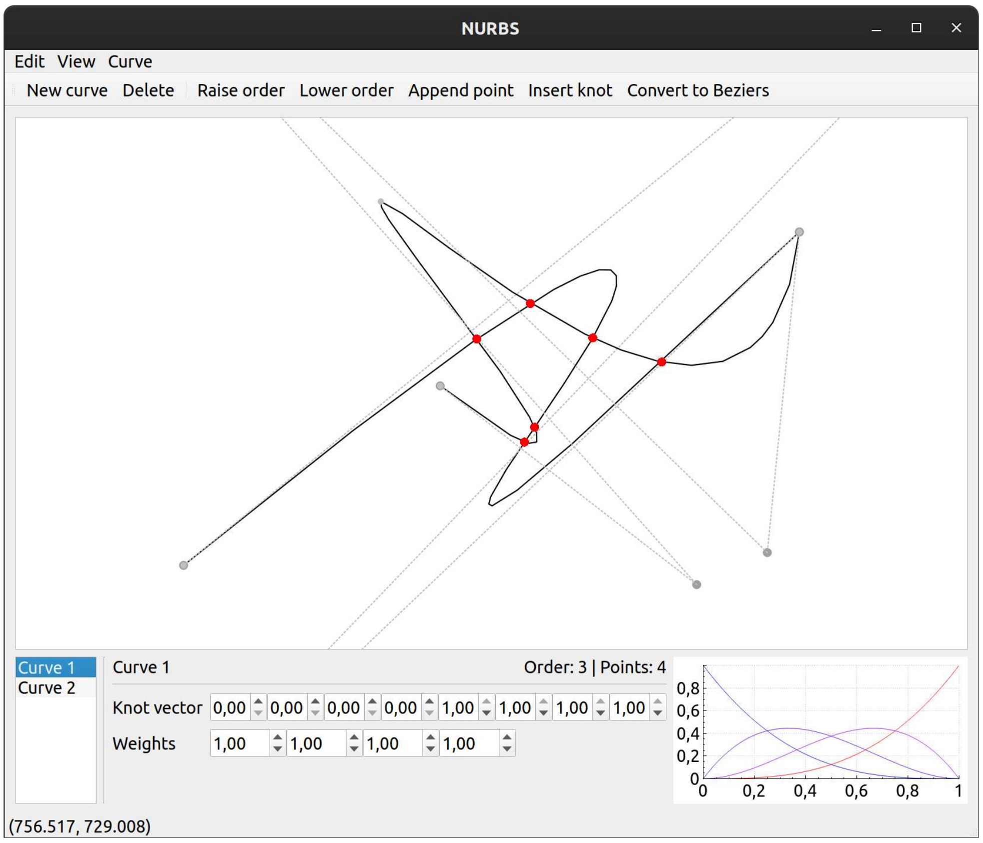

NURBS
Edit View Curve
New curve Delete Raise order Lower order Append point Insert knot Convert to Beziers
Curve 1
Curve 2
Curve 1
Order: 3 | Points: 4
Knot vector 0,00 0,00 0,00 0,00 1,00 1,00 1,00 1,00
Weights 1,00 1,00 1,00 1,00
0,8
0,6
0,4
0,2
0
0
0,2
0,4
0,6
0,8
1
(756.517, 729.008)

Figure 4.6: Two NURBS curves with their intersections highlighted in red.

in a file or created by a different NURBS implementation, as long as it follows the message structure.

The message contains some data that is relevant to the robot and user, but not related to the NURBS curve itself. This includes the path's unique ID and its start and end nodes. Though it is important to mention, it will not be used in the path's visualization.

While the immediate purpose for this message is to visualize it with RViz, its real, long-term purpose is to be used for communication between a mobile robot and its controller, which is how it was designed.

4.2.3 RViz plugin development

RViz listens for ROS messages of a given type and, upon receiving data, draws the data appropriately. In this case, the message is a Path with timestamps, named `PathStamped`, which is a custom message type and is not included in ROS. RViz supports custom plugins written in C++, also called *displays*, that add support for drawing custom message

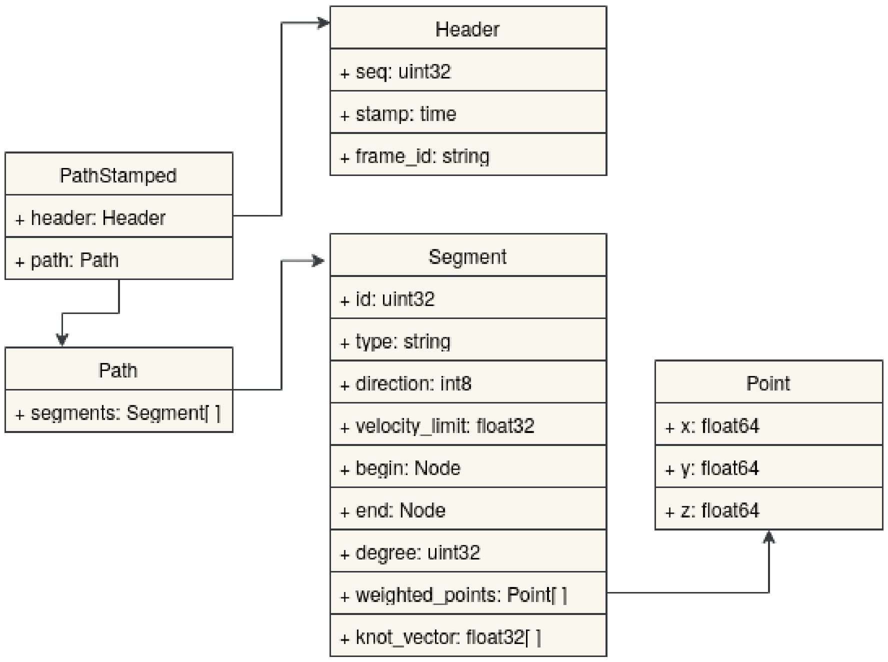


Figure 4.7: UML diagram of the PathStamped message type.

types.

RViz itself is created in Qt which means it reuses many of Qt's concepts when it comes to plugin development. One important feature is the ability to add custom editable properties in the GUI. When making a plugin, each property provides a *signal* that is emitted when the property is edited, which can be connected to *slots* that execute when the signal emits.

While it uses Qt for its user interface, RViz also uses the *Ogre* (Object-oriented Graphics Rendering Engine) library to render its 3D scenes and manage objects in a scene. An instance of this plugin creates a single object in the scene. The scene may contain other displays or meshes as well as the NURBS path drawn by the plugin. A scene may have multiple instances of the plugin, i.e., listeners for multiple ROS topics.

The `processMessage` method parses received messages into usable NURBS curves. The method works by reading all the data fields of a given `PathStamped` message and uses them to construct a NURBS curve using the library from Chapter 3. The plugin stores the last received curve to draw and perform other operations on.

The `updateVisuals` method creates the line to be drawn in the scene. Two line drawing styles are implemented: `BillboardLine` and `ManualObject`. A `BillboardLine` path creates a series of *billboards*, or 3D planes always facing the camera, to draw its lines. This allows the user to set the width of the line. The line has two properties: its color and its line width. A `ManualObject` path is defined only by a set of points and a color, and it is rendered as an OpenGL line strip, meaning that it has a fixed width of 1 pixel on all displays.

5 Testing and results

5.1 NURBS library

To test if the library is truly suited for navigation of mobile robots, a series of tests was made on a fourth degree curve with a uniform knot vector, as defined in Table 5.1. A short C++ program was created to sample the curve at a series of uniformly spaced parameter values, and the resulting data was plotted using Python and the `matplotlib` library. If the methods for evaluating and differentiating NURBS curves are implemented correctly, the following should be true: the first three derivatives are continuous, indicating G3 continuity, and the curvature and curvature derivative should be continuous.

x	y	w
0	0	1.0
10	10	1.0
0	20	1.0
10	30	1.0
0	40	1.0
10	50	1.0

Table 5.1: Control points for test curve

The curve's first three derivatives were uniformly sampled across a hundred points from 0.0 to 1.0 using the `derivativeAt` method of the `Curve` class. The partial derivatives on the x and y axes can be seen on Figures 5.1, 5.2 and 5.3. The test curve is split into two knot spans at $t = 0.5$, which is shown on the plots as a dashed vertical line.

It is obvious from Figure 5.3 that, while the third derivative is indeed continuous, it has a sharp peak in the middle, meaning that the fourth derivative will not be continuous. This aligns with the fact that a p-th degree curve will have continuity of order $p-1$. Each knot span's derivative is a polynomial which is one degree lower than the previous.

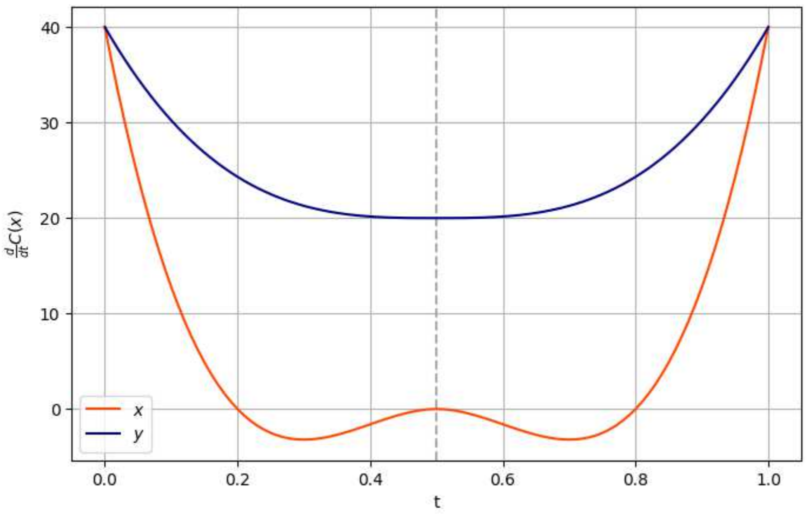


Figure 5.1: The first derivative, with x in red and y in blue.

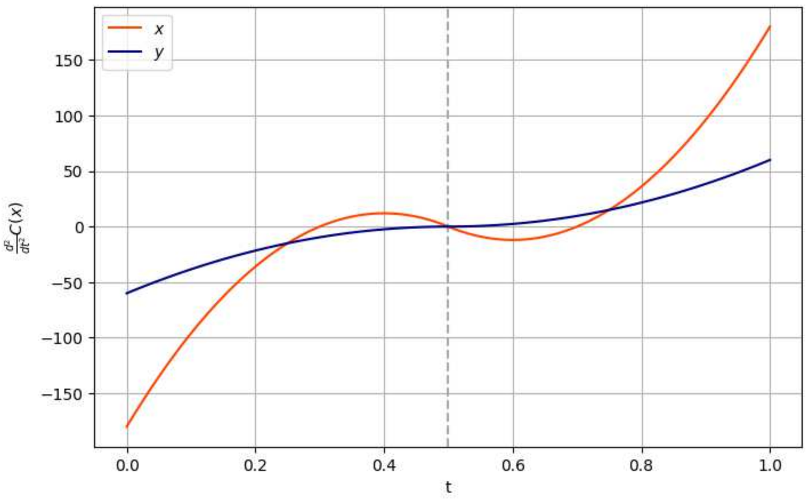


Figure 5.2: The second derivative, with x in red and y in blue.

The curvature and its derivative were uniformly sampled along the curve along with the derivatives, using the `curvatureAt` and `curvatureDerivativeAt` methods. The result can be seen in Figure 5.4.

The `polyline` method was tested along with the derivatives and curvature. For reference, the `valueAt` method was uniformly sampled along the same points as the previous tests. Figure 5.5 shows a side-by-side comparison of the two methods and their vertices, with the x and y axes swapped for clarity. Both curves appear visually identical but have different point densities. To illustrate the efficiency of both methods, the absolute value of the curvature at the given points is displayed using a color ramp. The naive method with uniformly sampled points, shown on the bottom, has more points overall.

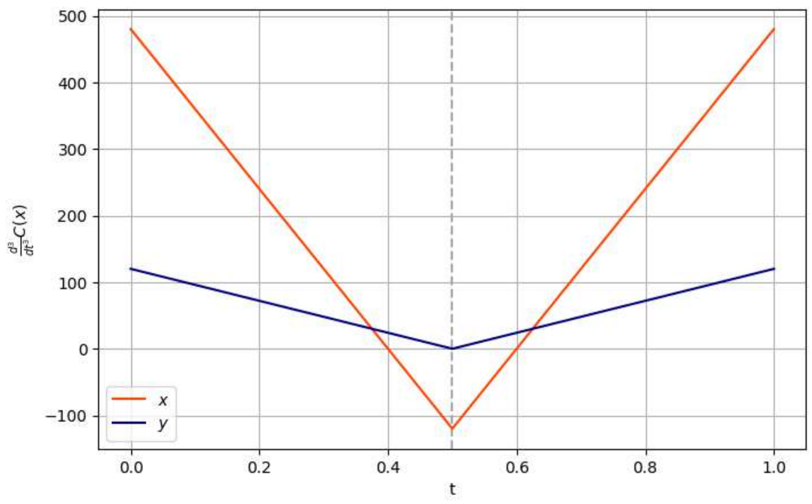


Figure 5.3: The third derivative, with x in red and y in blue.

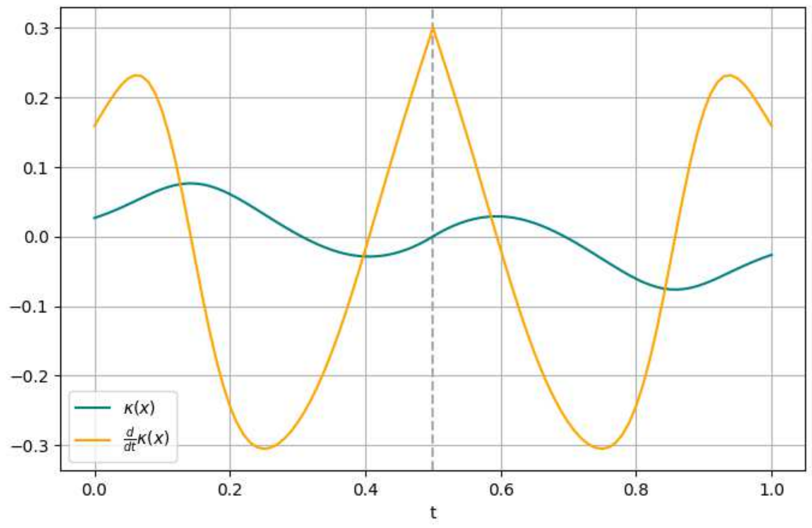


Figure 5.4: The curvature κ in teal and its derivative in orange.

The middle portion has equally dense points on areas with high curvature and low curvature, while the ends of the curve are more sparse due to the fact that we are sampling by parameter values, not by length. The `polyline` method, on the other hand, has fewer points overall. Despite the method not taking the curvature formula into account, there is a clear correlation between point density and curvature. The ends of the curve are in fact more dense than the uniformly sampled method, while the middle has relatively few points. Uniform sampling in larger intervals may lead to fewer points along straight sections, but it would also result in sparse points along high-curvature segments.

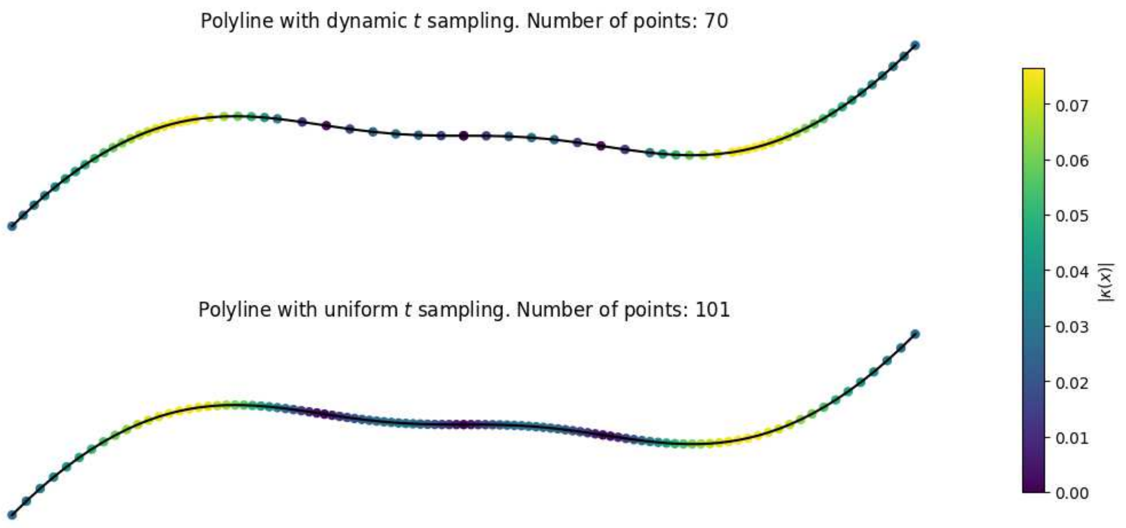


Figure 5.5: Comparison of two polyline algorithms. (Top) `polyline` method. (Bottom) Uniform sampling of t.

5.2 RViz plugin

Multiple configurations of NURBS curves were tested in the RViz plugin to test its accuracy. The configurations can be seen in Listing 2. For the sake of clarity only the data relevant to NURBS curves is listed. The message uses the z coordinate to define weights. The reasoning behind this design choice was that the control points are defined in homogenous space and projected onto the plane $w = 1$.

In order to run ROS and RViz locally, they need to be installed on an *Ubuntu* system. The setup for testing with RViz requires three programs. In one terminal the `roscore` command must be executed and left running, which runs the ROS server that connects all nodes. The second terminal needs to run the `rviz` command to run RViz. The third terminal is used to send messages via the `rostopic` command.

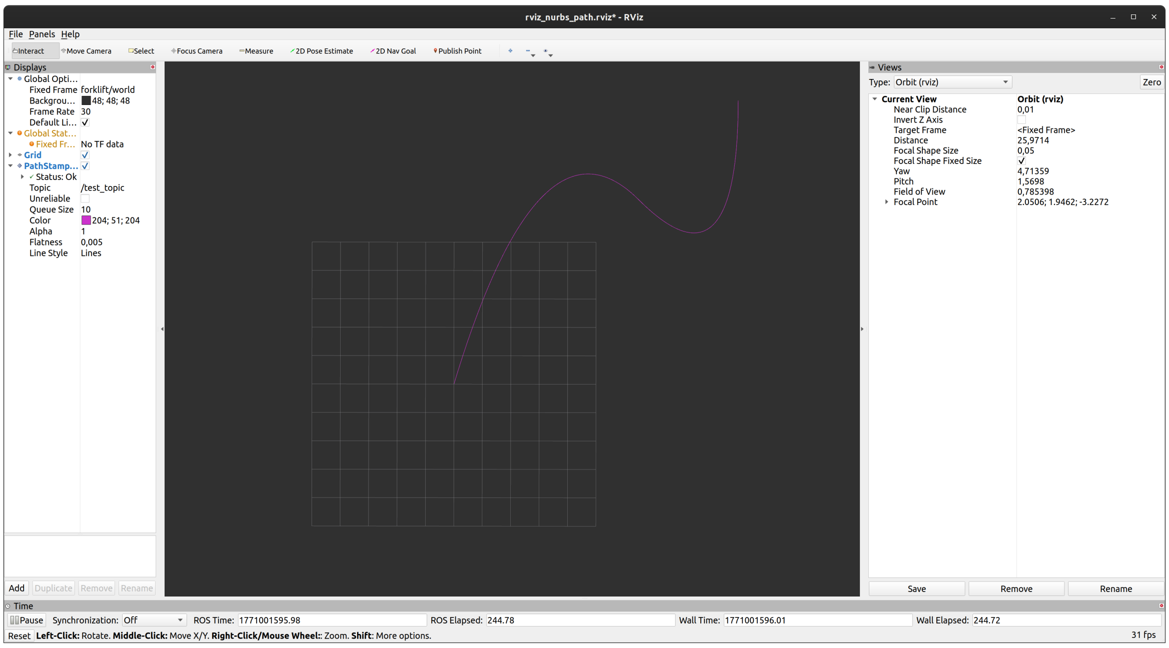
Having built the custom message types and the plugin, RViz must be set up to track messages of type `PathStamped` on a topic with an arbitrary name. Then, by running `rostopic pub [/topic_name] PathStamped [.yaml data]` the message gets sent to the topic and read by its subscriber, RViz. Once it receives the message it will be displayed as a pink line. The viewport can be moved around and the line properties can be edited.

```yaml
degree: 2
weighted_points:
-
    x: 0.0
    y: 0.0
    z: 1.0
-
    x: 3.0
    y: 10.0
    z: 1.0
-
    x: 10.0
    y: 3.0
    z: 1.0
-
    x: 10.0
    y: 10.0
    z: 1.0
knot_vector:
    - 0.0
    - 0.0
    - 0.0
    - 0.5
    - 1.0
    - 1.0
    - 1.0
```

```yaml
degree: 2
weighted_points:
-
    x: 0.0
    y: 0.0
    z: 1.0
-
    x: 6.0
    y: 20.0
    z: 2.0
-
    x: 20.0
    y: 6.0
    z: 2.0
-
    x: 10.0
    y: 10.0
    z: 1.0
knot_vector:
    - 0.0
    - 0.0
    - 0.0
    - 0.5
    - 1.0
    - 1.0
    - 1.0
```

```yaml
degree: 2
weighted_points:
-
    x: 0.0
    y: 0.0
    z: 1.0
-
    x: 3.0
    y: 10.0
    z: 1.0
-
    x: 30.0
    y: 9.0
    z: 3.0
-
    x: 10.0
    y: 10.0
    z: 1.0
knot_vector:
    - 0.0
    - 0.0
    - 0.0
    - 0.1
    - 1.0
    - 1.0
    - 1.0
```

Listing 2: NURBS test configurations in YAML format: (left) no changes, (middle) with weights, (right) with weights and knot vector

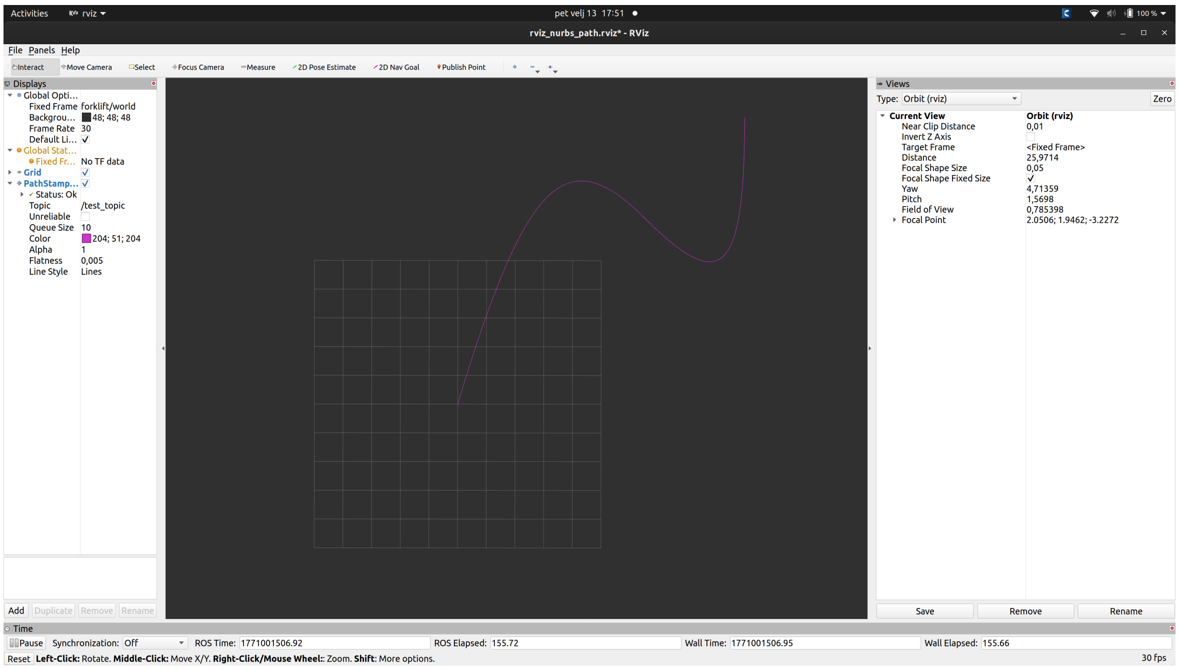


```
Global Opti…
    Fixed Frame      forklift/world
    Backgrou…        ■ 48; 48; 48
    Frame Rate       30
    Default Li…      ✓
Global Stat…
    Fixed Fr…        No TF data
Grid                 ✓
PathStamp…           ✓
    Status: Ok
    Topic            /test_topic
    Unreliable       ☐
    Queue Size       10
    Color            ■ 204; 51; 204
    Alpha            1
    Flatness         0,005
    Line Style       Lines
```

```
Type: Orbit (rviz)
Current View                 Orbit (rviz)
    Near Clip Distance       0,01
    Invert Z Axis            ☐
    Target Frame             <Fixed Frame>
    Distance                 25,9714
    Focal Shape Size         0,05
    Focal Shape Fixed Size   ✓
    Yaw                      4,71359
    Pitch                    1,5698
    Field of View            0,785398
    Focal Point              2.0506; 1.9462; -3.2272
```

```
Synchronization: Off   ROS Time: 1771001595.98   ROS Elapsed: 244.78   Wall Time: 1771001596.01   Wall Elapsed: 244.72
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.   31 fps
```

Figure 5.6: RViz plugin displaying a uniform, unweighted NURBS curve

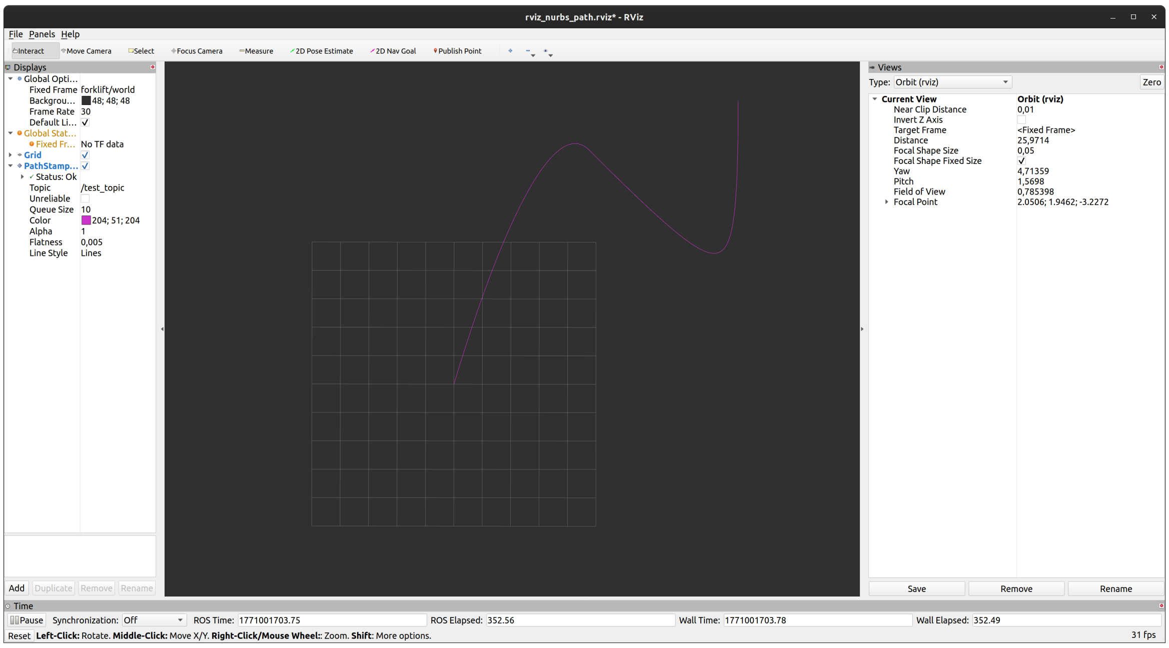


```
Displays
Global Opti…
  Fixed Frame   forklift/world
  Backgrou…     48; 48; 48
  Frame Rate    30
  Default Li…   ✓
Global Stat…
  Fixed Fr…     No TF data
Grid            ✓
PathStamp…      ✓
  Status: Ok
  Topic         /test_topic
  Unreliable
  Queue Size    10
  Color         204; 51; 204
  Alpha         1
  Flatness      0,005
  Line Style    Lines
```

```
Views
Type: Orbit (rviz)
Current View            Orbit (rviz)
  Near Clip Distance    0,01
  Invert Z Axis
  Target Frame          <Fixed Frame>
  Distance              25,9714
  Focal Shape Size      0,05
  Focal Shape Fixed Size ✓
  Yaw                   4,71359
  Pitch                 1,5698
  Field of View         0,785398
  Focal Point           2.0506; 1.9462; -3.2272
```

```
Time
Pause  Synchronization: Off  ROS Time: 1771001506.92  ROS Elapsed: 155.72  Wall Time: 1771001506.95  Wall Elapsed: 155.66
Reset  Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.  30 fps
```

Figure 5.7: RViz plugin displaying a uniform, weighted NURBS curve



```
Displays
Global Opti…
  Fixed Frame   forklift/world
  Backgrou…     48; 48; 48
  Frame Rate    30
  Default Li…   ✓
Global Stat…
  Fixed Fr…     No TF data
Grid            ✓
PathStamp…      ✓
  Status: Ok
  Topic         /test_topic
  Unreliable
  Queue Size    10
  Color         204; 51; 204
  Alpha         1
  Flatness      0,005
  Line Style    Lines
```

```
Views
Type: Orbit (rviz)
Current View            Orbit (rviz)
  Near Clip Distance    0,01
  Invert Z Axis
  Target Frame          <Fixed Frame>
  Distance              25,9714
  Focal Shape Size      0,05
  Focal Shape Fixed Size ✓
  Yaw                   4,71359
  Pitch                 1,5698
  Field of View         0,785398
  Focal Point           2.0506; 1.9462; -3.2272
```

```
Time
Pause  Synchronization: Off  ROS Time: 1771001703.75  ROS Elapsed: 352.56  Wall Time: 1771001703.78  Wall Elapsed: 352.49
Reset  Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.  31 fps
```

Figure 5.8: RViz plugin displaying a non-uniform, weighted NURBS curve

6 Applications and improvements

6.1 NURBS library applications

The library was designed with mobile robot navigation in mind and as such it has nearly everything necessary to define and use 2-dimensional paths.

The library was created for the purpose of designing top-down paths for vehicles which move indoors, on flat floors, and as such works only in 2D space. As it stands it can be used for any 2D application, including controlling robot manipulators, e.g. in plotting or CNC machines.

The `piecewiseBezier` method can be used to convert NURBS curves into Bézier curves which, unlike NURBS, are supported in most graphics applications.

6.2 Improvements and missing features

One possible improvement to the library would be the ability to define NURBS curves in any dimension, including 1D and from 3D onward. This could be achieved using Eigen's class templates which can define the dimensions of matrices at compile time, in this case the width of the control point matrix and the `_cached_vbf` matrix. However, expanding the library to NURBS surfaces in 2-dimensional *parametric* space (parameters t and v) would require significant refactoring as all the methods rely on a scalar t value.

Three-dimensional splines can be used to define precise paths for industrial robot manipulators. One-dimensional splines have limited uses, however one possible application would be to directly control the joints of a robot with precise control over its timing and speed.

The library was written using the universally adopted *C++17* standard, meaning that it misses out on some features of modern C++. Additionally, during development Eigen was updated to version *5.0.0* and ROS2 has become the standard in favor of ROS1, which is slowly being deprecated. Using modern standards could result in better optimization and performance.

Some common operations on NURBS curves, such as raising and lowering curve order, have not been implemented or have been implemented partially. These were not necessary for the task of planning mobile robot paths. However, these functions must exist in a fully-featured NURBS library and may be implemented later. For example, in theory, the derivative of a span of degree p can itself be expressed as a span of degree $p-1$, though this has not been implemented. Additionally, some methods were adapted directly from sources that were written many years ago and not with modern software architecture in mind. Though the most time-critical and commonly used methods have already been optimized, rewriting these methods could improve both the readability and performance of the library.

The example application could use more features for the analysis of NURBS curves, such as the derivative and curvature plots from the previous chapter. These could be implemented using *QCustomPlot* much like the basis function plots. A feature to export curve data in a standardized format such as *JSON* or *YAML* could prove useful in testing and debugging curves.

6.3 Future work

One possible way to extend the library for ease of use is to create Python bindings to allow scripting with NURBS curves. This would prevent the user from having to write and recompile C++ code for simple tasks such as plotting, as was done in Chapter 5. This would also allow interoperability with other popular Python libraries such as *NumPy* and *matplotlib*.

The next steps for the library and its ROS message type are to implement missing features in the library, and to integrate the library with existing roadmap planning software for the user and navigation software for mobile robots.

7 Conclusion

Among all types of parametric curves, Bézier curves are the most widely used as they are easy to implement and fast to compute. NURBS are the most flexible, however they have historically been avoided for their complexity and unintuitive design. If implemented properly, they can be an effective alternative to both Bézier curves and B-splines, as NURBS are a superset of both. For use in mobile robot navigation, they provide some features that may be difficult or impossible to implement using other parametric curves.

Geometric continuity is presented as one of the major requirements that a spline appropriate for mobile robot paths must fulfill. Namely, G^3 continuity is necessary for the derivative of the curvature to be continuous, which in turn minimizes stops in the robot's path as well as strain on its steering mechanism. A fourth degree NURBS curve was proven to have G^3 continuity along its length without the extra work often needed to maintain continuity.

Through the use of vectorization it is possible to compute NURBS curves and their operations at a speed which is suitable for real-time use and comparable to Bézier curves. The library's simplicity and abstraction allows its users to implement transformation methods in a way that is intuitive for their end-user and appropriate for designing paths for autonomous mobile robots, as shown by the example program made in Qt and the RViz plugin.

The library is demonstrated to efficiently compute NURBS curves of the fourth degree. Its curvature and derivatives were plotted to confirm geometric continuity. However, the development and testing of the library have also revealed several areas of improvement and possible features for the future.

Acknowledgements

This work was produced with help from *Romb Technologies*. Parts of the *NURBS* library were adapted from the *Bézier* library by Mirko Kokot. [1] Many of the algorithms were adapted from *The NURBS Book* by Piegl and Tiller. [5]

The source code for the library and example program is available on GitHub. [2]

[1]Available at `https://github.com/romb-technologies/Bezier`.
[2]Available at `https://github.com/romb-technologies/NURBS/`

References

[1] B. A. Barsky and A. D. DeRose, "Geometric continuity of parametric curves," Tech. Rep. UCB/CSD-84-205, Oct 1984. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1984/5752.html

[2] M. Kokot, D. Miklić, and T. Petrović, "Path continuity for multi-wheeled agvs," 2021. [Online]. Available: https://arxiv.org/abs/2103.01619

[3] M. Kokot, "Optimal path following method for a generalized kinematic model of vehicles with steer-and-drive wheels," Ph.D. dissertation, Faculty of Electrical Engineering and Computing, University of Zagreb, 2025.

[4] "A primer on bézier curves," 2016. [Online]. Available: https://pomax.github.io/bezierinfo/

[5] L. Piegl and W. Tiller, *The NURBS book*. Springer Berlin Heidelberg, 1997.

[6] G. Farin, "From conics to nurbs: A tutorial and survey," *IEEE Computer Graphics and Applications*, vol. 12, pp. 78–86, 1992. https://doi.org/10.1109/38.156017

[7] K. Qin, "General matrix representations for b-splines," in *Proceedings Pacific Graphics '98. Sixth Pacific Conference on Computer Graphics and Applications (Cat. No.98EX208)*, 1998, pp. 37–43. https://doi.org/10.1109/PCCGA.1998.731996

[8] A. Bellamy-Royds, T. Bah, C. Lilley, D. Schulze, and E. Willigers, "Scalable vector graphics (svg) 2," World Wide Web Consortium (W3C), W3C Candidate Recommendation, Oct. 2018. [Online]. Available: https://www.w3.org/TR/SVG2/

[9] G. Guennebaud, B. Jacob *et al.*, "Eigen," https://libeigen.gitlab.io, 2010.

Abstract

Design and visualisation of mobile robot paths

Ana Marija Devčić

The thesis describes the design of mobile robot paths, with a focus on material handling vehicles, such as forklifts. It analyzes the differences between Bézier curves, B-splines and NURBS for this purpose, with the most important property being G3 continuity. A matrix notation of NURBS is described. A C++ library for the creation and usage of NURBS curves was developed using the Eigen library. The thesis also describes the visualization of parametric curves and the tools most commonly used in robotics. An interactive Qt desktop application was created for testing the NURBS curve library, and an RViz plugin was developed to be used with mobile robots in practice. The library and plugin are tested on a series of examples and the results show that the library is suitable for use in mobile robot path planning.

Keywords: NURBS; mobile robots; parametric curves; Eigen; geometric continuity; Qt

Sažetak

Dizajn i vizualizacija putanja robotskih vozila

Ana Marija Devčić

Rad opisuje dizajn putanja mobilnih robota, pogotovo vozila koja prevoze teret, kao viličari. Analizira razlike između Bézierovih krivulja, B-spline krivulja i NURBS krivulja u tu svrhu, s najvažnijim svojstvom da ima G3 kontinuitet. Opisan je matrični zapis NURBS krivulja. Razvijena je C++ biblioteka za stvaranje i korištenje NURBS krivulja korištenjem Eigen biblioteke. Rad također opisuje vizualizaciju parametarskih krivulja i alate koji se najčešće koriste u robotici. Izrađena je interaktivna Qt desktop aplikacija za testiranje NURBS biblioteke krivulja, a razvijen je i RViz dodatak za korištenje s mobilnim robotima u praksi. Biblioteka i dodatak testirani su na nizu primjera, te rezultati pokazuju da je biblioteka prikladna za korištenje u planiranju putanja mobilnih robota.

Ključne riječi: NURBS; mobilni roboti; parametarske krivulje; Eigen; geometrijski kontinuitet; Qt

Appendix A: NURBS.h

```
class Curve
{
    public:
    ~Curve() = default;
    Curve(Eigen::MatrixX2d points, int p = 3);
    Curve(const PointVector& points, int p = 3);
    Curve(Eigen::MatrixX3d wpoints, Eigen::ArrayXd knotvector, int p =
     ↪  3);
    Curve(const Curve& curve);
    Curve(Curve&&) = default;
    Curve& operator=(const Curve&);
    Curve& operator=(Curve&&) = default;
    unsigned order() const;
    void elevateOrder(uint t = 1);
    void lowerOrder();
    PointVector controlPoints() const;
    Point controlPoint(unsigned idx) const;
    void setControlPoint(unsigned idx, const Point& point);
    std::pair<Point, Point> endPoints() const;
    void reverse();
    PointVector polyline(double flatness = 0.5) const;
    Point valueAt(double t) const;
    Eigen::MatrixX2d valueAt(const std::vector<double>& t_vector)
     ↪  const;
    BoundingBox boundingBox(bool use_roots = true) const;
    const Curve& derivative() const;
    const Curve& derivative(unsigned n) const;
    Vector derivativeAt(double t) const;
    Vector derivativeAt(unsigned n, double t) const;
    std::vector<double> roots() const;
    std::vector<double> extrema() const;
    double curvatureAt(double t) const;
    double curvatureDerivativeAt(double t) const;
    Vector tangentAt(double t, bool normalize = true) const;
    Vector normalAt(double t, bool normalize = true) const;
    double projectPoint(const Point& point) const;
    PointVector intersections(const Curve& other) const;
    double weight(int idx) const;
    void setWeight(int idx, double value);
    Eigen::VectorXd weights() const;
    Eigen::ArrayXd knotVector() const;
    void setKnot(int idx, double value);
    double knot(int idx) const;
    void appendPoint(Point point);
    void insertKnot(double t, int r);
    std::pair<Curve, Curve> splitCurve(double t) const;
    std::vector<Curve> piecewiseBezier() const;
    Eigen::VectorXd getBasisFunctionsAt(double t) const;
    int getKnotSpanIndex(double t) const;
    double length() const;
    double length(double t) const;
    void removeKnot(int ix, int k = 1);
    void removeControlPoint(int ix);
    Curve join(Curve& other);
    void applyContinuity(const Curve& source_curve, const
     ↪  std::vector<double>& beta_coeffs);

    protected:
    Eigen::MatrixX3d weighted_control_points_;
    inline void resetCache();

    private:
    unsigned N_{};
    unsigned p_{};

    mutable std::optional<std::vector<double>> cached_roots_;
    mutable std::optional<BoundingBox> cached_bounding_box_;
    mutable std::optional<PointVector> cached_polyline_;
    mutable double cached_polyline_flatness_{};

    Eigen::ArrayXd T_;
    mutable std::vector<Span> spans_;

    Span& getKnotSpan(double t) const;
    int getKnotMultiplicity(double t) const;
    void normalizeKnotVector();
};
```

Appendix B: Span.h

```cpp
class Span
{
public:
  ~Span() = default;
  Span(Eigen::Ref<Eigen::MatrixX3d> wpoints_,
   ↪  Eigen::Ref<Eigen::ArrayXd> knot_v, uint p);

  Span(Eigen::Ref<const Eigen::MatrixXd> basis_func, Eigen::Ref<const
   ↪  Eigen::MatrixXd> vbf, Eigen::Ref<const Eigen::RowVectorXd> wbf,
   ↪  double start, double end);

  inline double start() const { return start_; }
  inline double end() const { return end_; }
  bool contains(double t) const;
  void update(Eigen::Ref<Eigen::ArrayXd> knots,
   ↪  Eigen::Ref<Eigen::MatrixX3d> wpoints);
  void updateControlPoints(Eigen::Ref<Eigen::MatrixX3d> wpoints);
  Eigen::MatrixXd basisFunction() const;
  PointVector polyline(double flatness = 0.5) const;
  Point valueAt(double u) const;
  Point derivativeAt(int n, double u) const;
  Point derivativeAt(double u) const;
  void resetCache();
  double length() const;
  double length(double t) const;
  Eigen::RowVectorXd cachedWBF() const;
  Eigen::MatrixXd cachedVBF() const;
  BoundingBox boundingBox() const;
  std::vector<double> extrema() const;
  PointVector intersections(const Span& other) const;

private:
  mutable std::optional<double> cached_length_;
  mutable std::optional<PointVector> cached_polyline_;
  mutable std::optional<Eigen::VectorXd> cached_chebyshev_coeffs_;
  mutable std::optional<BoundingBox> cached_bounding_box_;

  Eigen::MatrixXd basis_function_;
  Eigen::RowVectorXd cached_wbf_;
  Eigen::MatrixXd cached_vbf_;
  const uint p_;
  double start_, end_;

  static inline BoundingBox fastBoundingBox(const Eigen::MatrixXd& vbf,
    ↪  const Eigen::RowVectorXd& wbf, const Eigen::MatrixXd&
    ↪  inverse_basis_function);
};
```