Optimizing Contour Lines on Bathymetric Charts

Andreas Østergaard Jakobsen, 201908104

Jonas Skøtt Dam, 201906295

Master's Thesis in Computer Science

June 14, 2024

Advisor: Gerth Stølting Brodal

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

This thesis concerns optimizing contour lines on bathymetric charts. Given accurate depth data and and initial contours approximating the accurate depths, we wish to produce optimized contour lines such that (1) the maximum curvature along the contour is minimized and (2) the area of the deeper parts of the chart are maximized.

We define an optimization problem modeling the task, with a loss function capturing the two desired properties as well as constraints defining feasible solutions. To do this, we make extensive use of Bézier curves as a precise way to represent curves mathematically, and as such we describe existing theory of Bézier curves as a foundation.

We develop an optimization algorithm capable of producing solutions to the optimization problem defined. To this extent we analyze the loss function of the optimization problem and conclude that exploiting properties of it is not a viable approach. Instead, we consider the loss function as a black box, and describe the direct search methods Compass Search and Combined Compass Search as well the random search method Fixed Step Size Random Search (FSSRS) for traversing the solution space. We benchmark the methods against each other and find that FSSRS is able to find the lowest-loss solution when the methods are run for the same amount of time.

Lastly, we describe how we use variations of the Hilbert R-tree data structure to increase the efficiency of the optimization algorithm in regards to checking if a solution is feasible. From benchmarks we see that using Hilbert R-trees can decrease the constraint checking query time by a factor up to 775 on the largest dataset we tested.

Acknowledgements

We would like to thank our advisor Gerth Stølting Brodal for guiding us throughout this project. We would also like to thank Jacob Truelsen for providing us with the idea for the problem and datasets considered in this thesis. Finally, we thank Mathias Rav for his feedback on the thesis.

Andreas Østergaard Jakobsen,
Jonas Skøtt Dam,
Aarhus, June 14, 2024.

Contents

Abstract ii

1 Introduction **1**
1.1 Introduction to Optimization Problem 2
1.2 Introduction to Datasets 3
1.3 Reading Guide 4

2 Theory of Bézier Curves **6**
2.1 Bernstein Polynomials 6
2.1.1 Partition of Unity 7
2.2 Definition of a Bézier Curve 7
2.2.1 Convex Hull Property 8
2.3 De Casteljau's Algorithm 9
2.4 Computing Roots of a Curve 13
2.4.1 Quadratic curves – the Quadratic formula 13
2.4.2 Cubic curves – Cardano's method 14
2.5 Continuity of Connected Bézier Curves 14
2.5.1 Parametric Continuity 14
2.5.2 Geometric Continuity 14
2.6 Axis-aligned Bounding Boxes 15
2.6.1 Convex hull based approach 15
2.6.2 Derivative based approach 15
2.7 Intersection of Curve and Line Segment 16
2.8 Intersection of Curve and Curve 17
2.8.1 Subdivision method 17
2.9 Interior Point Problem 18
2.10 Curvature 20
2.10.1 Ternary Search Method for Maximum Curvature 21
2.11 Computing Area 21

3 Contours as an Optimization Problem **24**
3.1 Optimization in Standard Form 24
3.2 Bézier Spline Representation 25
3.3 Solution Representation 26
3.4 Constants 26
3.4.1 Defining the Chart Scale 26
3.4.2 Accurate Contours 26
3.4.3 Target Curvature and Area Score 26
3.5 Defining the Loss Function 27
3.5.1 Metric: Curvature 27
3.5.2 Metric: Area 28
3.5.3 Combining both Metrics 29

3.6 Defining the Constraints . . . 30
3.6.1 Depth Constraint . . . 30
3.6.2 Intersection Constraint . . . 32
3.6.3 Topology Constraint . . . 33
3.6.4 Static Endpoint Requirement . . . 34
3.7 Final Optimization Problem . . . 34
3.8 Analysis of the Optimization Problem . . . 34
3.8.1 Examining the Loss Function . . . 35

4 Black Box Optimization **36**
4.1 Direct Search Methods . . . 36
4.1.1 Compass Search . . . 36
4.1.2 Combined Compass Search . . . 37
4.2 Random Search . . . 38
4.2.1 Rastrigin's Family of Random Searches . . . 38
4.2.2 Alternative Step Sizes . . . 38

5 Constraint Checking with R-trees **41**
5.1 The Original R-tree . . . 41
5.1.1 Dynamic and Static Variations . . . 42
5.2 The Hilbert Packed R-tree . . . 43
5.2.1 The Hilbert Curve . . . 43
5.2.2 Packing a Tree . . . 43
5.3 Lazy Update R-tree . . . 44
5.3.1 Dynamic Trees Not Needed . . . 44

6 Contour Optimization Algorithm **46**
6.1 Searching for Solutions . . . 46
6.1.1 Local Updates Only . . . 47
6.2 Implementation of Constraint Checking . . . 47
6.3 Configuration Summary . . . 48
6.4 Correctness . . . 49
6.5 Visualization of Results . . . 49

7 Benchmarks **53**
7.1 Comparing Configurations by Loss . . . 53
7.2 Behavior of Area and Curvature . . . 56
7.3 Hyperparameter Optimization . . . 58
7.4 R-tree Performance . . . 60

8 Conclusion **63**

Bibliography **64**

A Loss over 10 000 **Steps** **67**

B Curvature vs. Area **68**

C Visualization of Datasets D2 and D3 **70**

Notation	Definition
$B_i^n(t)$ / B^n	Bernstein basis polynomial.
$\mathbf{P}_i$	The i-th control point of a Bézier curve.
$\mathbf{P}_i^k$	Intermediate point in de Casteljau's algorithm. In this case the parameter is implied, so in reality shorthand for $\mathbf{P}_i^k(t)$.
$\boldsymbol{B}_n(t)$	Bézier curve of degree n.
$\boldsymbol{B}[\mathbf{P}_0,\dots,\mathbf{P}_n](t)$	Bézier curve with specified control points. In cases where the parameter is omitted e.g. $\boldsymbol{B}[\mathbf{P}_0,\dots,\mathbf{P}_n]$ the curve is still parameterized implicitly.
$x = (v_0,\dots,v_n)$	x is a vector of values $v_0,\dots,v_n$
$x = \{v_0,\dots,v_n\}$	x is a set of values $v_0,\dots,v_n$
$\boldsymbol{p},\boldsymbol{u},\boldsymbol{v}$	Lowercase boldface math letters are vectors in $\mathbb{R}^2$.
$\boldsymbol{p}^x$	The first entry of $\boldsymbol{p}$.
$\boldsymbol{p}^y$	The second entry of $\boldsymbol{p}$.
$\ell = [\boldsymbol{p}_0,\boldsymbol{p}_1]$	A line segment consisting of all points on the linear interpolation between $\boldsymbol{p_0}$ and $\boldsymbol{p_1}$.
$\mathbf{S}$	A Bézier spline; represented by a sequence of connected Bézier curves.
$\mathbf{S}^h$	The height of all points along a Bézier spline $\mathbf{S}$.
$\boldsymbol{X}$	A solution to the optimization problem; represented as a set of Bézier splines.
$\boldsymbol{A}$	An accurate contour line; represented as a sequence of line segments.
$\boldsymbol{A}^h$	The height of all points along an accurate contour $\boldsymbol{A}$.
$\boldsymbol{C}$	A set of accurate contours
$C^{\mathbf{S}}$	Set of accurate contours in $\boldsymbol{C}$ of height $\mathbf{S}^h$
$X^{\mathbf{S}}$	Set of Bézier splines in $\boldsymbol{X}$ of height $\mathbf{S}^h$
$\boldsymbol{p} \in \boldsymbol{B}$	$\boldsymbol{p}$ is a point on the Bézier curve in the interval $t \in [0,1]$
$\boldsymbol{B} \in \mathbf{S}$	$\boldsymbol{B}$ is a Bézier curve in the spline $\mathbf{S}$

Chapter 1

Introduction

This thesis concerns the problem of simplifying contour lines on bathymetric charts. *Bathymetric charts* are, generally, maps that convey information about the depth of bodies of water. Analogously to topological charts that concern themselves with the height of terrain above water, a 2D bathymetric chart illustrates water depth by using contour lines. In this context, *contour lines* are curves connecting points on the map of an equal depth d, put differently all points on a contour line represent an exact depth of d. This data could be of particular interest to, for example, ships navigating at sea, as the water depth dictates what paths are possible for the ships to take.

With modern technology, it is possible to measure water depth at an incredible level of detail. However, a very high level of detail is not always desirable in all applications. This project considers the aforementioned setting, where a human navigator perceives the bathymetric chart and wants to read off useful information from it.

A lot of detail in the data can easily lead to visual clutter when plotted, making it hard to gain any useful insights from looking at it. As such, it is often necessary to use simplifications of the real depth data in charts that have to be perceived by humans, simply to improve readability. In order to go from real-depth data to a readable chart, several steps are involved. We highlight the most important steps below to better put our work into context:

First, **detailed data** of the seabed is collected and put into a digital elevation model (DEM) [11].

Contour lines can be extracted from the DEM as points connected by line segments [2, 1]. We refer to these as **accurate contour lines**.

Contours represented by connected line segments can now be approximated with *curves*, giving what we refer to as **initial contour lines** (since they are input to the algorithms studied in this thesis).

The initial contour lines are changed iteratively in an effort to improve the approximation by (1) making them more true to the accurate contours while (2) enhancing the readability to human observers. We can refer to these as **optimized contour lines**.

Putting it very succinctly and in terms of the above flow, our work is concerned with the last part mentioned, namely taking both accurate and initial contour lines as input and outputting optimized contour lines. The different types of contours are illustrated in Figure 1.1. Our approach to the above problem will be to define a mathematical optimization problem with

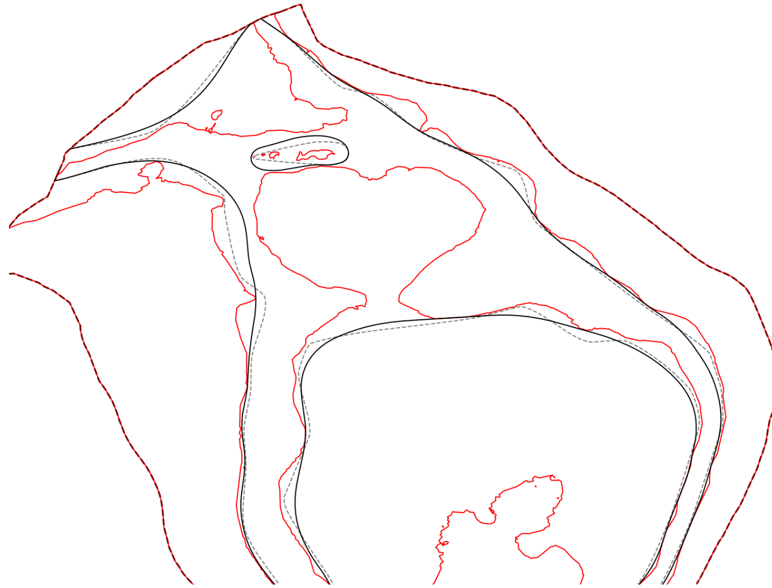

Figure 1.1: An illustration of the different kinds of contour lines. Dark red dashed lines indicate the boundary of the dataset. Red lines are accurate contours, gray dashed lines are initial contours and black lines are optimized contours.

an objective function subject to some constraints. As such, we need to introduce a way of representing contour lines, which in the present case are Bézier curves. We of course also present an optimization algorithm for solving the said problem. This algorithm can be run in different configurations, each of which we conclude by evaluating experimentally. In order to give a more concrete idea of the optimization problem we are interested in solving in this thesis, we will introduce the objective and the constraints at an intuitive level in the following section.

1.1 Introduction to Optimization Problem

Recall that the goal is, on a high level, to optimize the initial contour lines to decrease visual clutter and thereby increase human readability while not deviating too much from the accurate contours. This objective is not precise enough to give a uniquely defined optimization problem and thus more well-defined metrics are required. The problem is defined according to an objective subject to constraints.

Objective The measures we aim to optimize for are the *curvature* and the *area* of the contour lines. Curvature is formally described in Section 2.10 and Section 3.5 contains a precise description of how the objective is defined as a function of curvature and area. For now, we just provide a visual interpretation of their effect on the contours.

Firstly, curvature is a measure of how sharply a curve turns/bends at a given point. It is defined in such a way that a high curvature corresponds to a sharper turn, while a low curvature corresponds to a less sharp turn. With this in mind, we would like to output contours that have

low curvature. More precisely we desire a curvature lower than some predefined *target curvature*.

Secondly, the area measure in this case concerns the area of the deeper regions of the contour map. The thinking is that maximizing deeper areas of the map pushes the optimized contours closer to the accurate contours, thus increasing the navigable area for vessels. The importance of the area depends on a predefined *area score*.

Both the curvature and area metric are defined in detail in Chapter 3, where they are combined into a single so-called *loss function*. The loss function defines an exact way to compare the quality of one contour map with another with respect to the mentioned metrics. The target curvature and area score used for this project are provided in Section 3.4.

Constraints Constraints are used to ensure that the contour maps we produce have certain properties. In essence they ensure that the contour lines convey information that is in line with the underlying data. It is desirable that an optimized solution consist of valid contour lines that are safe for navigation at sea. For this purpose we introduce three types of constraints.

Firstly, we must obey the *intersection constraint* to keep contour lines meaningful. The constraint simply requires that no contour line intersects with another contour line or itself. The reason for this constraint comes from the fact that, as mentioned, contour lines represent points having the exact same depth. If two contour lines intersect and have different depths, this indicates the point of intersection has two depths. In reality this is meaningless and should be considered misbehavior.

Secondly, we have the *depth constraint*. According to the depth constraint, we must ensure that the deep region of any optimized contour is entirely contained inside the deeper region of the accurate contour. This has the effect that an optimized contour always underestimates the depth. Without this constraint, there might be areas where the optimized contour line overestimates the depth thus encouraging vessels to travel where they might go aground.

Lastly, the *topology constraint* ensures that the optimized contour lines are in line with the underlying data. The constraint requires contour lines to keep the same relation between each other. That is, if a contour line is in one region of another contour before the optimization, it must also be in the same region after the optimization.

1.2 Introduction to Datasets

In this thesis we have access to three datasets which we refer to as D1, D2, and D3. D1 is small and mostly used for testing, validation, and illustrative purposes. D2 and D3 are larger and more aligned with what we aim to develop a method to solve.

Each dataset consists of two main components: (1) accurate contours modeling what we think of as the true water depth and (2) initial contours modeling an approximation of the accurate contours. Accurate contours are represented as polylines, that is, sequences of connected line segments. Initial contours are represented with splines. Splines are sequences of connected curves, which allow for a much more space-efficient representation. The curves, in this case, are *Bézier curves* which we cover in detail in Chapter 2. A subset of the accurate contours is used to define the *boundary* of the datasets, as boundaries are represented as contours of zero depth. The boundary consists of one or multiple closed polylines that split the plane into an interior and exterior region. All other geometries of our dataset lie in the interior region of the boundary. Figure 1.2 illustrates a part of D1. As seen in the figure, each initial contour is an approximation of an accurate contour. The amount of data points in each dataset is shown in Table 1.1. This is to note that the initial contour has fewer points than the accurate contours and thus require less space to store.

Properties: The contours in the dataset obey certain properties that are important with regard to having a well-defined problem. Specifically, the initial contours given satisfy the depth,

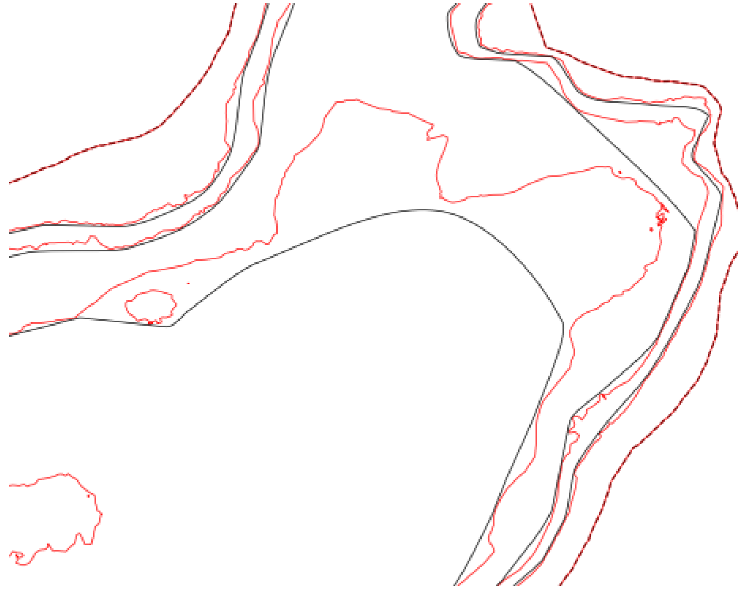

Figure 1.2: A cutout of D1 showing accurate contours in red, initial contours in black, and the boundary as a red-black dashed line.

	Accurate Contours Data Points	Initial Contours Data Points
D1	7 994	1 087
D2	403 720	48 278
D3	427 803	92 606

Table 1.1: Amount of two-dimensional data points used to represent the accurate contour and initial contour, respectively, in each dataset.

intersection, and topology constraints introduced above. Additionally, the endpoints of the splines modeling both the accurate contours and the initial solution are either connected to each other (forming a closed curve) or to the boundary of the dataset. This ensures the contours in combination with the boundary are continuous. Finally, contours are given in either clockwise or counter-clockwise order. This order helps determine which side of the contour corresponds to points of deeper or shallow elevation. For any contour, the port side is the shallow region and the starboard side is the deep. Figure 1.3 illustrates this.

1.3 Reading Guide

We will present the work undertaken in the following order. In Chapter 2 we define the Bézier curve along with several important properties of them. In the same chapter, we also describe different computations that can be performed with Bézier curves. In Chapter 3 we present our optimization problem in detail. This entails defining the loss function as well as the constraints.

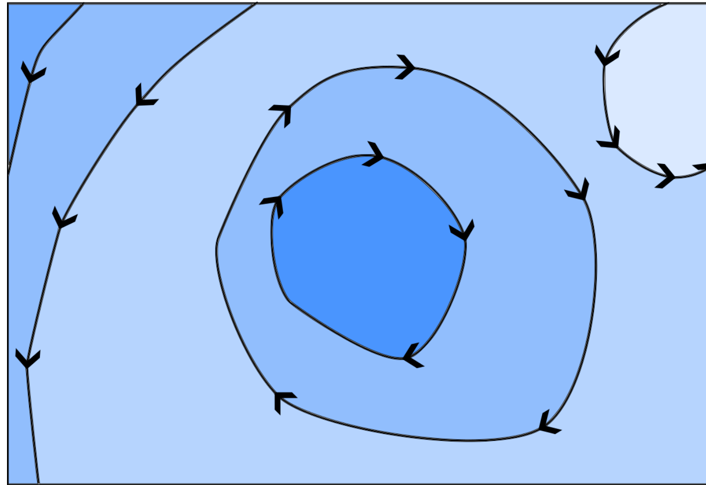

Figure 1.3: Arrows indicate the order of contour lines. Light and dark blue areas represent shallow and deep regions respectively.

We also provide a small analysis of which type of problem we categorize it as in Section 3.8. In order to frame our approach to implementing an algorithm for solving the optimization problem, we dedicate the next two sections to describing more general theories on solving optimization problems as well as specialized data structures. Concretely, Chapter 4 describes different ways of searching the solution space of the problem while Chapter 5 introduces R-trees as a data structure helpful for determining the feasibility of solutions. In Chapter 6 we give a description of the optimization algorithm that has been implemented and show the output it can produce. The last section, Chapter 7, presents the results of running benchmarks on the implementation, comparing different configurations and hyperparameters.

Chapter 2

Theory of Bézier Curves

Bézier curves are a tool for representing curves by means of control points. Bézier curves get their name from the French engineer Pierre Bézier (1910-1999), and were developed as part of his work at the car manufacturer Renault, supposedly in order to aid the design of automobile bodies and other parts. Since their introduction in the 1960s, Bézier curves and related splines have become a fundamental part of most *computer aided design* (CAD) software and computer graphics in general [32].

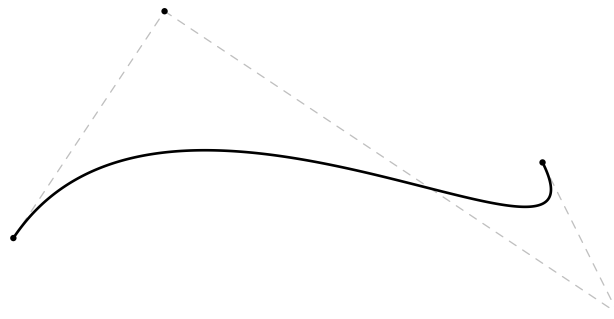

Figure 2.1: A cubic Bézier curve defined by four control points. Control points are connected by a dashed line only for illustrative purposes.

In the following sections, we will present the formal definition of a Bézier curve as well as different geometric problems relevant to the project. In each subsection, we briefly introduce how the problems relate to optimizing contour lines. The descriptions are based on the textbook by Gerald E. Farin [10] and lecture notes by Thomas W. Sederberg [32].

2.1 Bernstein Polynomials

Mathematically, a Bézier curve is a parametric function/curve defined by a linear combination of polynomials [10]. Essentially this means that both the x and y coordinates of all points on the curve are defined by the sum of specific polynomials parameterized by a common parameter t.

Thus, we begin by presenting the foundation upon which Bézier curves are built, namely the *Bernstein polynomials*. These polynomials are referred to by their degree, as we are used to with polynomials in standard form. However, a single Bernstein polynomial of some degree $n \geq 1$ has a set of so-called *basis polynomials* associated with it. Explicitly the definition of the i-th

Bernstein polynomial of degree n is

$$B_i^n(t) = \binom{n}{i}(1-t)^{n-i}t^i \ ,$$

for $i = 0, \ldots, n$ and $t \in [0,1]$. If we fix the degree to some specific number, say, $n = 3$, the four resulting polynomials can be nicely visualized as in Figure 2.2.

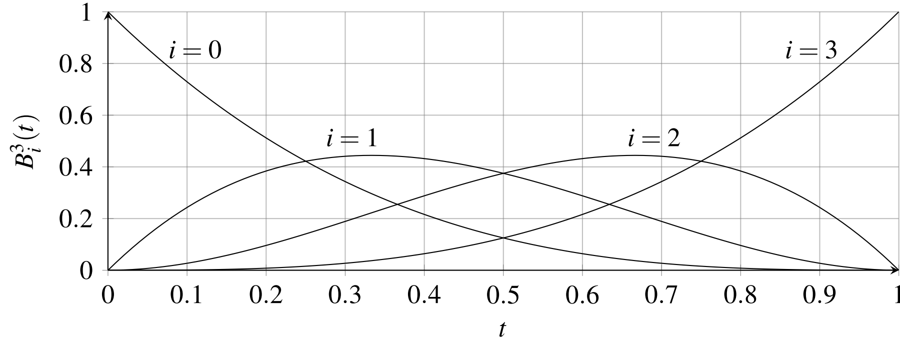

Figure 2.2: Plot of all Bernstein polynomials of degree $n = 3$.

2.1.1 Partition of Unity

An important property of Bernstein polynomials of *any* degree is that they form a *partition of unity* meaning

$$\sum_{i=0}^{n} B_i^n(t) = 1 \ . \tag{2.1}$$

This fact can be proved with the *binomial theorem*, which describes how to expand a binomial (i.e., a polynomial with only two terms, in this case, a and b) raised to an integer power $m > 0$ and states that

$$(a+b)^m = \sum_{k=0}^{m} \binom{m}{k} a^{m-k} b^k \ . \tag{2.2}$$

The proof follows directly, as the definition of a Bernstein polynomial is identical to the expanded binomial in Equation (2.2) giving

$$\sum_{i=0}^{n} B_i^n(t) = \sum_{i=0}^{n} \binom{n}{i}(1-t)^{n-i}t^i = ((1-t)+t)^n = 1 \ .$$

2.2 Definition of a Bézier Curve

A Bézier curve such as seen in Figure 2.1 arises when Bernstein polynomials are used in conjunction with *control points*. To define a Bézier curve $\boldsymbol{B}_n(t) = (x(t), y(t))$ of degree n, is a matter of combining the Bernstein polynomials of degree n and the control points $\mathbf{P}_i = (x_i, y_i)$ for $i = 0, \ldots, n$ as follows:

$$\boldsymbol{B}_n(t) = \sum_{i=0}^{n} \mathbf{P}_i B_i^n(t) = \sum_{i=0}^{n} \mathbf{P}_i \binom{n}{i}(1-t)^{n-i}t^i \ . \tag{2.3}$$

To make the control points of some curve more explicit, we will sometimes use the notation $\boldsymbol{B}[\mathbf{P}_0,\ldots,\mathbf{P}_n](t) = \boldsymbol{B}_n(t)$, where the degree of the curve can be seen by the number of control points in the list. Since a Bézier curve is a parametric function, we will use the term *component function* in order to refer to its constituents $x(t)$ and $y(t)$ individually. An example curve and its component functions are shown on Figure 2.3.

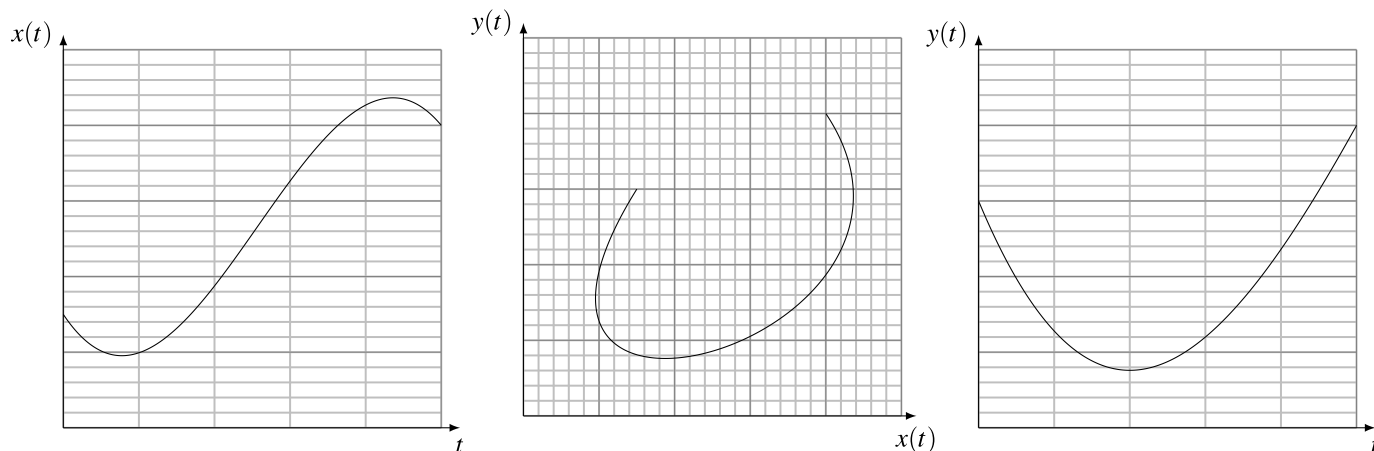

Figure 2.3: **Middle**: An example cubic Bézier curve $\boldsymbol{B}_3(t)$. **Left**: The component function $x(t)$ of the curve shown in the middle. **Right:** The component function $y(t)$ of the curve in the middle.

Interpreting the different parts of Equation (2.3) is relatively straightforward. The polynomials $B_i^n(t)$ can be seen as *blending functions* that dictate how much a particular control point $\mathbf{P}_i$ contributes to the position of the Bézier curve at a specific time t. This interpretation makes sense due to the partition of the unity property of Bernstein polynomials, so the contribution across all different points will always sum to 1. When $t \in [0,1]$ the choice of Bernstein polynomials for blending functions has the effect that the curve will always start in $\mathbf{P}_0$ and end in $\mathbf{P}_n$, a property referred to as *endpoint interpolation*. This follows directly from the definition: Say we are looking at a so-called *cubic* Bézier curve, i.e., one that has degree $n = 3$ and thus 4 control points. The curve will be expressed as below, and the described property is seen by evaluating at $t = 0$ and $t = 1$ respectively, so if

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

then $\boldsymbol{B}_3(0) = \mathbf{P}_0$ and $\boldsymbol{B}_3(1) = \mathbf{P}_3$. It should be noted that Bézier curves *can* be defined over arbitrary parameter ranges $t \in [t_0, t_1]$, in which case Equation (2.3) has to be modified slightly. Such a curve would be subscripted with the parameter range in the notation $\boldsymbol{B}_{[t_0,t_1]}(t)$. However, the curves in the present thesis are always with $t \in [0,1]$, so we refer to [32] for an introduction to Bézier curves over arbitrary parameter intervals.

2.2.1 Convex Hull Property

The *convex hull property* of a Bézier curve states that the curve will always be contained within the convex hull of the control points defining it. This is useful in several ways, most notably as a tool for determining intersections of curves as explained in Section 2.8 and storing curves in geometric data structures as explained in Section 2.6.1. Arguing for the convex hull property can be done by looking at two related concepts, namely a *convex combination* and its relation to the *convex hull*: Consider a set of points $p_0, p_1, \ldots, p_n$ in $\mathbb{R}^n$. A *convex combination* of these points is the linear combination

$$\lambda_0 p_0 + \lambda_1 p_1 + \ldots + \lambda_n p_n = \sum_{i=0}^{n} \lambda_i p_i$$

where coefficients λ_i must all be non-negative and all sum to one, i.e.,

$$\lambda_i \geq 0 \text{ for all } i \quad \text{and} \quad \sum_{i=0}^{n} \lambda_i = 1 \ .$$

One way to define the *convex hull* of the points is to say that it is the set of all possible convex combinations of $p_0, p_1, \ldots, p_n$ [7]. With these definitions in place, we can now argue the convex hull property of Bézier curves.

Firstly, any point on a Bézier curve is a convex combination of the control points of the curve. A Bézier curve is a linear combination of the control points, as seen from the definition, Equation (2.3). In addition, both conditions for being a convex combination are satisfied due to properties of Bernstein polynomials in the $[0,1]$ interval, namely non-negativity and partition of unity from Equation (2.1). Secondly, since all convex combinations of the control points taken together define the convex hull of the points, *any* convex combination of the control points are guaranteed to be within the convex hull. Thus, all points on the Bézier curve must be within the convex hull of the control points, since they are, at heart, just convex combinations of the control points.

2.3 De Casteljau's Algorithm

From the above definition of a Bézier curve, it is apparent that points on the curve can be computed by directly evaluating Equation (2.3) for different values of $t \in [0,1]$. However, this turns out to be a bad approach, as numerical instability becomes an issue when taking powers of very small values, when evaluating the Bernstein polynomials. The numeric instability of Bernstein polynomials is illustrated in Figure 2.4, where lower values of t result in a incorrect, jagged curve. Luckily, *de Casteljau's algorithm* [4] provides a method for evaluating points on a Bézier curve by means of repeated linear interpolation between only the control points of the curve. In this section, we present how Bézier curves can also be expressed recursively with linear interpolations and how this fact is utilized in de Casteljau's algorithm.

Linear interpolation: If two 2-dimensional points $\boldsymbol{u}$ and $\boldsymbol{v}$ are given, a point $\boldsymbol{p}$ at some ratio $t \in [0,1]$ along the straight line connecting the two points can be computed by linear interpolation (abbreviated **lerp**) using:

$$\boldsymbol{p} = (1-t)\boldsymbol{u} + t\boldsymbol{v} \ .$$

Linear interpolation of Bézier curves: Say we have a degree n Bézier curve $\boldsymbol{B}[\mathbf{P}_0, \ldots, \mathbf{P}_n](t)$ that we want to evaluate at some specific t. Instead of evaluating the polynomials directly, the exact same point can be computed by linear interpolation between points on two $n-1$ degree *sub-curves* that taken together use the same control points as the curve we are trying to compute. Concretely, the first sub-curve will use all the same control points as the original except endpoint $\mathbf{P}_n$, while the second sub-curve will exclude the endpoint $\mathbf{P}_0$ only. This is captured in the following theorem.

Theorem 2.3.1 (Interpolation of Bézier Curves) *The point at parameter value t on a degree n Bézier curve $\boldsymbol{B}[\mathbf{P}_0, \ldots, \mathbf{P}_n](t)$ can be computed by linear interpolation between points at parameter value t on two degree $n-1$ sub-curves as*

$$\boldsymbol{B}[\mathbf{P}_0, \ldots, \mathbf{P}_n](t) = (1-t)\boldsymbol{B}[\mathbf{P}_0, \ldots, \mathbf{P}_{n-1}] + t\boldsymbol{B}[\mathbf{P}_1, \ldots, \mathbf{P}_n] \ .$$

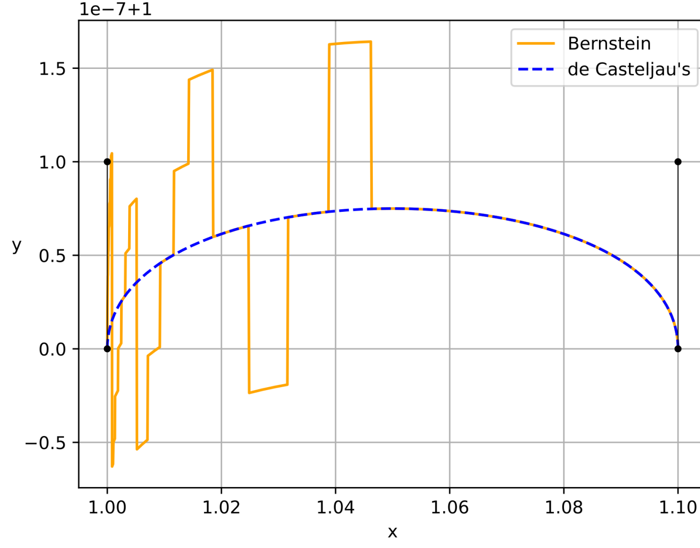

Figure 2.4: Plot showing a cubic Bézier curve computed with respectively Bernstein polynomials and de Casteljau's algorithm. Only de Casteljau's algorithm has the numerical stability to render the curve accurately.

The proof of this relies only on the definition of a Bézier cuve, Equation (2.3), and the recursive formula for the binomial coefficient.

$$
\begin{aligned}
&(1-t)\boldsymbol{B}[\mathbf{P}_0,\dots,\mathbf{P}_{n-1}]+t\boldsymbol{B}[\mathbf{P}_1,\dots,\mathbf{P}_n]\\
&=(1-t)\sum_{i=0}^{n-1}\mathbf{P}_i B_i^{n-1}+t\sum_{i=1}^{n}\mathbf{P}_i B_{i-1}^{n-1}\\
&=(1-t)\sum_{i=0}^{n-1}\mathbf{P}_i\binom{n-1}{i}t^i(1-t)^{n-i-1}+t\sum_{i=1}^{n}\mathbf{P}_i\binom{n-1}{i-1}t^{i-1}(1-t)^{n-i}\\
&=\sum_{i=0}^{n-1}\mathbf{P}_i\binom{n-1}{i}t^i(1-t)^{n-i}+\sum_{i=1}^{n}\mathbf{P}_i\binom{n-1}{i-1}t^i(1-t)^{n-i}\\
&=\mathbf{P}_0(1-t)^n+\mathbf{P}_n t^n+\sum_{i=1}^{n-1}\mathbf{P}_i\binom{n-1}{i}t^i(1-t)^{n-i}+\sum_{i=1}^{n-1}\mathbf{P}_i\binom{n-1}{i-1}t^i(1-t)^{n-i}\\
&=\mathbf{P}_0(1-t)^n+\mathbf{P}_n t^n+\sum_{i=1}^{n-1}\mathbf{P}_i\left(\binom{n-1}{i}+\binom{n-1}{i-1}\right)t^i(1-t)^{n-i}\\
&=\mathbf{P}_0(1-t)^n+\mathbf{P}_n t^n+\sum_{i=1}^{n-1}\mathbf{P}_i\binom{n}{i}t^i(1-t)^{n-i}\\
&=\sum_{i=0}^{n}\mathbf{P}_i\binom{n}{i}t^i(1-t)^{n-i}\\
&=\sum_{i=0}^{n}\mathbf{P}_i B_i^n\\
&=\boldsymbol{B}[\mathbf{P}_0,\dots,\mathbf{P}_n](t)
\end{aligned}
$$

Intuition behind de Casteljau's algorithm: We begin by fixing some terminology and notation. As mentioned in the definition, a Bézier curve of degree n has $n+1$ *control points* each denoted $\mathbf{P}_i$ (note only a single subscript letter i used) for $i = 0,\ldots,n$. Furthermore, de Casteljau's algorithm gives rise to what we call *intermediary points* each denoted $\mathbf{P}_i^k(t)$ (note both subscript i and superscript k) for $k = 0,\ldots,n$ and $i = 0,\ldots,k-n$. These are points on a k degree Bézier curve for a specific value of t. They are defined recursively from Theorem 2.3.1. Given a n degree Bézier curve $\boldsymbol{B}_n$ we have $\mathbf{P}_0^n(t) = \boldsymbol{B}_n(t)$. Other intermediate points are defined as

$$\mathbf{P}_i^n(t) = (1-t)\mathbf{P}_i^{n-1}(t) + t\mathbf{P}_{i+1}^{n-1}(t)$$

and are points on the two sub-curves defined in Theorem 2.3.1. Intermediary points are always functions of the parameter t, so when we use notation $\mathbf{P}_i^k$ the parameter is implicit. It can be helpful to think about the two types of points in the following way:

> **Control points** are always given when a Bézier curve is defined. They are fixed in space and are *not* a function of the parameter t.
>
> **Intermediary points** are always computed, in some way, from the control points. They appear as a result of linear interpolations between either control points or other intermediary points and, as such, they change as a function of the parameter t.

Note: If $k = 0$ in the notation $\mathbf{P}_i^k$, then the point is a control point of the curve e.g. $\mathbf{P}_0^0 = \mathbf{P}_0$ and $\mathbf{P}_3^0 = \mathbf{P}_3$.

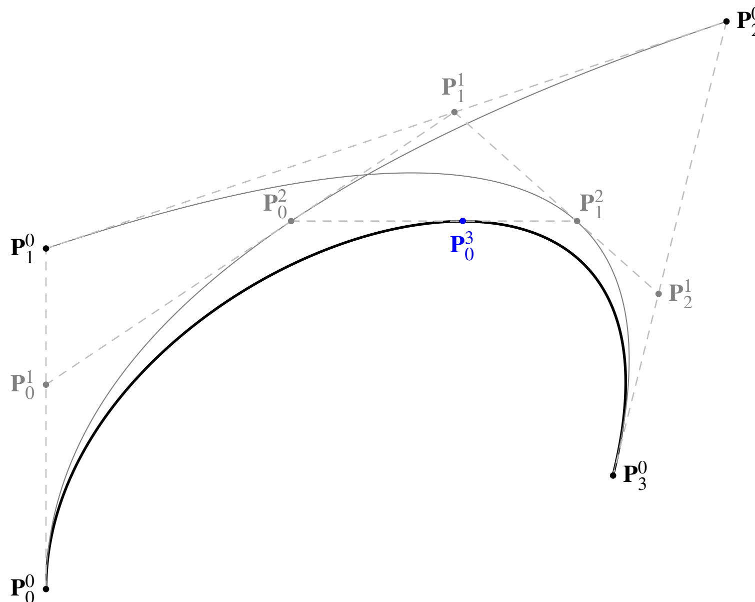

Figure 2.5: Illustration of how to compute point $\mathbf{P}_0^3$ with $t = 0.6$ given Bézier curve $\boldsymbol{B}[\mathbf{P}_0^0,\mathbf{P}_1^0,\mathbf{P}_2^0,\mathbf{P}_3^0]$. All control points and intermediary points in de Casteljau's algorithm are shown. Control points of the curve are drawn in black, while the intermediary points of the algorithm are drawn in gray.

We can now introduce the idea used in de Castaljau's algorithm by means of an example. Say we are given a cubic Bézier curve $\boldsymbol{B}[\mathbf{P}_0^0,\mathbf{P}_1^0,\mathbf{P}_2^0,\mathbf{P}_3^0]$, i.e., one defined by four control points – although the algorithm works for any degree curve. The curve can be seen as the thickest line

in Figure 2.5. We want to compute the value of the curve at $t = 0.6$, which is to say, we want to compute point $\mathbf{P}_0^3$. To that end, we turn to the recursion of Theorem 2.3.1, which tells us that we can do this by computing a linear interpolation between points $\mathbf{P}_0^2$ and $\mathbf{P}_1^2$, since these are the points at $t = 0.6$ on the two light gray quadratic curves $\boldsymbol{B}[\mathbf{P}_0^0, \mathbf{P}_1^0, \mathbf{P}_2^0]$ and $\boldsymbol{B}[\mathbf{P}_1^0, \mathbf{P}_2^0, \mathbf{P}_3^0]$ respectively. Now, since the points $\mathbf{P}_0^2$ and $\mathbf{P}_1^2$ are points on a Bézier curve, they can be computed in turn as a linear interpolation of linear Bézier curves. For example, $\mathbf{P}_0^2$ depends on $\mathbf{P}_0^1$ and $\mathbf{P}_1^1$ and analogously $\mathbf{P}_1^2$ depends on $\mathbf{P}_1^1$ and $\mathbf{P}_2^1$ and so on. The recursion tree will unfold as below, the leaves being linear interpolations of control points.

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

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

$$\mathbf{P}_0^1 = (1-t)\mathbf{P}_0^0 + t\mathbf{P}_1^0 \qquad \mathbf{P}_1^1 = (1-t)\mathbf{P}_1^0 + t\mathbf{P}_2^0 \qquad \mathbf{P}_2^1 = (1-t)\mathbf{P}_2^0 + t\mathbf{P}_3^0$$

Note: The number of equations will be quadratic in the degree of the curve, if we were to generalize to higher degree curves. Dynamic programming can be used to avoid computing the same point multiple times e.g. $\mathbf{P}_1^1$ in the above example.

The correctness of this approach comes down to the proof of Theorem 2.3.1, but to drive home the point, we can take a look at $\mathbf{P}_0^3$ where we plug in the expressions of all intermediary points and simplify giving

$$\mathbf{P}_0^3 = \mathbf{P}_0^0 \cdot (1-t)^3 + \mathbf{P}_1^0 \cdot 3(1-t)^2 t + \mathbf{P}_2^0 \cdot 3(1-t)t^2 + \mathbf{P}_3^0 \cdot t^3 \ ,$$

which is exactly the definition of a point on a cubic Bézier curve as in Equation (2.3)! We can now present de Casteljau's algorithm in its general form as follows.

De Casteljau's algorithm:

Given a Bézier curve $\boldsymbol{B}[\mathbf{P}_0^0, \mathbf{P}_1^0, \dots, \mathbf{P}_n^0]$ and some parameter $t \in [0,1]$, compute

$$\mathbf{P}_i^k = (1-t)\mathbf{P}_i^{k-1} + t\mathbf{P}_{i+1}^{k-1}$$

for $k = 1, \dots, n$ and $i = 0, \dots, k-n$. Then $\mathbf{P}_0^n$ is the point with parameter value t on the Bézier curve $\boldsymbol{B}[\mathbf{P}_0^0, \mathbf{P}_1^0, \dots, \mathbf{P}_n^0]$.

As an added bonus, the intermediary points computed by the algorithm can be used to split the curve. By *splitting the curve* we mean producing two curves that taken together form the same curve as the original. These two curves will be defined by points $\mathbf{P}_0^0, \mathbf{P}_0^1, \dots, \mathbf{P}_0^n$ and $\mathbf{P}_0^n, \mathbf{P}_1^{n-1}, \dots, \mathbf{P}_n^0$ respectively, and have the same degree as the original. Looking at Figure 2.5, it is possible to split the cubic curve at parameter value $t = 0.6$ into two new cubic curves, with the first being defined by $\boldsymbol{B}[\mathbf{P}_0^0, \mathbf{P}_0^1, \mathbf{P}_0^2, \mathbf{P}_0^3]$ and the second with points $\boldsymbol{B}[\mathbf{P}_0^3, \mathbf{P}_1^2, \mathbf{P}_2^1, \mathbf{P}_3^0]$.

Future work: Through the literature we have identified another way of evaluating polynomials, namely *Horner's method*, which is described in [5]. Using this method requires rewriting the Bézier curve from the Bernstein basis into a so-called power basis. Although we have found indications that this can incur some numerical issues, as explained in [6], it is not disputed that Horner's method is optimal in terms of the number of arithmetic operations needed to evaluate the polynomial.

Summary

Summarizing the previous two sections, Bézier curves are parametric curves defined algebraically as a linear combination of *control points* and *Bernstein basis polynomials* as in Equation (2.3). We highlight the most relevant properties and algorithms as follows.

Endpoint interpolation. Given a curve $\boldsymbol{B}[\mathbf{P}_0, \ldots, \mathbf{P}_n](t)$ with $t \in [0,1]$ then the curve will start in $\mathbf{P}_0$ and end in $\mathbf{P}_n$. This property is key to chaining multiple curves together to form a contour line.

Convex hull property. Given a curve $\boldsymbol{B}[\mathbf{P}_0, \ldots, \mathbf{P}_n](t)$ with $t \in [0,1]$ then the curve will always be contained within the convex hull of the control points. This fact was due to the **partition of unity** property of Bernstein polynomials.

De Casteljau's algorithm. An algorithm based on a recursive formula used to evaluate points on a Bézier curve. It provides a numerically stable alternative to evaluating Bernstein polynomials directly.

2.4 Computing Roots of a Curve

Many different geometric problems of Bézier curves are equivalent to finding roots in the parametric functions that define all points along the curve. Examples of these include, but are not limited to: determining if a curve intersects a line segment as in Section 2.7 or computing a tight bounding box around a curve as in Section 2.6.2. In this section, we provide an overview of the methods used to compute the roots of Bézier curves.

Since these parametric functions, in our case, are always cubic polynomials at most, we focus our attention on finding roots of quadratic and cubic polynomials. As Bézier curves are parametric, i.e., they are defined by component functions $x(t)$ and $y(t)$, when we refer to a root of the curve $\boldsymbol{B}_n(t) = (x(t), y(t))$, we are referring to a parametric value $t \in [0,1]$ such that either $x(t) = 0$ **or** $y(t) = 0$. When relevant, we will specify which case we are referring to.

2.4.1 Quadratic curves – the Quadratic formula

Say we have a quadratic curve $\boldsymbol{B}[\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2](t)$ and we want to find its roots. Since the component functions are quadratic polynomials, the roots can be found using the omnipresent *quadratic formula* stating that, if $f(t) = at^2 + bt + c$ then

$$f(t) = 0 \text{ when } t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}\,. \tag{2.4}$$

The formula can be used provided we rewrite the Bézier component functions in standard form. This can be done as follows:

$$\begin{aligned}
\boldsymbol{B}_2(t) &= \mathbf{P}_0(1-t)^2 + \mathbf{P}_1 2(1-t)t + \mathbf{P}_2 t^2 \\
&= \mathbf{P}_0(t^2 - 2t + 1) + \mathbf{P}_1 2(t - t^2) + \mathbf{P}_2 t^2 \\
&= (\mathbf{P}_0 - 2\mathbf{P}_1 + \mathbf{P}_2)t^2 + 2(\mathbf{P}_1 - \mathbf{P}_0)t + \mathbf{P}_0
\end{aligned}$$

giving us the coefficients $a = \mathbf{P}_0 - 2\mathbf{P}_1 + \mathbf{P}_2$, $b = 2(\mathbf{P}_1 - \mathbf{P}_0)$ and $c = \mathbf{P}_0$. We compute the roots of $x(t)$ by plugging the coefficients into Equation (2.4) using only the x-coordinates of the points, disregarding any resulting t not in the unit interval. Roots of $y(t)$ are found in a similar fashion.

2.4.2 Cubic curves – Cardano's method

Say we have a cubic curve $\boldsymbol{B}[\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3](t)$ and we want to find its roots. The component functions of the curve are cubic, and luckily there is a standard algebraic method of finding roots of these. It is called *Cardanos' method* and involves substantially more calculation steps than the quadratic formula, which is to say that we will not go into the mathematical details of it. Our approach has simply been to implement the math as we have seen it described in [36], without drilling into the rationale behind it. As the roots of a cubic polynomial are general complex numbers we disregard any roots that are not real numbers in the unit interval.

Future work: An obvious next step with regards to root finding would be to research and implement numerical methods e.g. *Newton's method*, experimentally evaluate its performance and compare to the algebraic methods described above.

2.5 Continuity of Connected Bézier Curves

We express contour lines as a series of Bézier curves chained together such that each individual Bézier curve ends where the next one begins. In the literature, this construction has several different names: *composite Bézier curve*, *Bézier spline* or *polybézier* to name a few. We will use Bézier spline when referring to them, as it seems to be most common.

When connecting curves in the above mentioned fashion, the concept of *continuity*[32] is used to characterize the transition between two curves. For parametric curves, a distinction between *parametric continuity* and *geometric continuity* is made, with the latter being "less strict" in a mathematical sense.

2.5.1 Parametric Continuity

As the name suggests, parametric continuity imposes some constraints on the parameter of the curves involved. Two curves $\boldsymbol{G}_{[t_0,t_1]}$ and $\boldsymbol{H}_{[t_1,t_2]}$ are said to be C^k continuous (have k-th order parametric continuity) if

$$\boldsymbol{G}(t_1) = \boldsymbol{H}(t_1), \boldsymbol{G}'(t_1) = \boldsymbol{H}'(t_1), \ldots, \boldsymbol{G}^{(k)}(t_1) = \boldsymbol{H}^{(k)}(t_1)\ , \tag{2.5}$$

with $^{(k)}$ meaning the k-th derivative. We can think of the value k as being proportional to how "smooth" the transition between curves should be. Being C^0 continuous only guarantees that they share a common endpoint. Going a step further, C^1 continuity requires that the first derivatives, i.e., the tangent vectors at the common endpoint are equivalent in both direction and magnitude. The problem with using parametric continuity in our case is that it requires the curves to have the same parameter value in their common endpoint for all orders of parametric continuity, which can be seen in from t_1 being used on both sides of the equalities in Equation (2.5).

2.5.2 Geometric Continuity

In our implementation, each curve in a spline is defined with parameter $t \in [0,1]$. This also means that when we "chain" them together into a spline, their parameter value at the common endpoint will be different. Mathematically, if we have two curves $\boldsymbol{Q}_{[0,1]}$ and $\boldsymbol{Z}_{[0,1]}$ then their common endpoint is where $\boldsymbol{Q}(1) = \boldsymbol{Z}(0)$, so we violate already the requirement for C^0. This motivates the use of geometric continuity [8], which is independent of the parameterization of the curves.

With geometric continuity, the objective is to obtain continuity in a more visual sense – the transition from one curve to the next has to *look* smooth. If we denote orders of geometric continuity as G^k and explicitly give the control points of the curves as $\boldsymbol{Q}[\mathbf{V}_0, \ldots, \mathbf{V}_n](t)$ and $\boldsymbol{Z}[\mathbf{U}_0, \ldots, \mathbf{U}_n](t)$, then G^0 continuity requires only that two curves share a common endpoint,

i.e., $\mathbf{V}_n = \mathbf{U}_0$. Establishing G^1 continuity is a matter of ensuring that the curves have a common tangent line in the point where they meet. This turns out to be the case when points $\mathbf{V}_{n-1}, \mathbf{V}_n = \mathbf{U}_0$ and $\mathbf{U}_1$ are collinear, a result that can be found in [8].

2.6 Axis-aligned Bounding Boxes

Axis-aligned rectangles are a fundamental geometry that lend themselves well to being stored in geometric data structures, or to being used to query the data structures. In later section we look at problems such as finding intersection points between two Bézier curves. Bounding boxes of curves can be used in this regard, and as such we present methods for computing axis-aligned bounding boxes (AABB) next.

2.6.1 Convex hull based approach

A naive approach to finding the bounding box of some Bézier curve is to make use of one its fundamental properties, namely the *convex hull property* [32]. Recall that this property guarantees that a Bézier curve $\boldsymbol{B}[\mathbf{P}_0, \ldots, \mathbf{P}_n](t)$ is contained within the convex hull of its control points. In this case, where an axis-aligned bounding rectangle of the curve is desired, it can thus be computed solely from the maximum and minimum coordinate values of the control points on each axis, i.e, the minimum bounding rectangle (MBR) of the control points.

2.6.2 Derivative based approach

The convex hull of the control points of a curve, and thus the AABB found in the above approach, is not *tight* in the sense that it only encompasses the actual curve. In fact, it is easy to fabricate an example, where the naive AABB is quite large relative to a *tight* AABB computed from the parametric curve itself. This is exactly the case illustrated in Figure 2.6.

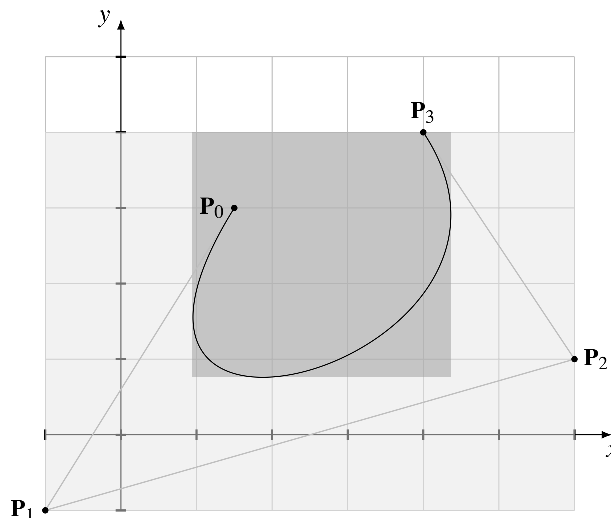

Figure 2.6: A cubic Bézier curve with two different AABBs illustrated. The light gray rectangle being the result of the convex hull approach, while the dark gray rectangle is the result of the derivative based approach. Notice the difference in size.

A tighter axis-aligned bounding box can be computed by looking to the extremities of the parametric functions that together define the curve, i.e., if $\boldsymbol{B}_n(t) = (x(t), y(t))$ we look to the extremities of $x(t)$ and $y(t)$ separately. The x-coordinate of all points along the curve are defined

by $x(t)$. As such, finding roots in the derivative $x'(t)$ gives us the t value where the curve has an extremity, or more visually, where the curve has horizontal or vertical tangents (see Figure 2.3). All such points along the curve are potential candidates for defining the tight AABB. The Bézier curve is only defined for $t \in [0,1]$, so any roots found outside of this interval are discarded. On a high level, we get the following procedure for finding a tight AABB of a Bézier curve $\boldsymbol{B}$:

1. Find all t-values that are roots of $x'(t)$ and $y'(t)$ respectively.
2. Discard all $t \notin [0,1]$, and add the endpoints $t = 0$ and $t = 1$ to the list.
3. For all t_i in the list compute $\boldsymbol{B}(t_i) = (x(t_i), y(t_i))$ and find $\min_i x(t_i)$ and $\max_i x(t_i)$, which are the x-coordinates of the two points defining the AABB. The minimum and maximum y-coordinates are found identically.

In theory, the *most tightly* fitting bounding box of the curve might not be axis-aligned at all. With the above method established, computing it is a matter of (1) axis-aligning the curve itself by means of linear transformations (2) applying the above method.

2.7 Intersection of Curve and Line Segment

The need for determining intersections between Bézier curves and line segments arises from the fact that we represent optimized contour lines with Bézier curves and accurate contours as line segments. Deciding whether a curve and a line segment intersect is useful for deciding if a contour violates a depth constraint. In the following explanation we focus on cubic Bézier curves as these are used for the optimization, but the same approach generalizes to curves of higher order.

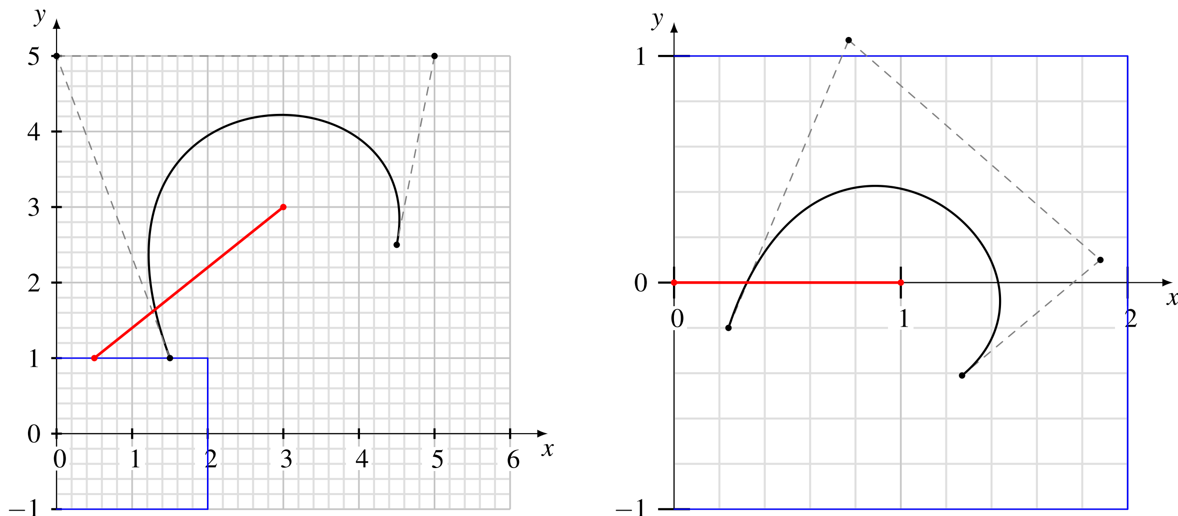

Figure 2.7: Visualization of axis alignment. The blue square encloses the same region in both plots. **Left**: A Bézier curve with a line segment crossing it. **Right**: The curve and line segment after translation, rotation and scaling.

Given a Bézier curve $\boldsymbol{B}[\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3](t)$ and a line segment $\ell = [\boldsymbol{p}_0, \boldsymbol{p}_1]$, it is possible to compute whether the two have an intersection point. Both geometries can be positioned arbitrarily in the real plane $\mathbb{R}^2$. The problem can be simplified to root finding by first applying the same linear operations to $\boldsymbol{B}$ and ℓ such that ℓ coincides with the x-axis. This transformation can be performed in multiple ways, and we did it in the following way (as illustrated in Figure 2.7):

First, a translation by $-\boldsymbol{p}_0$ is performed on all points making $\ell = [0, \boldsymbol{p} = \boldsymbol{p}_1 - \boldsymbol{p}_0]$, where 0 is the 2-dimensional zero vector. As ℓ has a point in the origin we simply need to rotate it down on the x-axis. The matrix

$$R = \begin{bmatrix} \boldsymbol{p}^x & -\boldsymbol{p}^y \\ \boldsymbol{p}^y & \boldsymbol{p}^x \end{bmatrix}$$

is a matrix of orthogonal column vectors satisfying

$$R \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \boldsymbol{p} .$$

As columns of R are orthogonal, R can be perceived as a scaling and rotating matrix. As a result, applying R on vectors defining a Bézier curve produces a scaled and rotated variant of the previous Bézier curve. As such, R^{-1} can be used as our final transformation to rotate $\boldsymbol{B}$ and ℓ such that ℓ lies in the $0-1$ interval of the x-axis. The intersections point(s), if there are any, are now the roots of the y-component, $y(t)$, provided that these are found within $t \in [0,1]$ and $x(t) \in [0,1]$. An illustration of this transformation is shown in Figure 2.7. Computing R^{-1} is simple: Each column of R are vectors of magnitude $||\boldsymbol{p}||$, thus the matrix $R/||\boldsymbol{p}||$ is an orthogonal matrix, meaning

$$\frac{R}{||\boldsymbol{p}||} \left(\frac{R}{||\boldsymbol{p}||} \right)^T = \frac{1}{||\boldsymbol{p}||^2} RR^T = I.$$

The equation tell us that the inverse matrix of an orthogonal matrix is equivalent to the transpose matrix. This provides the result

$$R^{-1} = \frac{1}{||\boldsymbol{p}||^2} R^T .$$

The reason that we can translate, rotate and scale a Bézier curve and be sure that its shape is retained is due to a property of Bézier curves called *affine invariance*. We only state this as a fact; the definition affine maps and a proof that Bézier curves are affine invariant can be found in the textbook by G. E. Farin [10].

2.8 Intersection of Curve and Curve

A natural constraint when dealing with contour lines on any map is, that they cannot be allowed to cross each other, thereby creating overlapping regions. Such an overlap would signify two different water depths within same physical region, which is impossible. Since we represent contour lines as Bézier curves, detecting such a violation requires a method to finding intersections between two curves.

For an overview and performance comparisons of curve-curve intersection algorithms we refer to [32]. Below we present the most basic method, which is based on a general technique called *subdivision*.

2.8.1 Subdivision method

The convex hull subdivision approach [22] relies, as the name suggests, on the property that a Bézier curve is completely contained within the convex hull of the control points of the curve. Only if the convex hulls of two curves overlap is it possible that the curves themselves do so as well. The subdivision algorithm uses this fact recursively: Given two curves, compute their hull and check if they overlap. If they do, the curves are subdivided (split) at their halfway, and the procedure is called recursively on the subdivided curves. As the recursion proceeds, non-overlapping hulls are discarded. When the overlapping convex hulls are sufficiently small in some well-defined way, the algorithm concludes that it has found an intersection.

The only concrete stopping criteria we have come across is mentioned by Sederberg [32] with reference to Wang [38]: The subdivision should terminate once the curve has been subdivided enough times that it approximates a straight line segment to within a specified tolerance ε. A closed formula for computing the number of subdivision necessary for this criteria to hold can be found in [32, Equation 10.4].

To increase performance, it is possible to use the axis-aligned bounding boxes of the curves instead of their convex hulls. This optimization is motivated by the fact that convex hulls are more expensive to compute *and* determine overlaps for, compared to bounding boxes. Intuitively the procedure is still correct, as the convex hull is fully contained within the bounding box, and the convergence argument thus still holds.

Future work: Several approaches to the curve-curve intersection problem are described in the literature, and we have only presented the most basic. A natural next step would be to implement and experimentally evaluate more advanced techniques such as interval subdivision [20], Bézier clipping [34] or implicitization [33]. Furthermore, interesting hybrids of these exist, namely the cocktail algorithm [17] and hybrid clipping [24].

2.9 Interior Point Problem

Although curve-line intersection as presented in Section 2.7 is useful for checking some type of constraints, it is not sufficient. Datasets can contain small spikes in elevation that an optimization step might jump entirely, which is illustrated in Figure 2.8: This is a clear constraint violation with no direct intersections occurring. Thus, an additional method is required to ensure no constraints

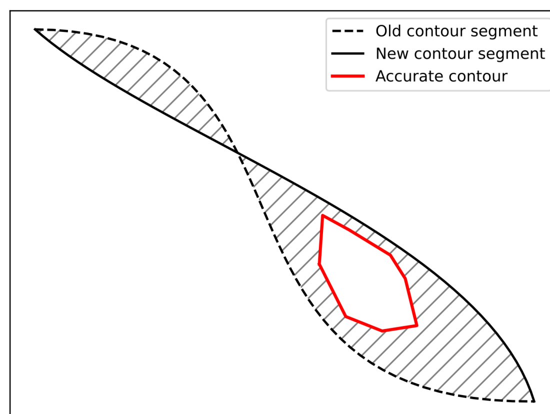

Figure 2.8: An optimization step updating an old contour segment to a new contour segment. The update crosses an accurate contour line without intersecting it.

are inside the hatched area on the figure. Our method for doing this combines intersection checking (either curve-line or curve-curve intersection) with a simpler problem we describe shortly. In order to prove this indeed provides the correct result, a more formal definition of the problem is required. We define the problem as follows:

Problem 2.9.1 (Interior Line Segment) *Given two Bézier splines* $\mathbf{S}$ *and* $\mathbf{S}'$ *with the same start- and endpoints and a line segment* $\ell = [\boldsymbol{p_0}, \boldsymbol{p_1}]$ *for points* $\boldsymbol{p_0}, \boldsymbol{p_1} \in \mathbb{R}^2$*. Let* P *be the set of all points that lie in the interior of the area enclosed by* $\mathbf{S}$ *and* $\mathbf{S}'$*. Decide if* $\exists \boldsymbol{p} : \boldsymbol{p} \in P \land \boldsymbol{p} \in \ell$*.*

Consider also the decision problem with a point instead of a line segment:

Problem 2.9.2 (Interior Point) *Given two Bézier splines* $\mathbf{S}$ *and* $\mathbf{S}'$ *with the same start- and endpoints and a point* $p \in \mathbb{R}^2$*. Let* P *be the set of all points that lies in the interior of the area enclosed by* $\mathbf{S}$ *and* $\mathbf{S}'$*. Decide if* $p \in P$*.*

Problem 2.9.2 is simpler than Problem 2.9.1 and we will shortly show that it is sufficient for our problem. Additionally it generalizes better to the case of checking if a Bézier curve is swallowed by an optimization step.

In Section 2.7 we saw how to determine if a line segment and a Bézier curve intersect. By performing curve-line intersection first we only need to consider the cases with no intersection, i.e., as on Figure 2.8. The idea is that with no intersections, all points will either lie inside or outside of the enclosed area P. This follows closely from the *Jordan curve theorem* [15, 37], which can be seen below as Theorem 2.9.1. The theorem concerns so-called *simple curves*, which defined as a continuous curve with no self-intersections:

Theorem 2.9.1 (Plane Sub-Division) *Every simple closed curve decomposes its plane into two open regions. An interior and exterior region.*

Theorem 2.9.2 (Curve Crossing) *Any simple curve joining an interior point of a region to an exterior point must cross the boundary in a point or a pair of points.*

The proofs are quite involved and out of the scope of this thesis but can be found in [37]. With these theorems in mind, the following can be proven.

Theorem 2.9.3 (Problem Equality) *Given two splines* $\mathbf{S}$ *and* $\mathbf{S}'$*, a line segment* $\ell = [\boldsymbol{p_0}, \boldsymbol{p_1}] : \boldsymbol{p_0}, \boldsymbol{p_1} \in \mathbb{R}^2$ *and an arbitrary point* $\boldsymbol{p} \in \ell$*. Assuming* ℓ *is not intersecting* $\mathbf{S}$ *or* $\mathbf{S}'$ *then Problem 2.9.1 and Problem 2.9.2 are equivalent.*

In the following, we use C to denote the curve created by joining $\mathbf{S}$ and $\mathbf{S}'$ at their endpoints. Let P be the interior region enclosed by C. To prove the theorem we need the assumption that ℓ does not intersect C. In this case, we want to argue that all points on ℓ are either all in the interior of P or in the exterior of P. Since $\mathbf{S}$ and $\mathbf{S}'$ are continuous and share endpoints, C is a continuous closed curve. Unfortunately, C is not guaranteed to be simple as it may self-intersect $n \geq 0$ times. Instead, consider the disjoint curves $C_0, ..., C_n$ achieved by splitting C at intersection points into multiple disjoint curves. All C_i for $i \in \{0, ..., n\}$ do not self-intersect as they otherwise would have been split. All C_i are still closed and continuous as they are connected at intersection points. They are thus simple closed curves. By assumption, there exists no point $\boldsymbol{p} \in \ell$ such that $\boldsymbol{p}$ is on the boundary of C. Further, no point is on the boundary of any C_i as they are constructed from disjoint segments of C. From Theorem 2.9.1 all C_i have a well-defined interior and exterior. Assuming a $\boldsymbol{p} \in \ell$ is in the interior of C_i and $\boldsymbol{p'} \in \ell$ the exterior of C_i contradicts with Theorem 2.9.2. As such all points on ℓ must be reside in either the interior or exterior. Let $P_i \subseteq P$ be the interior of C_i and denote $\bigcup_i P_i = P$, which means a point is in some P_i if and only if it is in P. Thus the line segment ℓ has to reside either completely in the interior of P or in the exterior of P. Considering just a single point on $\boldsymbol{p} \in \ell$ is a special case of the above, where an arbitrary point on the line segment is used and as such it is equivalent.

We consider now how to solve Problem 2.9.2. One simple approach would be constructing a polygon from C and use an interior point algorithm for polygons such as the *Ray Casting Algorithm* [35]. The algorithm can pictorially be described by casting a ray parallel to the x-axis out of the point in a direction as illustrated on Figure 2.9. Line segments of the polygon are now traversed and intersections with the ray are counted. If the number of intersections is odd the point is in the interior and otherwise in the exterior. The intuition for why this algorithm is correct lies in Theorem 2.9.2. Crossing the boundary of a polygon goes from exterior to interior

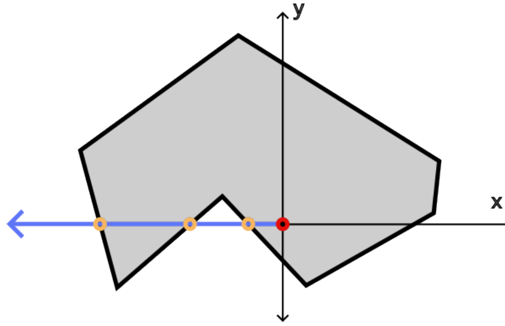

Figure 2.9: An illustration of the Ray Casting Algorithm. The red circle is the target point, and orange circles indicate intersection points

and vise versa. Having finite polygons the ray will eventually end up in the exterior. Every time it crosses the border the state is flipped, thus an odd number means the point in the interior.

A problem with this approach lies in the fact that Bézier curves are, as the name suggests, curves, and thus they do not form a polygon. They can however be approximated arbitrarily well with polygons by splitting them into many shorter line segments. Unfortunately, more splits decrease the numeric precision and slow down the point-in-polygon algorithm, as its running time is linear in the amount of line segments.

Instead, we tweak the algorithm to handle curves: Given a query for a point $\boldsymbol{p}$ on the curve C, apply a translation to the point and the curve such that $\boldsymbol{p}$ becomes the origin. By directing the ray towards negative x values, so it resides on the negative side of the x-axis. Computing intersection points is thus equivalent to finding roots with negative x value. C consists of cubic Bézier curves and computing roots of these are covered in Section 2.4.2.

As a last note consider a similar problem to Problem 2.9.1, where we consider a curve in the interior instead of a line segment:

Problem 2.9.3 (Interior Curve) *Given two Bézier splines* $\mathbf{S}$ *and* $\mathbf{S}'$ *with the same start- and endpoints and a simple curve D consisting of points* $\mathbb{R}^2$. *Let P be the set of all points that lie in the interior of the curve enclosed by* $\mathbf{S}$ *and* $\mathbf{S}'$. *Decide if* $\exists \boldsymbol{p} : \boldsymbol{p} \in P \wedge \boldsymbol{p} \in D$.

This problem can be solved with the same procedure as for a line segment if the intersection algorithm is substituted with a curve-curve intersection algorithm. The reason is that Theorem 2.9.2, as it states, is true for any simple curve and not just line segments.

2.10 Curvature

The purpose of introducing the curvature metric is to get a somewhat objective way of steering the optimization algorithm in a direction that produces "smooth" or "foreseeable" curves. In terms of curvature, this means that we would like to minimize the maximum curvature along the curve.

The curvature of a curve, usually denoted κ, is a measure of how *sharply* the curve moves in space. A perfect circle with radius ρ has a curvature of $\kappa = 1/\rho$. A straight line by definition has a curvature of zero. In the case of Bézier curves, the curvature is generally different for each point on the curve. As such, the curvature of some point on a curve is defined as the reciprocal radius of the *osculating circle* of that point on the curve. Figure 2.10 illustrates this circle. Informally, the osculating circle is the circle that "best fits" the curvature of the curve in the point. A more formal definition of the curvature uses the concept of the *multiplicity* of the intersection between a curve and a circle, which is further covered in [32]. For our purpose, the

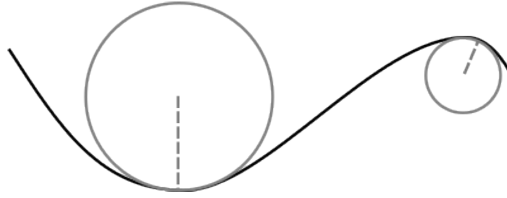

Figure 2.10: A sketch of the osculating circles of two points on a curve

curvature of $\boldsymbol{B}_n(t) = (x(t), y(t))$ can be computed from the first and second derivatives as:

$$\kappa(t) = \frac{|x'(t)y''(t) - y'(t)x''(t)|}{((x'(t))^2 + (y'(t))^2)^{3/2}}. \tag{2.6}$$

While the curvature is easily computed at some specific t, we have not found an analytical way of determining the maximum or minimum curvature along a curve. Thus we turn to the numerical approximation method explained in the following.

2.10.1 Ternary Search Method for Maximum Curvature

A naive approach to determining the maximum curvature is to compute the curvature of a bunch of points sampled evenly distributed along the curve. Unfortunately, when dealing with cubic Bézier curves, curvature extremes can be very localized. However, speaking very informally, the degree of the curve limits how erratic its shape can be, i.e., how many sharp twists and turns it can take. This is a result of the curvature formula in Equation (2.6) which is a third-degree polynomial divided by a fourth-degree polynomial. Ignoring the absolute operation in the numerator, the derivative of Equation (2.6) is a fifth-degree polynomial, thus the curvature has at most 5 extreme values. Maximizing an absolute formula is the same as finding the minimum and maximum of the same function without the absolute operator. The amount of minimum and maximum values are bounded by the number of roots in the derivative, thus the curvature formula has at most 5 extreme values.

The idea is to sample x points along the curve and run a ternary search between each point to find the maximum curvature. Ternary search is only guaranteed to find the maximum if the interval contains only one extreme point. When x gets larger the chance of having two extreme values in the space between two sample points gets smaller. By choosing the maximum value among each ternary search, the maximum curvature is approximated.

While we do not provide any theoretical guarantees with respect to this approach, we have tried to evaluate it experimentally as follows: We ran the ternary search approach with $x = 10$ to compute a maximum curvature on multiple random curves. This result was then compared with the curvature at 10 000 evenly spread points along the curve. The ternary search approach always returns the highest curvature. This shows that $x = 10$ provides reasonable results however, it might not be the most efficient approach. For a better understanding of what the most efficient x would be further experiments are required. This is left as future work.

2.11 Computing Area

As described in Chapter 1 our optimization must consider the deep area of contour lines. As mentioned in Section 1.2 this is the area of the starboard region. This area should naturally be as large as possible, thus ways of computing areas of Bézier curves are required. To be specific, we are interested in computing the area inside the shape that arises when closing the curve using a line segment connecting the two endpoints of the curve, as is illustrated in Figure 2.12.

As Bézier curves are parametric functions as described in Section 2.2, one can compute the area under the curve using integration. The idea follows from integration of non-parametric functions. Given a function C the area under the graph $y = C(x)$ is determined by

$$\int C(x)dx \, . \tag{2.7}$$

Recalling that the parametric definition of a cubic Bézier curves as $\boldsymbol{B}[\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3](t) = (x(t), y(t))$, its x and y position at some t is determined by the component functions $x(t)$ and $y(t)$. By substituting the values x and y from Equation (2.7) with the component functions of the parametric definition, we get $y(t) = C(x(t))$ as a function of t. Using the *chain rule*

$$\frac{dx}{dy} = \frac{dx}{dt}\frac{dt}{dy}$$

which is equivalent to

$$dx = \frac{dx}{dt}dt \, ,$$

Equation (2.7) can be rewritten as an integral over t as

$$\int y(t)x'(t)dt \, . \tag{2.8}$$

In our context, Bézier curves are always defined in the interval $t \in [0,1]$, thus the last step is to compute the integral in this interval. The area computed with this integral is illustrated as a dark gray shade on Figure 2.11. Equation (2.8) quite intuitively reveals why this is the acquired area. When $x(t)$ is increasing, $x'(t)$ is positive, and so the contribution of $y(t)$ to the integral is positive. When the curve changes direction, $x(t)$ is decreasing and $x'(t)$ is negative. As such, the contribution of $y(t)$ is instead "subtracted" from the integral, seen as the light-gray hatched area on Figure 2.11.

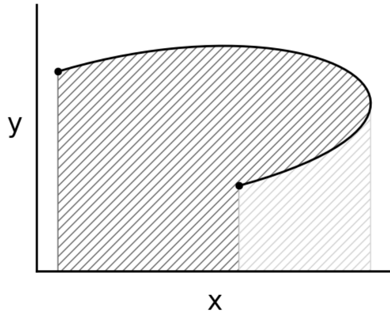

Figure 2.11: The area computed by integration is shown in dark gray.

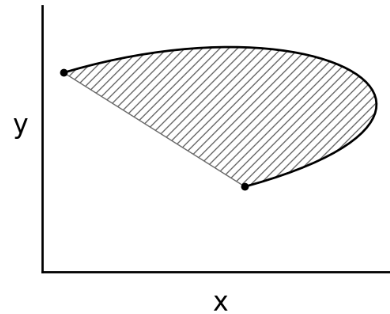

Figure 2.12: The desired area for a curve.

Evaluating the definite integral of Equation (2.8) is indeed possible, as $x(t)$ and $y(t)$ are both third-degree polynomials. Further, the product $y(t)x'(t)$ is a fifth-degree polynomial (in the context of cubic Bézier curves) for which known integration methods exist. The calculation is, according to [14], quite extensive and tedious, so we present only the simplified formula from the paper. We use the notation $\mathbf{P}_i^x$ and $\mathbf{P}_i^y$ to refer to the x and y values of the i-th control point on

the curve, respectively:

$$\begin{aligned}20\int_0^1 y(t)x'(t)dt &= (\mathbf{P}_1^x - \mathbf{P}_0^x)(10\mathbf{P}_0^y + 6\mathbf{P}_1^y + 3\mathbf{P}_2^y + \mathbf{P}_3^y) \\ &+ (\mathbf{P}_2^x - \mathbf{P}_1^x)(4\mathbf{P}_0^y + 6\mathbf{P}_1^y + 6\mathbf{P}_2^y + 4\mathbf{P}_3^y) \\ &+ (\mathbf{P}_3^x - \mathbf{P}_2^x)(\mathbf{P}_0^y + 3\mathbf{P}_1^y + 6\mathbf{P}_2^y + 10\mathbf{P}_3^y)\ ,\end{aligned}$$

Note that the factor of 20 is only there to simplify the expression. To compute the desired integral, the right-hand side of the equation must be divided by 20.

Now, as Figure 2.11 illustrates, the area we get from Equation (2.8) is not the area shown on Figure 2.12, which is what we are really after. In the case where the endpoints of the Bézier curve coincide with the x-axis, the areas will in fact be equal. However, since curves are not guaranteed to be aligned with the x-axis in the coordinate system of the dataset (in fact, most of them are definitely *not*), we have to take into account the trapezoid formed by connecting the endpoints of the curve with the x-axis. Fortunately computing the area of such a trapezoid is simple. Recall the endpoints of the cubic curve are $\mathbf{P}_0$ and $\mathbf{P}_3$, the area is computed by

$$\frac{(\mathbf{P}_3^x - \mathbf{P}_0^x)(\mathbf{P}_3^y + \mathbf{P}_0^y)}{2}\ , \tag{2.9}$$

and can then be subtracted from the integral to achieve the desired area as shown on Figure 2.12. In Equation (2.9) the term $(\mathbf{P}_3^x - \mathbf{P}_0^x)$ is the width of the trapezoid and $(\mathbf{P}_3^y + \mathbf{P}_0^y)/2$ is the average height, thus this formula is simply the area of a rectangle with height equally between the two side length of the trapezoid.

Combining Equation (2.8) and Equation (2.9) we can express the area of a given Bézier curve $\boldsymbol{B}$ as follows:

$$Area(\boldsymbol{B}) = \int_0^1 y(t)x'(t)dt - \frac{(\mathbf{P}_3^x - \mathbf{P}_0^x)(\mathbf{P}_3^y + \mathbf{P}_0^y)}{2}\ .$$

Chapter 3

Contours as an Optimization Problem

In this section, we return to the optimization problem described briefly in Chapter 1 and give a formal definition of it. The aim is to describe how the problem should be modeled as an optimization problem: A formal definition of our aim follows in Section 3.1. From there, we discuss first how a Bézier spline is best represented in the optimization and how the representation is then used to define problem solutions, which happens in Sections 3.2 and 3.3. We then describe the function we use to assess solutions, i.e., the objective function of the problem in Section 3.5. We complete the formulation by defining the constraints of the problem in Section 3.6 and sum it all up in Section 3.7. To better motivate the methods we have studied in order to solve the optimization problem, we finish this section with a small analysis of the problem in Section 3.8.

3.1 Optimization in Standard Form

Using common terminology, we want to define an *objective function* $f : \mathbb{R}^m \to \mathbb{R}$ taking inputs $X \in \mathbb{R}^m$ subject to *constraints* defined by additional functions $c_i : \mathbb{R}^m \to \mathbb{R}$ for $i = 0, ..., n$. Using this notation, an optimization problem in *standard form* is described by

$$\begin{array}{ll} \underset{X \in \mathbb{R}^m}{\text{minimize}} & f(X) \\ \text{subject to} & \begin{array}{ll} c_i(X) = 0, & i \in \gamma \\ c_i(X) \leq 0, & i \in \theta \end{array} \end{array} , \tag{3.1}$$

where γ and θ are indices of equality and inequality constraints. All $X \in \mathbb{R}^m$ are considered a *solution*. However, not all solutions are *feasible*. Only a solution $X \in \mathbb{R}^m$ satisfying all constraints is considered a feasible solution. The goal is typically to find an optimal feasible solution. That is, find an X^* satisfying the constraints such that $f(X^*) \leq f(X)$ for all feasible $X \in \mathbb{R}^m$.

We want to model the optimization problem in standard form, as optimization techniques for such problems are well-studied and it provides a precise description of the problem. In the following sections, we describe how the problem of optimizing contour lines can be modeled in standard form. This includes constructing a suitable objective function f, representing contours as a vector $X \in \mathbb{R}^m$ and mathematically defining constraints on the contours.

Note on terminology: In the above $f(X)$ is referred to as the *objective function*, which is common terminology in mathematical optimization. Another term for $f(X)$ that we use is *loss function*, the reason being that this term more explicitly captures our intent to minimize the value of $f(X)$. With that being said, the reader can treat the terms as synonyms.

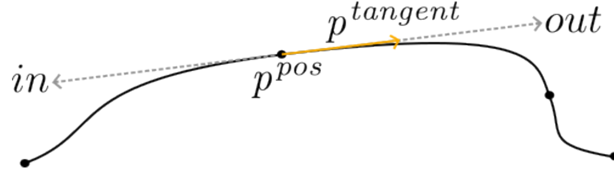

Figure 3.1: An illustration of a spline point. Each spline point has a position (p^{pos}), tangent ($p^{tangent}$) and scalars (out, in)

3.2 Bézier Spline Representation

As both objective f and constraints c_i depend on the solution X, we first consider how X should represent contour lines. To represent contour lines of a solution we use Bézier splines, as defined in Section 2.5, which in turn consist of multiple connected Bézier curves. There are multiple ways to represent this mathematically. We present two:

Representation 1: One obvious method is representing a Bézier spline as a list of Bézier curves. We exclusively use cubic Bézier curves for contours. Cubic Bézier curves are defined by four points in $\mathbb{R}^2$, thus one Bézier curve can be represented by a vector in $\mathbb{R}^8$. A Bézier spline of n Bézier curves can thus be represented as a vector of size $8n$. One problem with this approach is that extra measures are required to ensure the geometric continuity of the spline as discussed in Section 2.5. Recall that, to have geometric continuity requires two Bézier curves have a common endpoint, as well as a common tangent direction at this point. Since the vector representation of a Bézier spline uses two distinct points to model the endpoints of adjacent curves, extra constraints are needed to ensure that these points are equal and share tangent direction. For this reason, we use another representation, which merges the common endpoints by default.

Representation 2: In this representation, the spline is expressed by what we call *spline points* instead of all the control points of the Bézier curves. A *spline point* contains five pieces of information: The coordinates of where two curves meet, a tangent vector at that point, and a the length of the tangent vectors in each direction. Mathematically, a spline point is a 5-tuple

$$p = (pos_x, pos_y, angle, out, in)$$

with the following named fields: The position is a coordinate pair $p^{pos} = (pos_x, pos_y)$, the tangent vector $p^{tangent} = (\cos(angle), \sin(angle))$, and the length of the tangent going out (out) and in (in) at the point. Figure 3.1 illustrates this representation. Given two spline points p_0 and p_1 in this representation, the cubic Bézier curve $\boldsymbol{B}[\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3](t)$ between them has control points

$$\begin{aligned}
\mathbf{P}_0 &= p_0^{pos} \\
\mathbf{P}_1 &= p_0^{pos} + p_0^{tangent} \cdot p^{out} \\
\mathbf{P}_2 &= p_1^{pos} - p_0^{tangent} \cdot p^{in} \\
\mathbf{P}_3 &= p_1^{pos}.
\end{aligned} \tag{3.2}$$

Thus it is easy to convert between the two representations. A Bézier spline of n Bézier curves is represented by $n+1$ spline points. Note that this representation enforces geometric continuity automatically. When changing p^{pos} or $p^{tangent}$ for some spline point p, both the next and previous Bézier curve is affected, as p controls the endpoint and tangent of both curves. A spline is denoted $\mathbf{S} = (p_0, p_1, \ldots, p_n)$ where each p_i is a spline point. This invites for the notation $p \in \mathbf{S}$ meaning p is a spline point in $\mathbf{S}$. However, often the Bézier curves are needed and not just the spline points. Thus we also introduce the slightly abused notation of $\boldsymbol{B} \in \mathbf{S}$ to indicate a Bézier curve $\boldsymbol{B}$ of the spline $\mathbf{S}$. This corresponds to constructing the Bézier curve defined by two consecutive spline points using Equation (3.2).

3.3 Solution Representation

Using the spline point representation for Bézier splines simplifies the task of defining the set of feasible solutions. A solution is simply a sequence of Bézier splines of some fixed size determined by the initial solution. Given an initial solution $\overline{\boldsymbol{X}} = (\overline{\mathbf{S}_i} : i \in \{0,\dots,n-1\})$ where each spline $\overline{\mathbf{S}_i}$ has $|\overline{\mathbf{S}_i}|$ spline points, then a solution $\boldsymbol{X} = (\mathbf{S}_i : i \in \{0,\dots,n-1\})$ must also have n Bézier splines each of size $|\overline{\mathbf{S}_i}|$ respectively. This is required to keep the number of optimization variables constant which is required from the standard form Equation (3.1). Note that the standard form requires a vector $X \in \mathbb{R}^m$, and as we represent each spline $\mathbf{S}_i$ in the solution using the spline point representation from the previous section, the total dimension is

$$m = 5 \sum_{i=0}^{n-1} |\overline{\mathbf{S}_i}|. \tag{3.3}$$

3.4 Constants

Our optimization problem has certain constants used in the loss function and constraints. These are not optimized but used to ensure that everything behaves correctly. These include constants such as the accurate contour lines and their depth, the Bézier spline depths, the target curvature and area score. In particular the target curvature and area score are defined from the *chart scale* that the chart should be rendered in. As such, we dedicate a short subsection to explain the chart scale.

3.4.1 Defining the Chart Scale

As it is very impractical to display geographical in a 1-to-1 manner, it is natural to scale data by some amount, before displaying it. The scaling factor called the *chart scale*, is different with each dataset, and is written as $x : y$ meaning x units on the chart correspond to y units in the dataset. D1 has a chart scale of 1 : 50000 while D2 and D3 have a chart scale of 1 : 25000. Using the chart scale allows for conversion between the chart and data. As an example, consider measuring one millimeter on a chart with chart scale 1 : 50000. This corresponds to 50000mm = 50m in the dataset. To summarize if two data points have a distance of 50, they will be one millimeter apart on the chart with a chart scale of 1 : 50000.

3.4.2 Accurate Contours

Firstly the accurate contours are important constants as they are used to ensure depth constraint are satisfied which which is further elaborated in Section 3.6.1. We denote the set of all accurate contours by $\boldsymbol{C} = \{\boldsymbol{A}_i : i \in \{0,\dots,k\}\}$, where each $\boldsymbol{A}_i$ is a accurate contour line. The accurate contours are represented with a height h and a poly-line p of connected line segments. As such, we will refer to the height of an accurate contour $\boldsymbol{A}_i^h$, the polyline $\boldsymbol{A}_i^p$ and line segments $\ell \in \boldsymbol{A}_i^p$. The height is a metric of distance above the water surface, thus for bathymetric contours these values are negative. When we write $\boldsymbol{A}_i^h < \boldsymbol{A}_j^h$ it means that the i-th contour is lower than the j-th. A optimized contour $\mathbf{S}$ is likewise associated with a constant height which we denote by $\mathbf{S}^h$. Note that as the height is a constant value it is not directly included in the solution definition seen in Section 3.3.

3.4.3 Target Curvature and Area Score

The loss function which we introduce shortly requires two additional constants. A target curvature $\kappa' = \frac{1}{r'}$ and an area score A. The target curvature is the max curvature we desire and the area score is a relation between area to loss. The following values for these constants are chosen by

Geodatastyrelsen based on their experience. The target curvature is $\frac{1}{5\text{mm}}$ in chart scale equivalent to $r' = 5$mm in chart scale. The area scale is 2cm^2 in the chart scale, meaning 2cm^2 on the chart corresponds to 1 loss. The datasets in this project are represented in meters and have different chart scales. Thus, for a dataset of chart scale $1 : x$,

$$r' = \frac{5}{1000}x,\ A = \frac{2}{10000}x^2 \ .$$

3.5 Defining the Loss Function

Before any optimization can be done a way of comparing the different solutions is needed. For this, we will define a loss function which is a function that given a solution returns a floating point number which we call *loss*. A solution with lower loss will be considered a better solution, thus the loss function can be used to compare solutions. The loss function has to capture properties described in Chapter 1 in order for the optimized solution to reflect the wanted properties such as low detail and large sailable area. In the following sections, we describe our building blocks for constructing a suitable loss function.

3.5.1 Metric: Curvature

The first property the loss function should capture is the curvature. This includes the following desired properties:

1. The sharper the turns the higher the loss. Exceeding κ' by a small amount should be insignificant compared to large deviations.
2. The curvature should be as evenly distributed as possible for curves to appear more consistent. This means it is preferable to have a slightly too large curvature on a longer section of the curve than to have a large curvature in a small section.
3. All Bézier curves with curvature lower than some given target $\kappa' = \frac{1}{r'}$, result in constant loss. The reason for this is to discourage the algorithm from creating straight lines when a smooth curve of acceptable curvature fits. This means that a straight line and curve with target curvature, as well as everything in between, gets the same loss.

We recall Equation (2.6) defining curvature as seen earlier in Section 2.10 as

$$\kappa(t) = \frac{|x'(t)y''(t) - y'(t)x''(t)|}{\left(x'(t)^2 + y'(t)^2\right)^{3/2}} \ .$$

To see how the curvature formula naturally captures property 1, we examine it more closely: The numerator is the absolute cross-product of the first and second derivative and the denominator is the Euclidean distance of the derivatives raised to the third power. The derivative is the tangent of the Bézier curve at some point t while the double derivative is the change in tangent. Sharp turns correspond to large changes in the tangent, which in turn means large values of the second derivative and thus a large numerator. If the length of the tangent vector is large, then the denominator is also a large value, which reduces the curvature. This makes perfect sense as large adjustments are not as severe if the tangent has a large magnitude itself.

To handle property 2 we want to minimize the maximum curvature for each Bézier segment. We define the max curvature along a Bézier curve as

$$Curve_{max}(\boldsymbol{B}) = \max(\{\kappa_{\boldsymbol{B}}(t) : t \in [0,1]\}).$$

To handle property 3 the final formula for curvature loss of a Bézier curve is

$$Curve(\boldsymbol{B}) = \max(Curve_{max}(\boldsymbol{B}) \cdot r', 1). \tag{3.4}$$

This equation is easiest understood using curvature defined from the osculating circle. Note that if $r \geq r'$, meaning the radius r of the smallest osculating circle along $\boldsymbol{B}$ is larger than the one of the target, then $Curve(\boldsymbol{B}) = 1$. Thus all curves with lower max curvature than κ' will result in the same loss. It is no problem that the loss of a 'perfect' Bézier curve is 1 as opposed to 0 as the loss function is only used for comparisons.

3.5.2 Metric: Area

The curvature is important to achieve simplicity, however, it puts no requirements on the similarity of the optimized and accurate contours. It is preferable that the optimized contour maintains a degree of fidelity to the original, requiring another metric. For this purpose, we employ an area metric to maximize the area on the deeper side of the simplified curve. As described in Chapter 1 the optimized contour must lie entirely in the deeper region of the original, implying that the area on the deeper side of the optimized contour is less than that of the original. For this reason, equality in the area can only be achieved if the two contours are identical. Consequently, it seems reasonable to maximize the deep area in the loss function, as maximizing deep area without violating the depth constraint must mean maximizing similarity between the accurate contour and the optimized contour. Furthermore, maximizing this area also allows for more space for ships to maneuver when navigating according to the optimized contour map.

Instead of computing the whole area enclosed by the optimized contour, we simplify the computation by computing the area difference between the initial contour and our optimized contour. Both are suitable as the loss function is only used for comparisons.

We recall the notation developed in Section 3.3: any solution $\boldsymbol{X}$ consists of splines $\mathbf{S}$ and the initial solution denoted $\overline{\boldsymbol{X}}$ consists of splines $\overline{\mathbf{S}}$. Now, consider connecting all the endpoints of the Bézier curves of the initial solution. This forms a polygon represented as a sequence of line segments, which we can denote

$$L = \Big([\mathbf{P}_0, \mathbf{P}_3] : \boldsymbol{B}[\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3](t) \in \overline{\mathbf{S}} \Big) \ .$$

This polygon can then serve as a baseline for the area computation. With it, the global area gain of a contour can be defined as the difference in area between the initial polygon L and the contours computed by our algorithm. This area is illustrated as the hatched region in Figure 3.2a. Note that each line segment in L corresponds to a Bézier curve in $\overline{\mathbf{S}}$. Furthermore, there is one-to-one correspondence between the Bézier curves in $\mathbf{S}$ and $\overline{\mathbf{S}}$ as each $\boldsymbol{B} \in \mathbf{S}$ is just a modification of $\overline{\boldsymbol{B}} \in \overline{\mathbf{S}}$ as described in Section 3.2. Thus a $\boldsymbol{B} \in \mathbf{S}$ has a unique line segment $\ell \in L$. We denote by $\ell^{\boldsymbol{B}}$ the line segment corresponding to $\boldsymbol{B}$. For each Bézier curve $\boldsymbol{B}$ in $\mathbf{S}$, it is possible to compute the area 'between' $\boldsymbol{B}$ and $\ell^{\boldsymbol{B}}$ and we denote it $Area(\boldsymbol{B}, \ell^{\boldsymbol{B}})$. As illustrated on Figure 3.2b, we define 'between' as the region enclosed by connecting endpoints of $\boldsymbol{B}$ and $\ell^{\boldsymbol{B}}$. The sum of the areas computed in this way can then constitute a global area metric for the whole contour map as

$$Area_{total}(\mathbf{S}) = \sum_{\boldsymbol{B} \in \mathbf{S}} Area(\boldsymbol{B}, \ell^{\boldsymbol{B}}) \ .$$

When a Bézier curve is updated, computing the area gain as a sum of individual area gains is advantageous for local updates. If only a few curves are modified, the efficiency of recomputing the loss improves, as only the corresponding terms of the sum need to be considered. More precisely if a curve $\boldsymbol{B}$ is modified into $\boldsymbol{B}'$ then the new area $Area'_{total}$ is simply

$$Area'_{total} = Area_{total} - Area(\boldsymbol{B}, \ell^{\boldsymbol{B}}) + Area(\boldsymbol{B}', \ell^{\boldsymbol{B}'}) \ .$$

We will now discuss the method used for computing $Area(\boldsymbol{B}, \ell^{\boldsymbol{B}})$. The area is split up into two parts as illustrated on Figure 3.2b. The first part is the area of the polygon defined by connecting the endpoints of $\ell^{\boldsymbol{B}}$ with the endpoints of $\boldsymbol{B}$. The second part is the remaining

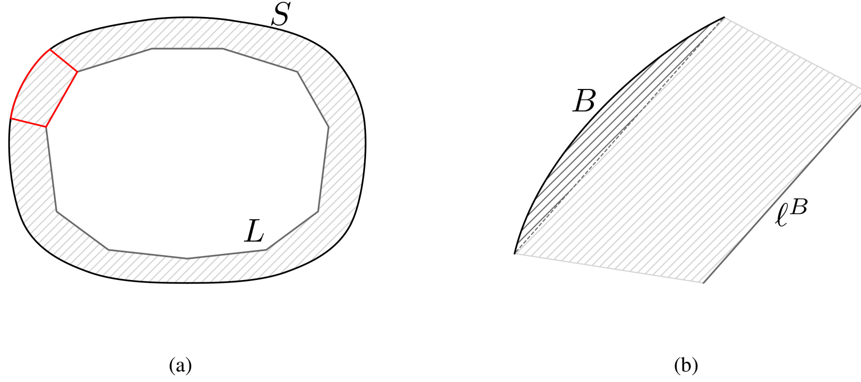

Figure 3.2: An illustration of the polygon L, an optimized contour $\mathbf{S}$ and area gain marked in hatched grey (a). (b) shows the part marked in red of (a). $\boldsymbol{B}$ is a Bézier curve and $\ell \in L$ is line segment. The dark hatched corresponds to $Area(\boldsymbol{B})$ and the light hatched area corresponds to the area of the polygon P

area inside the curve, which we already covered how to compute in Section 2.11 and denoted $Area(\boldsymbol{B})$. The area of a polygon is simple to compute using algorithms such as the *trapezoid formula* [1]. The trapezoid formula simply traverses line segments of the polygon and computes the area under the line segment. Putting all this together, we get the area between line segment $\ell^{\boldsymbol{B}} = [\boldsymbol{u}, \boldsymbol{v}]$ and Bézier curve $\boldsymbol{B}$ as

$$Area(\boldsymbol{B}, \ell^{\boldsymbol{B}}) = Area(\boldsymbol{B}) + \frac{1}{2} \sum_{(\boldsymbol{p}_0, \boldsymbol{p}_1) \in P} (\boldsymbol{p}_0^x - \boldsymbol{p}_1^x)(\boldsymbol{p}_0^y + \boldsymbol{p}_1^y) \,, \tag{3.5}$$

where $P = ([\boldsymbol{u}, \boldsymbol{v}], [\boldsymbol{v}, \mathbf{P}_3], [\mathbf{P}_3, \mathbf{P}_0], [\mathbf{P}_0, \boldsymbol{u}])$ is the polygon made from connecting endpoints of $\boldsymbol{B}[\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3](t)$ and $\ell^{\boldsymbol{B}}$.

3.5.3 Combining both Metrics

As mentioned earlier, our optimization has to consider both the curvature and gain in area. The Curve function in Equation (3.4) and Area function in Equation (3.5) are of different units, thus not directly comparable and we should not simply add them together. The Curve function is a product of a curvature ($\frac{1}{\mathrm{m}}$) and radius (m) and therefore the Curve function is unitless. The Area function outputs an area of unit m^2. By using the area score A defined in Section 3.4, we can convert the units of these two functions and in that way we obtain the combined function

$$Curve(\boldsymbol{B}) - \frac{Area(\boldsymbol{B}, \ell^{\boldsymbol{B}})}{A} \,. \tag{3.6}$$

Note that $Area$ is negated in the above formula, as minimizing $-Area$ is equivalent to maximizing $Area$. Thus minimizing Equation (3.6) results in lower curvature and larger area. Summing up all Bézier curves results in a final loss function we get

$$Loss(\boldsymbol{X}) = \sum_{\mathbf{S} \in \boldsymbol{X}} \sum_{\boldsymbol{B} \in \mathbf{S}} Curve(\boldsymbol{B}) - \frac{Area(\boldsymbol{B}, \ell^{\boldsymbol{B}})}{A} \,. \tag{3.7}$$

[1] `https://en.wikipedia.org/wiki/Shoelace_formula#Triangle_formula`

3.6 Defining the Constraints

In this section, we consider what defines a feasible solution and how one can decide if a solution is feasible. In Chapter 1 we described the criteria that must hold for our solution. In short, a feasible solution should satisfy:

Depth: The optimized contours are not allowed to be on the shallow side of the corresponding accurate contour at any point on the curve.

Intersection: The optimized contours may not intersect.

Topology: The optimized contours and initial contours must be topological equivalent.

In the following sections we describe these constraints more formally and, in addition, describe how they can be computed. Before diving into the actual constraints new definitions are required.

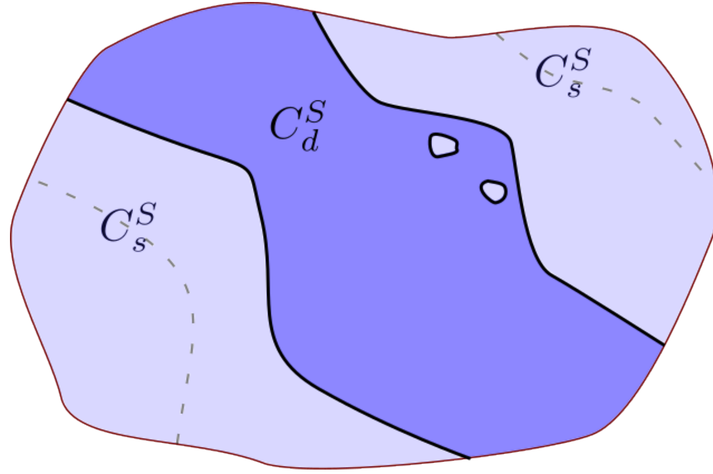

Figure 3.3: An illustration of the a deep region $C_d^{\mathbf{S}}$ (dark blue) and shallow region $C_s^{\mathbf{S}}$ (light blue) of a set of accurate contours $C^{\mathbf{S}}$ marked with a black line. Stripped lines indicate accurate contours of different heights

Consider the boundary of our problem, i.e., the edge of our dataset. As described in Section 1.2, the boundary, B, corresponds to one or multiple contour lines with a depth of zero. All other geometries in the dataset lie in the interior of the boundary. We use B_I to denote the set of all points in the interior of B.

Consider now also part of a solution $\mathbf{S} \in \boldsymbol{X}$ with height $\mathbf{S}^h$. Recall that $\mathbf{S}$ is a Bézier spline with a corresponding initial solution $\overline{\mathbf{S}}$. Recall also that $\mathbf{S}$ is a less detailed representation of some accurate contour. We denote the set of these accurate contours by $C^{\mathbf{S}} = \{\boldsymbol{A}_i : \boldsymbol{A}_i^h = \mathbf{S}^h, \boldsymbol{A}_i \in \boldsymbol{C}\}$, where $\boldsymbol{C}$ is the set of all accurate contours as mentioned in Section 3.4. As $C^{\mathbf{S}}$ is a set of contours of height h, it splits up the set B_I into sub-regions. A shallow region $C_s^{\mathbf{S}}$ less deep than h and the deeper region $C_d^{\mathbf{S}}$. Figure 3.3 illustrates this sub-division. There might exist multiple splines with height h in some solution $\boldsymbol{X}$. We denote the set of these $X^{\mathbf{S}} = \{\mathbf{S}_i : \mathbf{S}_i^h = \mathbf{S}^h, \mathbf{S}_i \in \boldsymbol{X}\}$. Similarly, $X^{\mathbf{S}}$ splits up B_I into two-regions which we define $X_s^{\mathbf{S}}$ and $X_d^{\mathbf{S}}$. Figure 3.4 illustrates this division.

3.6.1 Depth Constraint

The depth constraint must ensure that no optimized contour line overestimates the true depth. Mathematically, for all $\mathbf{S} \in \boldsymbol{X}$, the depth constraint is satisfied if and only if

$$C_s^{\mathbf{S}} \cap X_d^{\mathbf{S}} = \emptyset \,.$$

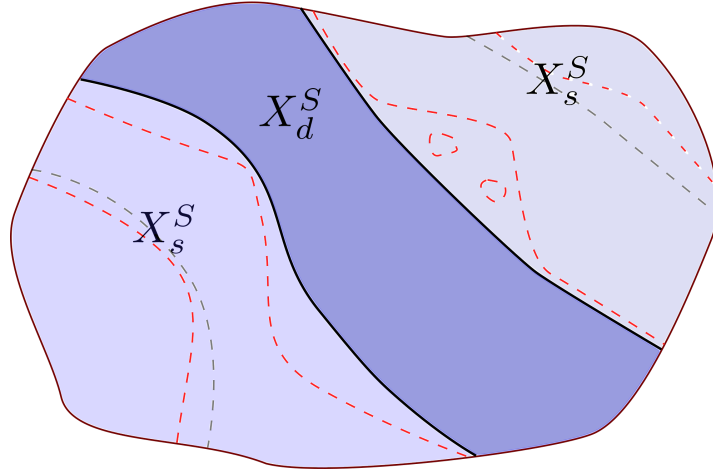

Figure 3.4: An illustration of the a deep region $X_d^{\mathbf{S}}$ (dark blue) and shallow region $X_s^{\mathbf{S}}$ (light blue) of a set of optimized contours $X^{\mathbf{S}}$ marked with a black line. Gray dashed lines indicate optimized contours of different heights and red dashed lines indicate accurate contours

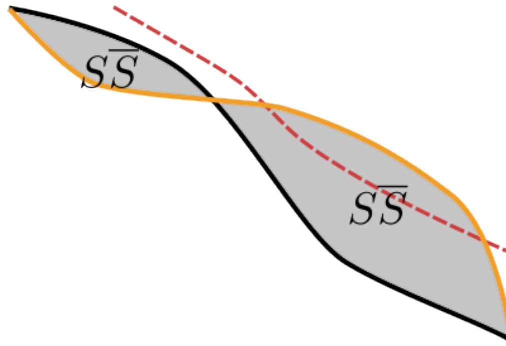

Figure 3.5: An illustration of an invalid update $\mathbf{S}$ shown in orange to an initial solution $\overline{\mathbf{S}}$ shown in black. The red stripped line represents an accurate contour

In other words, this means that no point on the deep side of the optimized contour is on the shallow side of its corresponding accurate contour.

A way of computing this is still required. In Section 2.9 we saw how one by computation can decide if a line segment intersects the interior of a closed Bézier spline. However, this cannot directly be utilized as $C^{\mathbf{S}}$ and $X^{\mathbf{S}}$ might not be closed if the contours connect with the boundary B. The algorithm can however be used to compute if a line segment in $C^{\mathbf{S}}$ is in the interior of the closed Bézier spline achieved by connecting the endpoints of $\mathbf{S}$ and $\overline{\mathbf{S}}$. We denote the set of interior point $\mathbf{S}\overline{\mathbf{S}}$ and Figure 3.5 illustrates this interior region. More precisely we can compute if

$$C_s^{\mathbf{S}} \cap \mathbf{S}\overline{\mathbf{S}} = \emptyset.$$

In the remaining part of this section, we show that this is enough:

Theorem 3.6.1 *Let $\overline{\mathbf{S}} \in \overline{\boldsymbol{X}}$ be part of a feasible solution $\overline{\boldsymbol{X}}$. Let $\mathbf{S}$ be an arbitrary Bézier spline with the same endpoints as $\overline{\mathbf{S}}$. Let $\boldsymbol{X}$ equal $\overline{\boldsymbol{X}}$ with $\overline{\mathbf{S}}$ substituted for $\mathbf{S}$. Then $C_s^{\mathbf{S}} \cap \mathbf{S}\overline{\mathbf{S}} = \emptyset \iff C_s^{\mathbf{S}} \cap X_d^{\mathbf{S}} = \emptyset$.*

To show that Theorem 3.6.1 we also consider the set $\overline{X}^{\mathbf{S}}$ as the set of contours in $\overline{\boldsymbol{X}}$ with height h. To ease the notation we use $D = X_d^{\mathbf{S}}$, $\overline{D} = \overline{X}_d^{\mathbf{S}}$ and $S = C_s^{\mathbf{S}}$ for this section. Recall that D and $\overline{D}$

respectively are the deep regions of an updated and feasible optimized contour of some height h. Similarly, S is the shallow region of the accurate contours of height h. As we assume $\overline{\boldsymbol{X}}$ to be feasible it is guaranteed that

$$S \cup \overline{D} = \emptyset, \tag{3.8}$$

as the initial solution would otherwise not be feasible. Additionally, consider the sets $\overline{D} \cap \mathbf{S}\overline{\mathbf{S}}$. These are shared points of $\overline{D}$ and $\mathbf{S}\overline{\mathbf{S}}$ and represent the set of points that are in $\overline{D}$ but not in D. Thus we describe D as

$$D = (\overline{D} \cup \mathbf{S}\overline{\mathbf{S}}) \setminus (\overline{D} \cap \mathbf{S}\overline{\mathbf{S}}).$$

For the first case assume $S \cap \mathbf{S}\overline{\mathbf{S}} = \emptyset$. It follows that $(\overline{D} \cup \mathbf{S}\overline{\mathbf{S}}) \cap S = \emptyset$, thus $S \cap D = \emptyset$. Likewise in the second case assume $S \cap \mathbf{S}\overline{\mathbf{S}} \neq \emptyset$, then

$$(\overline{D} \cup \mathbf{S}\overline{\mathbf{S}}) \cap S \neq \emptyset \subseteq S.$$

As a result of Equation (3.8) we have that,

$$\nexists \boldsymbol{p} : \boldsymbol{p} \in (\overline{D} \cap \mathbf{S}\overline{\mathbf{S}}), \boldsymbol{p} \in S,$$

and thus $S \cap D \neq \emptyset$. This shows us that checking in $\mathbf{S}\overline{\mathbf{S}}$ suffice. Checking against all points in S is infeasible to compute thus, we instead check against all line segments in $C^{\mathbf{S}}$. If there is no line segment in $\mathbf{S}\overline{\mathbf{S}}$, then $\mathbf{S}$ cannot enter the S and the depth constraint is satisfied. We have already seen a way to check if a line segments is in the interior of a closed Bézier spline in Section 2.9.

To summarize the method, intersection between each Bézier curve and constraint is checked. If they intersect, it is considered a violation. If not, an interior point algorithm is used for an arbitrary point on each line segment in $C^{\mathbf{S}}$ against $\mathbf{S}\overline{\mathbf{S}}$. The constraint is now satisfied if and only if no line segment intersects $\mathbf{S}$ and no point is in $\mathbf{S}\overline{\mathbf{S}}$. The correctness follows from Section 2.9. We denote the algorithm for checking if a line segment ℓ is in the set $\mathbf{S}\overline{\mathbf{S}}$ defined by two splines $\mathbf{S}$ and $\overline{\mathbf{S}}$

$$\mathit{Interior}(\ell, \mathbf{S}) = \begin{cases} 1 & \text{if } \exists \boldsymbol{p} : p \in \ell, \boldsymbol{p} \in \mathbf{S}\overline{\mathbf{S}} \\ 0 & \text{otherwise} \end{cases}.$$

Finally we define the depth constraint from the above function, $\mathbf{S}$ and $C^{\mathbf{S}}$ as

$$\mathit{Depth}(\mathbf{S}, C^{\mathbf{S}}) = \sum_{\boldsymbol{C} \in C^{\mathbf{S}}} \sum_{\ell \in \boldsymbol{C}^p} \mathit{Interior}(\ell, \mathbf{S}\overline{\mathbf{S}}).$$

Note that this only constraints a single Bézier spline, thus it must be applied to each $\mathbf{S} \in \boldsymbol{X}$ for a solution $\boldsymbol{X}$. The result of *Depth* equals the number of line segments in $\mathbf{S}\overline{\mathbf{S}}$, thus *Depth* is satisfied only when equal to 0.

3.6.2 Intersection Constraint

As mentioned in Section 1.2, contour lines represent a depth, thus two contours with different depth must never intersect. An intersection would indicate that a point has two depths, which cannot be true. The intersection constraint should prevent this by not allowing contours to intersect. Mathematically, given two contours $\mathbf{S}$ and $\mathbf{S}'$,

$$\nexists \boldsymbol{p} : \boldsymbol{p} \in \boldsymbol{B}, \boldsymbol{p} \in \boldsymbol{B}'$$

for Bézier curves $\boldsymbol{B} \in \mathbf{S}$ and $\boldsymbol{B}' \in \mathbf{S}'$. Checking the intersection between two Bézier curves is already covered in Section 2.8. We turn this algorithm into a 0-1-valued function that takes two Bézier curves $\boldsymbol{B}$ and $\boldsymbol{B}'$ and returns the result of the intersection check, that is,

$$\mathit{Intersection}(\boldsymbol{B}, \boldsymbol{B}') = \begin{cases} 1 & \exists \boldsymbol{p} : \boldsymbol{p} \in \boldsymbol{B}, \boldsymbol{p} \in \boldsymbol{B}' \\ 0 & \text{otherwise} \end{cases}.$$

Using the above function, intersection between two Bézier splines $\mathbf{S}$ and $\mathbf{S}'$ is computed as the function

$$Intersection(\mathbf{S},\mathbf{S}') = \sum_{\boldsymbol{B}\in\mathbf{S}} \sum_{\boldsymbol{B}'\in\mathbf{S}'} Intersection(\boldsymbol{B},\boldsymbol{B}') \ . \tag{3.9}$$

We use Equation (3.9) as the intersection constraint in the optimization problem. Note that the function computes the count of how many Bézier curves intersect, thus $Intersection(\mathbf{S},\mathbf{S}') = 0$ is required for the intersection constraint to be satisfied.

Computing intersections between all pairs of $\mathbf{S},\mathbf{S}' \in \boldsymbol{X} \times \boldsymbol{X}$ ensures that no contour lines intersect, however, less can suffice. As $\mathbf{S}$ is a contour line it splits up the plane on points of lower and higher elevations. Thus if $\mathbf{S}'$ has height $\mathbf{S}'^h > \mathbf{S}^h$, then it cannot intersect a contour of height $h < \mathbf{S}^h$ without also intersecting $\mathbf{S}$. Therefore, it suffices to only check the intersection of neighbor height contour lines.

3.6.3 Topology Constraint

The topology of the solution $\boldsymbol{X}$ and the initial solution $\overline{\boldsymbol{X}}$ must stay equivalent throughout the optimization as explained in Chapter 1. Recall the notation from Section 3.4: $X_d^{\mathbf{S}}$ and $\overline{X}_d^{\mathbf{S}}$ being sets of all points on the deep regions defined by all optimized contours of height $\mathbf{S}^h$ of $\boldsymbol{X}$ and $\overline{\boldsymbol{X}}$ respectively. Mathematically, for all $\mathbf{S} \in \boldsymbol{X}$ let Z and $\overline{Z}$ be the set of optimized contours lying in $X_d^{\mathbf{S}}$ and $\overline{X}_d^{\mathbf{S}}$ respectively then the topology constraint requires

$$Z = \overline{Z}. \tag{3.10}$$

That is, if a spline initially lies in the deep region of a contour, then it must also lie in the deep region after the optimization. Similar to the depth constraint we simplify the problem by just considering the interior $\mathbf{S}\overline{\mathbf{S}}$ of connecting $\mathbf{S}$ and $\overline{\mathbf{S}}$ at their common endpoints, and checking if any $\mathbf{S}'$ is in the interior. The correction is similar to Theorem 3.6.1.

Theorem 3.6.2 *Let $\overline{\mathbf{S}} \in \overline{\boldsymbol{X}}$ be part of a feasible solution $\overline{\boldsymbol{X}}$. Let $\mathbf{S}$ be an arbitrary Bézier spline with the same endpoints as $\overline{\mathbf{S}}$. Let $\boldsymbol{X}$ equal $\overline{\boldsymbol{X}}$ with $\overline{\mathbf{S}}$ substituted for $\mathbf{S}$. Finally, let z be the set of contours in the interior of $\mathbf{S}\overline{\mathbf{S}}$. If $z = \emptyset$ then $\boldsymbol{X}$ satisfy Equation (3.10).*

$\overline{Z} \cap z$ represent contours that are in $\overline{X}_d^{\mathbf{S}}$ but not in $X_d^{\mathbf{S}}$. This provides us with the following relation

$$Z = (\overline{Z} \cup z) \setminus (\overline{Z} \cap z)$$

Assuming $z = \emptyset$ then $Z = \overline{Z}$ meaning that contours in $\boldsymbol{X}$ and $\overline{\boldsymbol{X}}$ has the same relation, thus $\boldsymbol{X}$ also satisfy Equation (3.10).

Using Theorem 3.6.2 we can check if an update of a single spline of some feasible solution provides a solution satisfying the topology constraint. If a solution $\boldsymbol{X}$ has multiple splines differing from a feasible solution $\overline{\boldsymbol{X}}$, Theorem 3.6.2 can be used for each differing spline to check if $\boldsymbol{X}$ satisfies the topology constraint. For $\boldsymbol{X}$ to satisfy the topology constraint the set z in Theorem 3.6.2 has to be empty for all differing splines. This is true since if $z = \emptyset$ then $\boldsymbol{X}$ is feasible and can thus be fed to Theorem 3.6.2 targeting another spline.

The algorithm seen in Section 2.9 also works for checking if a Bézier curve is in the interior of a Bézier spline as of Problem 2.9.3. Thus the algorithm is used to define the function

$$Interior(\boldsymbol{B},\mathbf{S}) = \begin{cases} 1 & \text{if } \exists \boldsymbol{p} : \boldsymbol{p} \in \boldsymbol{B}, \boldsymbol{p} \in \mathbf{S}\overline{\mathbf{S}} \\ 0 & \text{otherwise} \end{cases}$$

for a Bézier curve $\boldsymbol{B}$ and Bézier spline $\mathbf{S}$. The topology constraint for two Bézier splines $\mathbf{S}$ and $\mathbf{S}'$ is now defined by the function

$$Topology(\mathbf{S},\mathbf{S}') = \sum_{\boldsymbol{B}'\in\mathbf{S}'} Interior(\boldsymbol{B}',\mathbf{S}\overline{\mathbf{S}}),$$

which is satisfied only when equal to 0. Applying the topology constraint for all pairs $\mathbf{S}, \mathbf{S}' \in \boldsymbol{X} \times \boldsymbol{X}$ ensures a correct solution, however, as we saw in Section 3.6.2 fewer constraints can also do the job. By only checking against the contours of neighboring heights, all topology errors should be caught. Say two non-neighboring contours violate the topology constraint. Then the violating contour *must* also violate the topology constraint with one of its neighbors.

3.6.4 Static Endpoint Requirement

This requirement is simply that endpoints of contours must be stationary. In this context, the endpoints of a contour are where the contour meets either the boundary of the dataset or itself resulting in a closed curve. It is not *strictly* needed, but having this requirement is a simple way to ensure that contour lines stay connected to the boundary or stay closed. Instead of modeling it as a constraint, we consider the endpoint position constant and thus not a part of the solution $\boldsymbol{X}$.

3.7 Final Optimization Problem

With all previous sections, we can finally define the optimization problem in standard form. For a solution $\boldsymbol{X} = (\mathbf{S}_i : i \in \{0, ..., n-1\})$ of n Bézier splines, with k depth constraints $\boldsymbol{C} = \{\boldsymbol{A}_i : i \in \{0, ..., k\}\}$, the problem is described by

$$\begin{array}{ll} \underset{\boldsymbol{X} \in \mathbb{R}^m}{\text{minimize}} & \mathit{Loss}(\boldsymbol{X}) \\ \text{subject to} & \begin{array}{ll} \mathit{Depth}(\mathbf{S}_i, H) = 0, & \mathbf{S}_i \in \mathbf{S}, H = \{\boldsymbol{A}_i \in \boldsymbol{C} \ : \boldsymbol{A}_i^h = \mathbf{S}_i^h\} \\ \mathit{Depth}(\mathbf{S}_i, B) = 0, & \mathbf{S}_i \in \mathbf{S}, B = \{\boldsymbol{A}_i \in \boldsymbol{C} \ : \boldsymbol{A}_i^h = 0\} \\ \mathit{Intersection}(\mathbf{S}_i, \mathbf{S}_j) = 0, & \mathbf{S}_i, \mathbf{S}_j \in \mathbf{S} \times \mathbf{S} \\ \mathit{Topology}(\mathbf{S}_i, \mathbf{S}_j) = 0, & \mathbf{S}_i, \mathbf{S}_j \in \mathbf{S} \times \mathbf{S} \end{array} \end{array}.$$

Note that there are two depth constraints. One is for the accurate contours with $\boldsymbol{A}_i^h = \mathbf{S}^h$ and one for the boundary $\boldsymbol{A}_i^h = 0$.

3.8 Analysis of the Optimization Problem

Optimization problems exist in numerous shapes thus large amounts of optimization algorithms have developed over time. Different algorithms take advantage of specific properties of the optimization problem for faster and better convergence. These are properties such as once or twice differentiable loss functions allowing for gradient base optimization or *Newton's method* respectively. Likewise, if the loss function and constraints are linear, *linear programming* is an obvious choice. These methods are iterative algorithms further detailed in [27].

In cases where the optimization problem does not have such properties, more general optimization techniques are required. These optimization problems are typically referred to as *black box optimization* since loss functions and constraints are treated as a black box with no assumptions on their interior functionality. These optimization techniques are more general and are applicable in a larger domain than previously named algorithms at the cost of efficiency or results.

Considering our model as presented in Section 3.7 it is clear that this is no simple optimization task as both the loss function and constraints are complicated non-linear functions. In the next subsection, we elaborate on our thoughts about the problem, more specifically the loss function.

3.8.1 Examining the Loss Function

The loss function as seen in Equation (3.7) is quite complex, consisting of multiple terms of area and curvature containing max and absolute functions. This leaves us with few to no properties to exploit, as we will explain below.

A differentiable loss function is desirable as it allows for computation of the gradient, which provides the steepest direction towards a lower loss. Intuitively choosing the steepest direction results in fewer optimization steps. While the *Area* function used in our loss function is quite involved, it is still just made up of simple terms, and thus computing a derivative should be a doable task.

However, the *Curve* function is not as simple. It consists both of taking max and absolute values, making it a non-smooth function and, thus not differentiable. One possible approach would be to split up the function into multiple smooth functions and compute derivatives of these. For a smooth function, the gradient provides information about the slope in the close neighborhood. When a function is non-smooth this property might no longer holds. That being said, a gradient might still be useful. One approach would be to numerically approximate the gradient. For a function $f(x)$ with $x \in \mathbb{R}^n$, some small value ε, basis vector e_i and $i = \{0, ..., n-1\}$ a popular approach for computing the partial derivative $\partial f / \partial x_i$ at x is

$$\frac{\partial f}{\partial x_i}(x) = \frac{f(x + \varepsilon e_i) - f(x)}{\varepsilon}.$$

Smaller ε values provide more precise derivatives up until a certain point where numeric instability overrules the result. We refer to [27, p 195] for a more elaborate explanation and bounds on numeric instability.

For the above reasons, we think it appropriate to consider the problem as a black box optimization problem. In Chapter 4 describe examples of algorithms suitable for these kinds of optimizations.

Remark: As black box optimization techniques are not able to exploit the properties of the functions, we cannot expect the same theoretical guarantees as exists with linear programming or gradient descent, for example, such as bounded conversion time and optimal results. As our optimization is run on large amounts of data, speed is crucial for our optimization. Finding the optimal value is not as important. As the output of our problem is, above all, meant to be visualized, ending up in a local minimum is not particularly detrimental. In other words, finding a local minimum close to the global minimum will suffice.

Future work: As mentioned, it would be interesting to experiment with approximating the gradient of our loss function. This would certainly provide a very different way of tackling the optimization problem perhaps shedding some light on the contents of the black box that we are dealing with in the present thesis.

Chapter 4

Black Box Optimization

As we have described in Section 3.8, searching for solutions to the optimization problem we have defined in Section 3.7 is a non-trivial task. The optimization problem has no clear properties that we are able to cleverly exploit, and we will thus consider black box optimization techniques. In this section, we present different approaches found in the literature to handle exactly this kind of black box optimization.

4.1 Direct Search Methods

In this section we describe *Direct Search*, a simple iterative black box optimization technique which, to our knowledge, was first phrased in 1961 [13]. In each iteration of a Direct Search algorithm the current solution is compared to a set of new solutions. A new and better solution is chosen, if it exists, and the algorithm proceeds to the next iteration. For this project, we refer to following definition from [19, p. 394]:

1. There is assumed to be an order relation $\prec$ between any two points x and y in the solution space. For instance, in unconstrained minimization, points x and y may be compared as follows: $x \prec y$ if $f(x) < f(y)$. That is, x is "better" than y because it yields a lower loss function value.

2. At any iteration, only a finite number of possible new candidates exists and the possibilities can be enumerated in advance.

Different Direct Search methods differ in how they choose new candidate solutions. A common approach in an optimization step is to choose the best candidate, where "best" is defined according to point 1 above. Note that this approach treats f as a black box, as no assumptions on the interior of f are made. Everything needed for such optimizations to run is an arbitrary function returning comparable values.

Direct search methods are often very simple and easy to implement. Having that said, Direct Search methods also have their liabilities. Many variations are pure heuristics with no guarantees of discovering a good solution. In certain cases, they are also slow, as convergence requires numerous steps. This is often a result of small step sizes and the problem of deciding if a search has hit a local minimum without having access to gradient information.

4.1.1 Compass Search

One of the earliest and simplest Direct Search methods is known as *Compass Search*. Having a solution $x \in \mathbb{R}^n$ and a step size s, candidate solutions are chosen by adding and subtracting s for

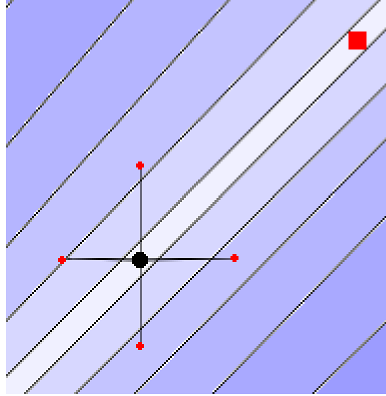

Figure 4.1: Compass Search on diagonal loss landscape. Lighter colors represent lower loss (desired) with a red square as a target.

each index in x e.g. the candidate set is described by

$$\{x + e_i \cdot s' : i \in \{0, \dots, n-1\},\ s' \in \{-s, s\}\},$$

where e_i is the standard basis vector. For a solution $x \in \mathbb{R}^2$, this corresponds to checking directions *north*, *south*, *east*, and *west* in the loss landscape as illustrated in Figure 4.1. If an improving solution is found, we iterate on that. If not, a new step size $s = s/2$ is chosen. Choosing a small initial step size s results in slow convergence while choosing a large s has a higher probability of so-to-speak overshooting. As Compass Search has $2n$ candidates, an optimization step is $O(n)$, which for large values of n this is quite slow.

Direct search has also been proven to perform better in lower dimensions [19, p 408]. In certain cases, Compass Search can also converge very slowly as illustrated on Figure 4.1. As the optimization step must move in a diagonal direction in order to hit the target square, it has to decrease s multiple times, leading to small optimization steps and thus slow convergence. All that said, Compass Search can in other cases quickly discover a reasonable solution, and is well suited for a naive solution for our optimization. As we see later in Section 6.1 a slight modification of Compass Search has been implemented, in which only a subset of the possible solutions are tried. Compass search will correspond to either moving a spline point or rotating or scaling its tangent. In Section 7.3 we experiment with how these parameters could be chosen.

4.1.2 Combined Compass Search

Consider the case where a spline point in our solution requires a move and a rotation to reach an optimal curvature. As our optimization problem has constraints, Compass Search might not succeed as a move, and rotation individually might violate a constraint while the combination will not. Thus, we expect that allowing optimization steps combining different dimensions can result in a lower loss.

This motivates what we call *Combined Compass Search* which simply extends the candidate set by all combinations of the previous candidate set. It is expected that this makes a single iteration slower, but results in lower loss. This approach is only applicable in low dimensions, as the number of combinations blows up. However, this is not a problem in our case, as our optimization algorithm only considers $x \in \mathbb{R}^5$ as described later in Section 6.1. In Section 7.1 most of our expectations are confirmed.

4.2 Random Search

In optimization, *Random Search* is the class of optimization algorithms that use randomness to define the method. These algorithms are interesting for the following reasons, as seen in [39]: Firstly, they are useful for ill-structured global optimization problems, where the loss function is non-convex and non-differentiable. Secondly, in contrast to deterministic methods such as *Branch and Bound*, which guarantees convergence to an optimal result, Random Search only guarantees convergence with some probability in order to gain efficiency. Finally, they are also easy to implement on complex black box problems. These are exactly the desired properties as described in Section 3.8, which is why we dedicate this section to describing them. Following terminology from [40], Random Search algorithms are categorized as *instance-based* or *model-based*.

Model-based Random Search algorithms generate solutions based on some distribution of the model, which may be updated during the optimization. It is not trivial how such an approach would apply to our problem as it would require some distribution of valid contour lines. Thus, we abandon this approach and instead consider instance-based Random Search.

Instance-based Random Search algorithms generate new solution based on the current solution, which is similar to Direct Search seen in Section 4.1. As the problem already provides an initial solution, and and optimal solutions probably is not far from the initial solution, instance-based algorithms seems like the direction to go.

4.2.1 Rastrigin's Family of Random Searches

The instance-based algorithms we consider in this project can be boiled down to two points of variation, being (1) the direction and (2) the step size. We denote the direction as a unit vector $d \in \mathbb{R}^n$ and the step size as a scalar value θ. That is, given a solution $x \in \mathbb{R}^n$, the solution of the next iteration, x', is computed by

$$x' = x + \theta d \ .$$

Choosing a suitable d and θ is the challenging part and it is done differently by the various optimization algorithms, but always some randomness is involved. Rastrigin [28] introduces a simple general approach which is to choose d uniformly from the n-dimentional unit sphere. This simplifies the problem into just picking a suitable step size. Keeping it simple, Rastrigin introduced the *fixed step size Random Search* (FSSRS) [28], which, as the name suggests, uses a predefined fixed step size combined with a uniformly random chosen d. At each step, the solution is updated so long as the update is improving the loss, i.e., $f(x') < f(x)$ for a loss function f. If a predefined step size is given, this method is extremely simple, and computing a new solution x' is fast. However, finding a suitable step size θ depends on the specific problem and is thus up to the user. Finding a suitable step size is possible in numerous ways. One of our configurations uses FSSRS, and we refer to Section 7.3 for elaboration of our parameter choice.

4.2.2 Alternative Step Sizes

Having to discover a suitable fixed step size, as is necessary with the above approach, comes with multiple drawbacks, namely (1) finding the optimal step size becomes an optimization problem in itself with no guaranties that a step size generalizes to other all datasets and (2) the optimal step size may vary throughout the optimization, which a fixed step size cannot model. This motivates other step size approaches like *Optimum Step Size Random Search* (OSSRS) [26] or *Adaptive Step Size Random Search* (ASSRS) [31]. The Optimum Step Size approach uses a variable θ and aims for the optimum θ' at each step, such that for any θ

$$f(x + \theta' d) \leq f(x + \theta d) \ .$$

Computing θ' may not be trivial, but is often simpler, as it is only a one-dimensional optimization problem. Using an optimal step size requires fewer steps for convergence, however,

each step is slower to compute compared to FSSRS. If computing θ' prime is too slow this method yields no performance boost. Additionally, computing θ' depends on f, thus it has to be tailored for each specific problem.

ASSRS was invented as an extension to FSSRS to solve drawbacks (2). The intuition of ASSRS is to increase the step size if a larger step size yields better loss and on the other hand, decrease the step size if all possible steps would end up in an infeasible solution. More precisely, assume we have some step size θ, a value a such that $1 > a > 0$, and a counter c. For a uniform random unit vector d and another step size $\theta' = \theta(1+a)$ compare

$$f(x), f(x+\theta d), f(x+\theta' d) \ .$$

If $f(x)$ is smallest, increment c and do nothing. If c exceeds some limit m, the step size is reduced. If $f(x+\theta d)$ is smallest, do the optimization step with step size θ. If $f(x+\theta' d)$ is smallest, do the optimization step with θ' and update $\theta = \theta'$. In this way, if the larger step size improves the results, it is chosen and if the step size is too large to yield improvement for some time, it is reduced.

This optimization has hyperparameters a, m, and the initial step size θ. If a bad step size is chosen the algorithm will eventually adapt, and this just results in a slow start. How fast the algorithm adapts depends on a and m.

Future work: The paper [31] in which Adaptive Step Size Random Search was introduced shows great potential for this method in higher dimensions compared to derivative-based approaches such as the Newton–Raphson method. Testing it for our problem would thus be interesting, however, this is left as future work.

Future work: Common for all previously described optimization methods is that they only do improving steps. Only accepting improvement steps is beneficial for fast convergence but also comes at a cost. In certain cases, it might be beneficial to perform a step resulting in worse loss, as it might expose the algorithm to new minima of possibly lower loss. An illustration is shown in Figure 4.2, where loss compromises have to be made to move from a local minimum (light blue dots) to the global minimum (dark red). For this reason, it would also be interesting to investigate

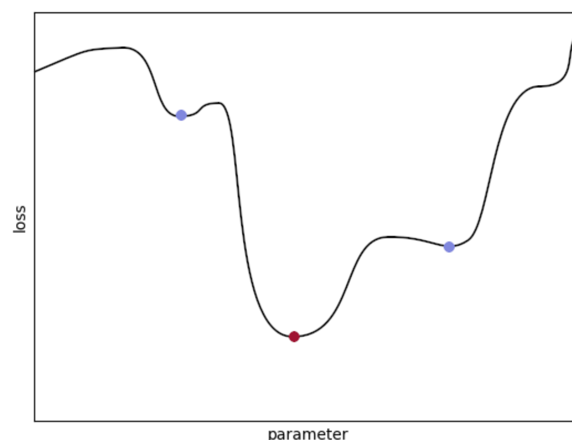

Figure 4.2: An example loss curve over one parameter. Light blue and dark red points mark local and global minimum respectively.

methods allowing steps compromising steps: *Simulated annealing* [18] is an interesting example that can be used in conjunction with the previously described Random Search methods. In short, the idea is to allow compromising steps with a certain probability. This probability is defined by a cooling function $h(t)$ where t is a time parameter. As t increases h 'cools down' resulting in lower probabilities of compromising steps. Simulated annealing requires a definition of a neighboring solution. For our problem, neighbor solutions could be defined as solutions close in Euclidean space. More precisely a solution x and $x' = \theta d$ would be neighbors for some step size and direction defined using one of the methods seen previously. Using simulated annealing for this optimization is left as future work.

Chapter 5

Constraint Checking with R-trees

In a single step of the optimization algorithm, we must verify that the changes made to the contour do not make the solution infeasible, i.e., it must not violate any of the constraints of the problem. The naive approach to this verification is to test the curves affected by the change against *all* accurate and optimized contours in the dataset. Evidently, this is neither a clever nor efficient method. This section details our effort to utilize so-called R-trees as a data structure to store contour lines and perform efficient queries on them. The textbook by Y. Manolopoulos et al. [25] has served as a reference for the section.

5.1 The Original R-tree

The R-tree was introduced by Guttman [12] in an effort to efficiently answer queries on geometric objects. Specifically, he tried his data structure in a Very-Large-Scale Integration (VLSI) context, where a query could be formulated as: "Is this area on the die already occupied?". The R-tree can efficiently answer this query because it creates an index of the objects according to their spatial location.

The R-tree is, in essence, a balanced search tree, where the geometric objects stored are represented by minimum bounding rectangles (MBRs). Each internal node in an R-tree represents an MBR that completely encloses its children. Internal nodes of the tree contain pointers to child nodes as well as the minimum bounding rectangle that encompasses all the rectangles in the child nodes. The leaves then contain the actual data objects (or pointers to them). As such, the root of the tree will have an MBR enclosing the whole spatial region along with pointers to child nodes with more and more fine-grained MBRs as we move down the tree. It should be apparent how this structure aids in indexing: If a query rectangle does not overlap with the MBR of a given node u in the tree, then it cannot overlap with any of the children of u. Thus the whole subtree rooted at node u can be safely ignored in the search. However, the MBR of some geometric objects can be larger than the object itself. This means that results returned by the tree have to be examined more closely in another routine, specific to the type of geometric object stored and the query type. Figure 5.1 illustrates a small example of MBRs in an R-tree over two Bézier curves.

To be more precise, and following the presentation in [25], an R-tree is a height-balanced search tree with fan-out given by two numbers m and M s.t. $m \leq M/2$. Leaf nodes contain entries of the form `(mbr, id)`, where `mbr` is the MBR of the geometric object stored in the entry, and `id` is some identifier of the object. Internal nodes contain entries of the form `(mbr, p)`, where `mbr` is the MBR of the (multiple) MBRs contained in the child node pointed to by `p`. The tree must satisfy the following properties:

- Every node in the tree contains between m and M entries, except for the root, which is allowed to have less than m entries.

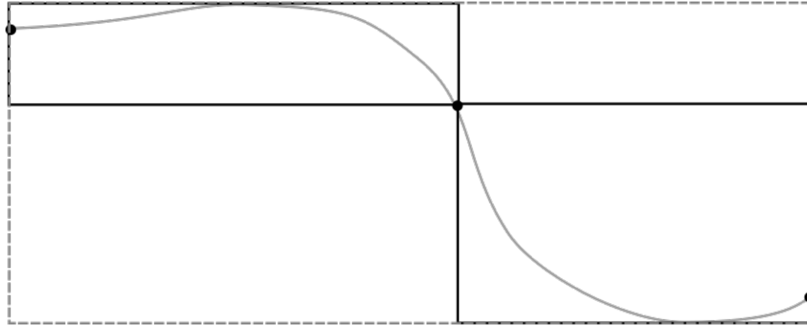

Figure 5.1: A small example of an R-tree over two Bézier curves. Black boxes are MBRs of the Bézier curves representing leaves. The dashed gray box is the MBR of the two leaves.

- The minimum number of entries in the root is 2.[1]
- All leaves of the tree are at the same level/depth.

In general, R-trees support the following operations: `Insert`, `Delete`, `Split`, and `Query`. Most of these are self-explanatory in regard to their effect on the tree. The `Split` operation is necessary to keep the size of the tree nodes within the allowed range: If an object is inserted such that the tree node u it ends up in exceeds M entries, the `Split` operation splits u into two or more new nodes.

We want to briefly mention two aspects of R-trees that will be relevant to us. Firstly, since nodes in the tree are not required to be at full capacity, it makes sense to talk about the *storage utilization* of a specific tree. Basically, how much of the space reserved by the data structure is filled with actual data. Secondly, the standard R-tree definition allows nodes at the same level in the tree to have overlapping MBRs. When querying either a point or area, this can lead to trying multiple search paths before finding the correct leaf. As such, building the tree in such a way that MBR overlap is minimized is often desired.

5.1.1 Dynamic and Static Variations

An important distinction between types of R-trees is whether they are designed to be *dynamic* or *static*. We will briefly explain the difference, and then argue for which type best suits our application.

> A **dynamic** R-tree is designed to handle objects being inserted one-by-one and to handle queries mixed into these insertions. When the tree is built in this way, the structure of the tree is impacted mostly by the co-called *splitting criteria* used, i.e., what to do when a node has reached its capacity and needs to be split into two new nodes. The typical storage utilization of dynamic trees is around 60% to 70% [3].
>
> A **static** R-tree is designed to have certain advantages when the set of objects is of a static nature and is all known beforehand. This enables some thought to be put into the way the tree is constructed from the objects, possibly giving it some desirable characteristics, such as good storage utilization or overlap minimization. The specific way the objects are prepared is called *packing*, and the static R-trees are primarily distinguished by their chosen approach to packing.

In the present context, we are trying to improve the performance of constraint checking. The **depth constraint** requires checking against the accurate contour lines. The accurate contours *do*

[1] As a special case, if the root is itself a leaf, it is allowed to contain only zero or one entry.

not change during execution, and furthermore are known to us from the get-go. In combination, these facts enable the use of static trees. As such we focus our attention on describing a static variation of the R-tree in the following section.

Note: Static R-trees are not static in the sense that they disallow insertions, deletions, and splitting. As such, a static tree can be built using packing to have a high storage utilization, but if lots of insertions and deletions are performed afterward, the properties of the tree will of course degenerate.

5.2 The Hilbert Packed R-tree

In this section, we first describe the properties of Hilbert Packed R-tree (HPR-tree) that make it interesting to us. We will then go on to explain how *packing* is done, which is the distinguishing feature of this static data structure.

The HPR-tree aims to provide close to 100% storage utilization without impacting the performance of queries, that is to say, it has at least the same performance as the *R*-tree*[2] [16]. In particular, the high storage utilization makes the HPR-tree a promising data structure in our application since, intuitively, it gives better scalability. In addition, as will become apparent below, packing using Hilbert values is simple to implement.

With the above motivation mentioned, it should be noted that HPR-trees are based on heuristics, and as such do not provide any improved theoretical guarantees with regard to query performance compared to traditional R-trees. However, empirical results show good query performance compared to both dynamic R-trees and other static R-tree versions [25, 16, 23]. We now look into how it is achieved.

5.2.1 The Hilbert Curve

The Hilbert curve [30, pp. 9–30] is a so-called *space-filling curve*, which means that it is a recursively-defined curve that, in the limit, visits all points in k-dimensional space exactly once and never crosses itself. In our application we have $k = 2$, so we think of the curve as a mapping from 2-dimensional coordinates to *Hilbert values*. Figure 5.2 shows how a Hilbert curve is constructed from a basic pattern (a curve of order 1). For higher orders, each "part" of the curve is replaced by the order 1 pattern, possibly rotated, and the ends are connected. What we mean by the curve visiting all points in the plane in the limit is, specifically, that we let the order of the curve go toward infinity. In practice, the appropriate order curve to use is determined by the precision of the coordinates of the underlying data.

The important property of the Hilbert curve is that it imposes a linear ordering on all points in the plane. Again, consider the curve of order 1 in Figure 5.2: Each tile in the 2-by-2 grid can be denoted by a (x,y) pair of coordinates. The curve imposes an ordering on these coordinates s.t. $(0,0) \mapsto 1$, $(0,1) \mapsto 2$, $(1,1) \mapsto 3$ and $(1,0) \mapsto 4$. So, when the tiles are sorted based on their Hilbert values, their ordering will follow the shape of the curve in the plane. The Hilbert curve is, as mentioned, a type of space-filling curve, which implies that others do exist, notably the Z-order curve. In [9] it is argued that, of the two, the Hilbert curve is the best mapping for preserving distance. As such, points that are close in the plane are most likely to have 1-dimensional values that are close to each other, if the Hilbert curve is used.

5.2.2 Packing a Tree

The defining characteristic of static R-trees is as mentioned in Section 5.1.1 the way they are *packed* a.k.a. *bulk loaded*, which essentially is just a term referring to how data is prepared before

[2] The R*-tree is often used as a benchmark with regards to querying, which is explained in [25].

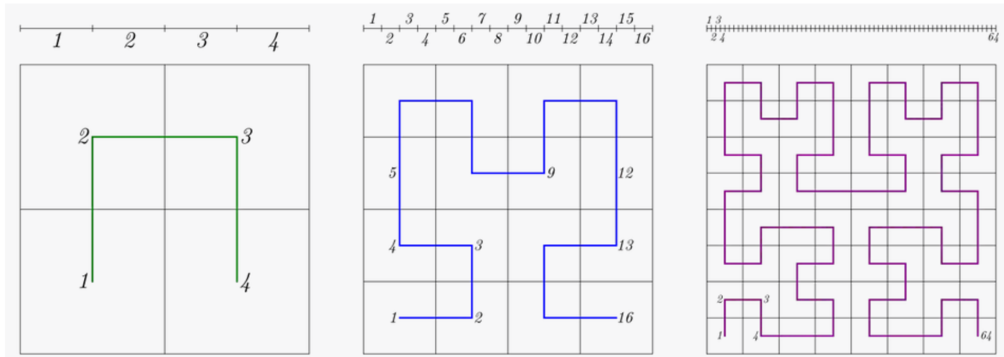

Figure 5.2: Image showing the 1st- (left), 2nd- (middle) and 3rd-order (right) Hilbert curves. The curve has been annotated with the Hilbert value of the points in 2-dimensional space along the curve.
Source: By User:Braindrain0000, CC BY-SA 3.0,
`https://commons.wikimedia.org/w/index.php?curid=47570255`

the tree is built. The concept of packing was introduced in [29], but we will go straight to Hilbert packing, as it deviates very little from the original approach.

The packing algorithm for the Hilbert Packed R-tree, as introduced in [16], takes a bottom-up approach, roughly as follows: The input rectangles are sorted according to their Hilbert value. The Hilbert values are computed from the center point of the rectangle. Now for some integer $c > 1$, go through the list of rectangles and assign each c consecutive rectangle to a leaf, creating new leaves as needed. Once this is done, all leaves will be at their capacity except potentially the last one. The MBR of the leaf nodes are computed and the procedure repeats to create the next layer in the tree until the root is reached. Note that the Hilbert value is only used to sort the input rectangles. Any subsequent sorting of MBR nodes is done from their creation time.

The nearly 100% storage utilization of the resulting R-tree follows directly from the packing method. In each layer of the tree, all nodes except one are guaranteed to be at full capacity. In this project, we used $c = 2$ thus constructing a binary tree.

5.3 Lazy Update R-tree

Our optimization problem is subject not only to depth constraints but also to **intersection constraints** as explained in Section 3.6.2. To briefly recap, two contour lines are not allowed to intersect, as this would indicate two different water depths at the same point, which is impossible. As such, we are forced to not only check for intersections between contour lines and depth constraints, but also between contour lines and *other* contour lines.

At first, this might seem like a problem. After all, the reason we can benefit from the (static) Hilbert Packed R-tree in Section 5.2 is that the line segments representing the depth constraints *do not change* during the optimization. On the contrary, all contour lines are subject to changes during a run of the optimization. Since the geometric objects we are trying to store are not static by nature, does it mean we cannot use a static R-tree? In the next subsection, we describe how this "problem" is handled.

5.3.1 Dynamic Trees Not Needed

We wish to store the contour lines in a data structure such that, given a Bézier curve (remember, contour lines are represented as Bézier splines a.k.a. multiple Bézier curves) we can efficiently determine if the curve intersects with any other curves in the data set. Even though the contour lines are not completely static throughout the optimization, we do have access to an initial

solution as the problem input. This fact alone enables us to use the HPR-tree. However, this also implies that we have to update the tree each time a Bézier curve is changed by the optimization algorithm, which happens in *every* iteration. We want to avoid this for two reasons:

- Updating an R-tree can be a slow procedure since the tree properties have to be upheld. Less time spent updating the tree should, as a consequence, give faster iterations.
- The more frequent updates are needed, the less suited the R-tree is for parallel access. With parallelization of the optimization algorithm as motivation, we want to keep the R-tree as un-changing as possible.

The Lazy Update R-tree (LUR-tree) is introduced in [21] as a way to postpone doing updates of the tree, without impacting the search query performance too much. We recall that the curves in the HPR-tree are represented as their MBRs. A change to a curve results in an updated MBR, which in turn should result in an update of the tree. The idea behind lazy update is to replace the normal MBR with an *extended MBR* (EMBR). As the name suggests, this EMBR is larger than the MBR by some fixed amount, if the EMBR was used when building the R-tree. Any update to the curve such that the curve is still within the EMBR by definition does not require an update. Only when a curve is changed such that it exceeds the bounds of the EMBR is an update required. Note that this change to the data structure is only useful if the changes in stored geometry positions are local in Euclidean space and tend to stay inside the EMBR. Luckily our application of R-trees only does small changes to the Bézier curves. Additionally, the optimized Bézier curves are expected to stay close to the corresponding initial curve. This expectation is validated in Section 7.4.

Bulk Updating: In [21] an update of the tree is performed by deleting the old element and inserting the new/updated element. Even though this happens lazily, the outcome of the update is, in essence, that only the curve that triggered the update obtains a new EMBR. Other curves might still be a single optimization step away from triggering an update.

In order to make the R-tree even more static, we present a heuristic we call *bulk updating*: When, at some point, a change to a curve triggers an update of the HPR-tree, the whole tree is rebuilt with current curves as described in Section 5.2.2. The intuition is as follows: Since in each iteration of the optimization a uniform random curve is chosen to be updated, then if some curve triggers a tree-update by being moved outside its EMBR, most likely, other curves in the tree are also only a few steps from triggering a tree-update. By giving all curves a new EMBR the next tree-update is deferred even more. When bulk updating is used, we can refer to our tree as a Lazy Bulk Update Hilbert Packed R-tree (LBUHPR-tree). Although we have not benchmarked the two methods against each other, we believe lazy bulk updating to be of comparable speed to lazy normal updates on average, because packing/bulk loading HPR-trees is highly efficient.

In Section 7.4 we document our benchmarks of using this data structure for our optimization problem. We provide a brief summary of the section here. The slowest performance was on D3 for which construction took 67.9 milliseconds on average. When MBRs were extended by 0.48 millimeter in chart scale in each direction, this incurred a total of 2 bulk updates over 10 000 optimization steps, with approximately the same query time and the average number of objects returned in each query increasing by about 3%. In other words, this seems very efficient.

Chapter 6

Contour Optimization Algorithm

Previously, we have introduced our optimization problem on an intuitive level in Chapter 1 and in detail in Chapter 3. In this section, our main goal is to describe how the optimization algorithm solves the optimization problem as stated in Chapter 1. This will include a lot of references to earlier sections of the thesis, as most of the theory used has already been accounted for. We will also introduce the so-called *configurations* that the optimization algorithm can use.

We begin in Section 6.1 by explaining how the algorithm searches for solutions, and notably how we have implemented it slightly differently from the theory in Chapter 4. Then we move on in Section 6.2 to explain how we check the feasibility of solutions. In Section 6.3 we summarize the different configurations we choose to benchmark and finally in Section 6.5 we show some example output of the optimization algorithm. Our algorithm has the following signature:

> **Input**: initial contours and accurate contours
>
> **Output**: optimized contours

Recall in the following sections that initial contours are Bézier splines that we want to improve upon. The accurate contours are polylines needed to constrain the problem. The optimized contours are represented by Bézier splines as well.

6.1 Searching for Solutions

In section Chapter 4 we presented different ways of optimizing solutions when looking at the optimization problem as a black box. We have implemented methods of both search variations described in that section, namely two direct search variations and one random search variation:

> **Direct search:** Both direct search methods have been implemented according to Section 4.1. In the configurations, we call them *Compass Search* and *Combined Compass Search* respectively.
>
> **Random search:** The random search method has been implemented according to Section 4.2. In the configurations, we refer to it as *FSSRS*.

Common to these is that, in theory, when going from one solution to the next, *all* variables of the loss function are changed simultaneously. We want to highlight that our algorithms look for new solutions by changing only a few variables at a time. This is true for all three variations implemented. In the following, we explain why this is possible as well as the motivation behind it.

6.1.1 Local Updates Only

As mentioned in Section 3.3 we formally represent solution contours using the spline point representation, which results in solution vectors in $\mathbb{R}^m$ where m is the dimension according to Equation (3.3). Generally speaking, the search methods described in Chapter 4 all go from one solution to the next by adding some search vector of the same dimension m to the solution vector. In practice, our changes are much more localized, with only a few entries of the m-dimensional search vector being different from zero. Mathematically, instead of considering updates to the loss function as updates all entries of $X \in \mathbb{R}^m$, we instead choose $X' \in \mathbb{R}^5$ uniformly random corresponding to changing a single spline point. This is a special case of the general unit search vector, and as such the correctness of the optimization methods follows from the arguments given in Section 3.6. To elaborate, our algorithm is correct if it produces a feasible solution. Such local updates are indeed also practically possible, the reason being as follows. Recall that solutions use the spline point representation. By the endpoint interpolation property of Bézier curves, updating a spline point can only affect the shape of the two curves that share the spline point as a common endpoint. Since we know what curves are affected by an update, it is easy to subsequently check the feasibility of the update.

Motivation: The main reason we choose to do local updates is as follows. Each update to the loss function requires the constraints to be satisfied in order to have a feasible solution. Any update leading to an infeasible solution can thus be considered a wasted update in terms of computation. Local updates increase the efficiency of the optimization step by so-to-speak detecting constraint violations sooner. Say we have a spline $\mathbf{S}$ made from n Bézier curves, i.e., $|\mathbf{S}| = n$, and accurate constraints $\boldsymbol{C}$ with $|\boldsymbol{C}| = m$. Then, if only one of the curves in an updated $\mathbf{S}$ is responsible for the solution being infeasible, we will have worst-case made nm computations in order to find out. By performing updates only on the smallest spline possible, i.e., $|\mathbf{S}| = 2$, we can detect constraint violations in $2m$ computations. This argument applies only to the naive constraint checking approach where all curves in a spline are checked against all constraints. However, when R-trees are used to filter which constraints the spline potentially intersects with, the smaller query box of a shorter spline will return fewer constraints, increasing performance.

Pitfalls: The most severe pitfall we have identified with regard to local updates is the likelihood of ending up in a local minimum. In some cases, the only way out of a local minimum might very well be to update several spline points simultaneously, which is currently not possible in our implementation. An example of such local minimum is shown in Section 6.5. However, as explained in Section 3.8 we choose to value the speed of computation over finding a global minimum.

6.2 Implementation of Constraint Checking

It remains to be described how constraint checking is implemented, i.e., how we make sure that the constraints of the optimization problem of Section 3.7 are upheld. We give a brief description of each constraint below.

Depth: The depth constraint is checked by the algorithm described Section 3.6.1. Specifically, we use $Depth(\mathbf{S}, H)$ where $\mathbf{S}$ is the spline containing only the two curves possibly affected by updating a spline point. Since we can see H as the set of accurate contours that an updated solution *might* be in violation with, reducing the size of the set should intuitively provide a performance increase. An important point of variability between the different configurations is how many accurate contours are passed to $Depth(\mathbf{S}, H)$ as the parameter H, namely

All Contours: The case where all accurate contours of the contour map are part of H we refer to as the *exhaustive search* variation.

Some Contours: The case where a Packed Hilbert R-tree (see Section 5.2) is used as a filtering mechanism to bring down the size of H we refer to as the *Hilbert R-tree* variation.

Intersection: The intersection constraint is checked by the algorithm described in Section 3.6.2. Specifically, we use $\mathit{Intersection}(\mathbf{S}, \mathbf{S}')$ where $\mathbf{S}$ is the spline containing only the two curves possibly affected by updating a spline point. Analogous to the case of depth constraints, the number of elements in the second parameter $\mathbf{S}'$ is proportional to the amount of computation needed. As such, we have similar two variations of intersection checking, one using exhaustive search and one using a LBUPHR-tree (see Section 5.3) as a filter.

Topology: The topology constraint has, in this thesis, not been implemented. However, implementing it would, similar to the other constraints, be a matter of following the procedure described in Section 3.6.3 and use $\mathit{Topology}(\mathbf{S}, \mathbf{S}')$. We skipped it as we figured optimization steps to be so small that they would never jump an entire optimized contours. When this is the case the intersection constrain is sufficient.

6.3 Configuration Summary

We can summarize the different configurations described above as follows. The configurations are different combinations of (1) optimization method and (2) constraint checking.

Compass: Uses Compass Search for solutions combined with Exhaustive Search for constraint checking.

Compass R-tree: Uses Compass Search for solutions combined with Hilbert R-trees (both HRP-tree and LBUHPR-tree) for constraint checking.

Combined Compass R-tree: Uses Combined Compass Search for solutions combined with Hilbert R-trees (both HRP-tree and LBUHPR-tree) for constrain checking.

FSSRS R-tree: Uses Fixed Step Size Random Search for solutions combined with Hilbert R-trees (both HRP-tree and LBUHPR-tree) for constraint checking.

Optimization step: Our optimization algorithm is iterative and we denote an iteration as an *optimization step*. The optimization step depends on the configuration but all configurations have the following steps in common:

1. Choose a uniform random spline point $p \in \mathbb{R}^5$.
2. Choose a step direction $d \in \mathbb{R}^5$ and step size $\alpha \in \mathbb{R}^5$ using the specified search method.
3. Compute new spline point $p' = \alpha \odot d + p$, where $\odot$ represent entry wise multiplication.
4. Let $\boldsymbol{X}$ be the old solution and $\boldsymbol{X}'$ the new solution with p'.
5. If $\mathit{Loss}(\boldsymbol{X}') \geq \mathit{Loss}(\boldsymbol{X})$, return $\boldsymbol{X}$ as the solution
6. If $\boldsymbol{X}'$ violates a constraint constraint, return $\boldsymbol{X}$ as the solution
7. return $\boldsymbol{X}'$ as the solution

Note: The step size used is a vector as opposed to a scalar as described in Chapter 4. This is a result of the *angle* of a spline point being more sensitive to scaling than its other entries. Using a vector different step sizes are used for each entry in the spline point. In Section 7.3 we experiment with different values of α.

The equation $Loss(\boldsymbol{X}') \geq Loss(\boldsymbol{X})$ is computed quite efficiently when only a single spline point is changed. Generally, when $Loss(\boldsymbol{X})$ is known, the time for computing $Loss(\boldsymbol{X}')$ is proportional to the number of spline points that are changed. This is explained in further detail in Section 3.5.2. Additionally, loss is computed before the constraints are checked and the constraints are only validated if loss has improved.

Future work: The optimization algorithm can be run a fixed amount of steps, a fixed amount of time or until it is manually stopped. It would be interesting to look at defining *stopping criteria* for the configurations. Defining a common stopping criteria in particular is not a trivial task, as the search methods behave differently.

6.4 Correctness

The correctness of our implementation relies on the feasibility of the initial contours we provide as input to the optimization algorithm. That is, it can be seen as an argument of mathematical induction, where the base case is feasible by assumption. Given that the initial contours are feasible, and that each step of the optimization algorithm outputs a new feasible solution by the arguments of Section 3.6, we argue that the optimized contours must also be feasible.

We also argue that the R-tree queries provide all relevant geometries needed to determine feasibility. In Section 3.6 we proved that checking for constraints only requires checking against geometries in the interior of the closed spline achieved by connecting the previous spline with the updated spline. Thus, when querying the R-trees the query rectangle is chosen as the MBR of this closed spline.

6.5 Visualization of Results

In this section, we visualize results of our optimization. For performance comparisons between configurations we refer to Section 7.1. In short, the FSSRS R-tree configuration performs best, thus all the following illustrations are the results of this configuration. Performing 100 000 optimization steps on D1 results in Figure 6.1.

Figure 6.1: D1 after 100 000 optimization steps. Black lines are optimized contours, pink lines are the accurate contours, and dashed red lines are the boundary.

The figure primarily shows nice smooth contour lines of low curvature. However, not all curves are as smooth as desired. In certain areas the target curvature cannot be achieved without violating the depth constraint. This is fully acceptable as constraints are more important than a low curvature. Other curves as seen in Figure 6.2 can be improved without violating constraints.

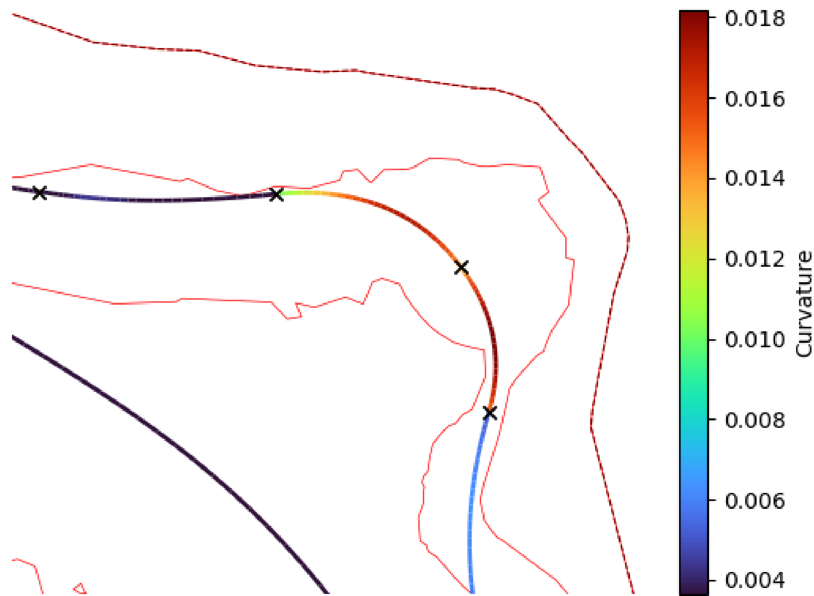

Figure 6.2: Part of D1 showing a local minimum which the optimization algorithm cannot escape. Colors indicate higher than desired curvature and crosses represent spline points.

This requires further explanation. The figure illustrates part of D1, where the solution is colored depending on the curvature along the curve. Black indicates that the curve has target curvature or lower and colors indicate different curvature above the target. Notice the colored spline and the accurate contour in red right under it. The spline *is* allowed to cross this accurate contour, as it represents a deeper depth than the spline. If the spline crossed this accurate contour, it could achieve the target curvature, and thereby reduce loss. It would obviously result in a higher area loss, but not as great as the decrease in curvature loss. This indicates that the optimization is stuck in a local minimum, a sub-optimal solution which it cannot escape. This does make sense if we inspect the spline points marked with crosses on the illustration. The optimization only updates one point at a time and only allows for improving solutions. It is not clear how a change to just a single of these spline points would decrease loss as they would either violate the depth constraint or increase the curvature. This motivates for optimization steps changing multiple points at a time and/or a search method with techniques to escape local minima like described in the future work paragraph of Chapter 4. That being said, the optimization is still a clear improvement to the initial solution. Figure 6.3 illustrates the initial solution compared to the optimized solution in part of D1 after 100 000 optimization steps. All sharp turns are improved and a large area of improvement is visible in the middle of the figure. These results indicate that the loss function in combination with the FSSRS configuration to some extent solves the problem described in Chapter 1.

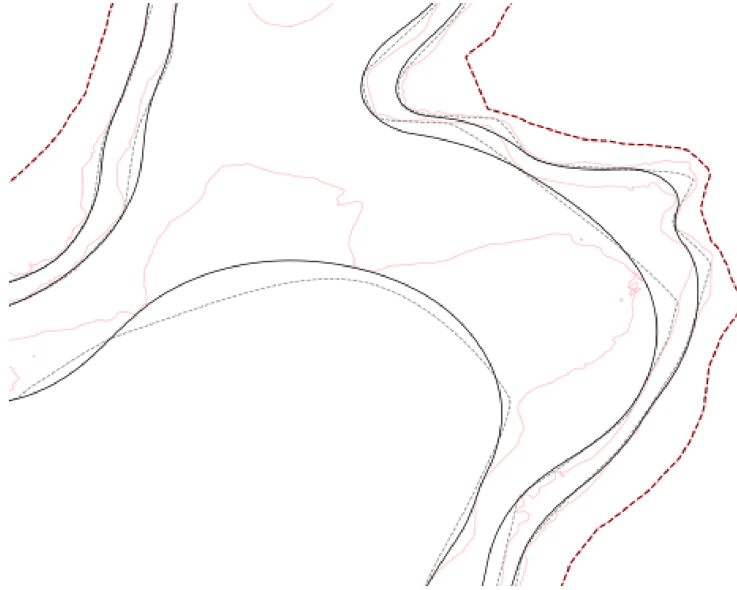

Figure 6.3: Part of D1 after 100 000 optimization steps. The gray dashed lines represents the initial solution and the black lines the optimized solution.

D2 and D3 are way larger, thus illustrating the whole dataset does not provide a clear picture of the optimization. Figures of optimization of these datasets are found in the appendix as Figure C.1, Figure C.2 and Figure C.3.

Chapter 7

Benchmarks

In this section we present a series of benchmarks that lead to our choice of hyperparameters for the search methods, as well as providing insight the how the different configurations perform. The benchmarks were conducted on an 8 core 3.4GHz machine with 8GB of memory running Arch Linux 64-bit.

7.1 Comparing Configurations by Loss

Loss defines how well a solution performs and thus well suited for comparing our different configurations. This section explores the change in loss as a function of either optimization steps or time. Inspecting loss as a function of optimization steps provides insight into the effectiveness of the different optimization steps strategies from Chapter 4. Combining it with loss as a function of time provides insight into how efficient each configuration is.

Loss as a function of steps: Firstly, we consider loss as a function of optimization steps. We expect FSSRS to perform worse in this case, as it considers only a single random direction as opposed to twelve directions of the Compass Search and even more for the Combined Compass Search. Likewise, it is expected that Combined Compass Search will perform better than Compass Search. We ran 10 000 optimization steps for each configuration on dataset D1, resulting in Figure A.1, which can be found in Appendix A. We zoom into two different parts of Figure A.1 in Figure 7.1 and Figure 7.2. Our expectations are initially validated in Figure 7.1. Combined Compass configuration performs slightly better than Compass, while FSSRS performs worse. FSSRS is however not too far behind. Choosing a random direction seems to compete well and after around 600 optimization steps it catches up to the others. Note that the Compass configuration is completely equivalent to the Compass R-tree configuration. This is expected as the R-tree purely affects efficiency and not how solutions are searched for.

Considering Figure 7.2 showing the rest of the optimization steps, FSSRS comes to dominate the other configurations as the number of steps increase. It seems that a random direction is better than any of the fixed solutions the others provide. A hypothesis as to why this is the case, is that it is a result of random directions having more "freedom" as they allow for search directions not available to the Compass configurations. Additionally, Compass Search performs many violating steps before it decreases its step size, thus many optimization steps are wasted.

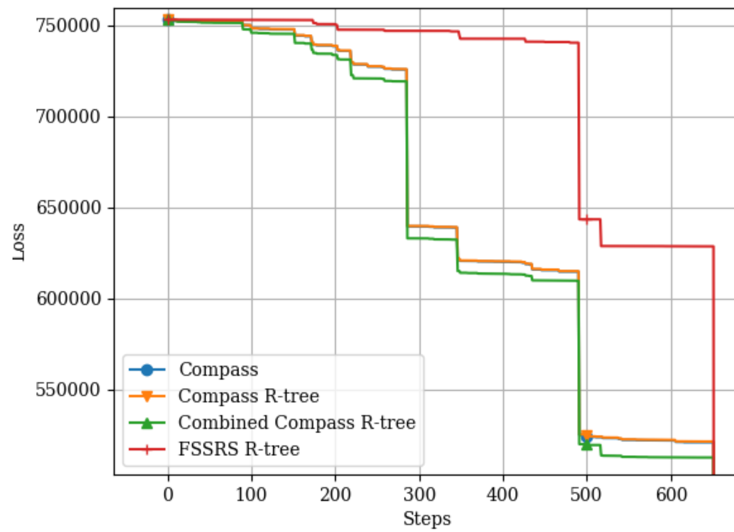

Figure 7.1: All configurations loss as a function of optimization steps on dataset D1. This plot is a cutout of Figure A.1

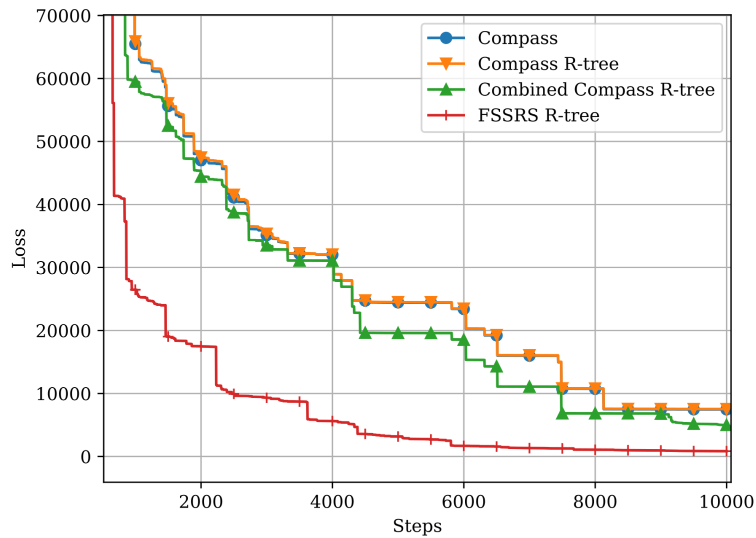

Figure 7.2: All configurations loss as a function of optimization steps on dataset D1. This plot is a cutout of Figure A.1

Loss as a function of time: From what we have seen above, the steps used in the FSSRS configuration work surprisingly well in terms of average loss decrease over the number of steps tested compared to the other configurations. As it only considers a single direction, it is also expected to be the most efficient. Consequently, FSSRS is expected clearly to outperform the other configurations when run over a fixed period of time. Figure 7.3 illustrates loss as a

function of time for all four configurations on dataset D1. The illustration clearly shows FSSRS to perform best. It also shows, that the small improvement the Combined Compass Search achieved in each step is neglected by the extra computation time as the simpler Compass R-tree configuration outperforms the Combined Compass R-tree configurations. Lastly, the Compass R-tree configurations shows a significant improvement in using R-trees for constraint checking.

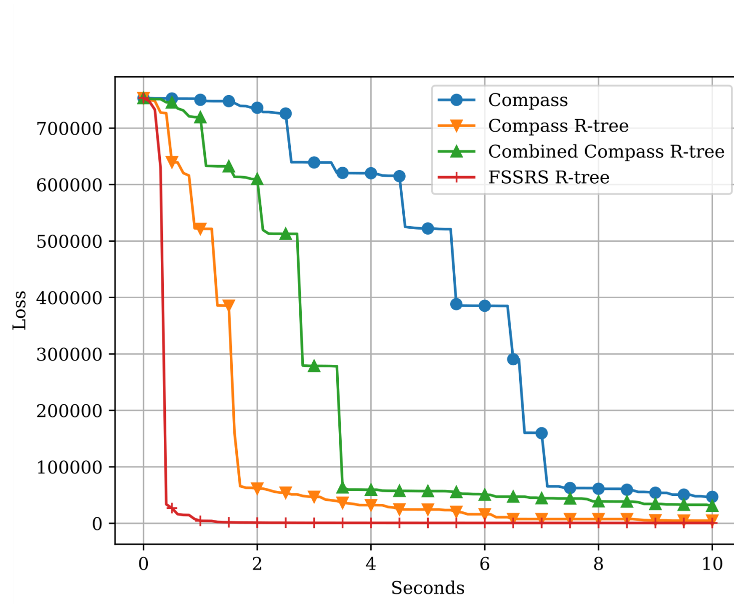

Figure 7.3: Loss as a function of time on dataset D1 shown for all described configurations. Lower loss is better.

After just one second it is hard to see further improvement of FSSRS, thus we also computed loss as a function of time on dataset D2. The illustration in shown in Figure 7.4.

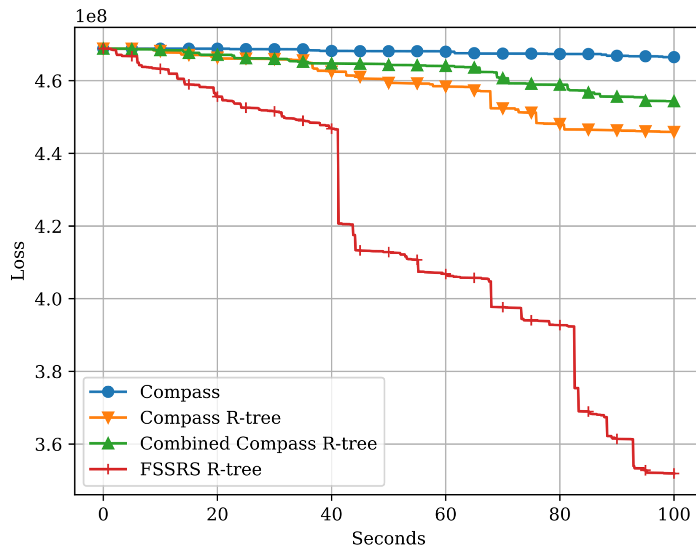

Figure 7.4: Loss as a function of time on dataset D2 shown for all described configurations. Lower loss is better.

The result of D2 shows similar results, but the final difference in loss between the different configurations becomes much clearer. Note that the illustration is merely a comparison of the configuration, thus it grants no explanation regarding how the configurations compare to the optimal solution.

7.2 Behavior of Area and Curvature

Recall that the loss function as defined in Equation (3.7) is a combination of a curvature metric and an area metric. In this section we are interested in examining how their contribution to the loss function change as a function of the number of optimization steps of the FSSRS configuration.

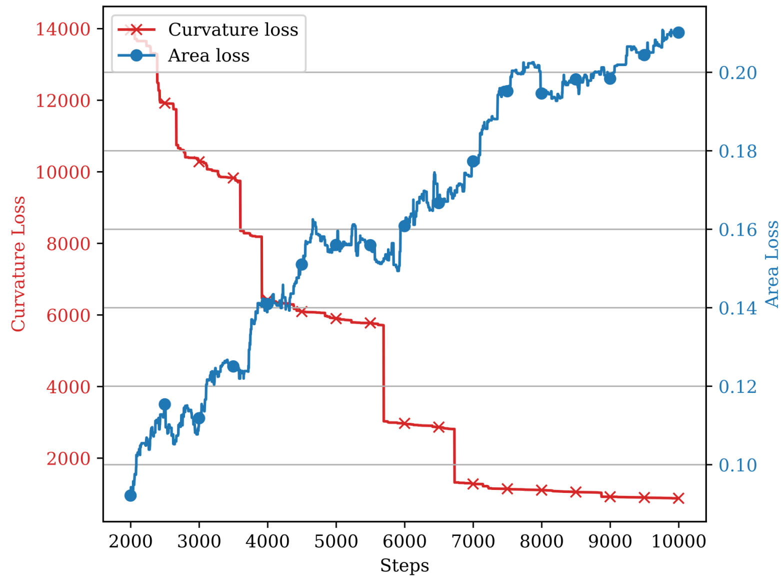

Figure 7.5: Change in area loss and curvature loss of FSSRS R-tree on dataset D1 over 10 000 optimization steps.

Figure 7.5 illustrates the curvature loss compared the the area loss of FSSRS over 10 000 optimization steps. The illustration reveals how the area is compromised for the benefit of curvature. Curvature loss decreases throughout the whole optimization while area loss increases. This makes perfect sense considering the different scales of the two. Values of curvature loss are larger than values of area loss by quite a margin. This is not necessarily poor behavior as curvature by definition of the loss function is more important than area. Recall that we can change how much area "matters" to the loss function by changing the so-called area score we defined in Section 3.4. We can look at an example to see how curvature is prioritized over area in Figure 7.6. Considering the spline in the lower left corner, which is an example of area not being as important. The spline can certainly be improved to contain more area without compromising the curvature.

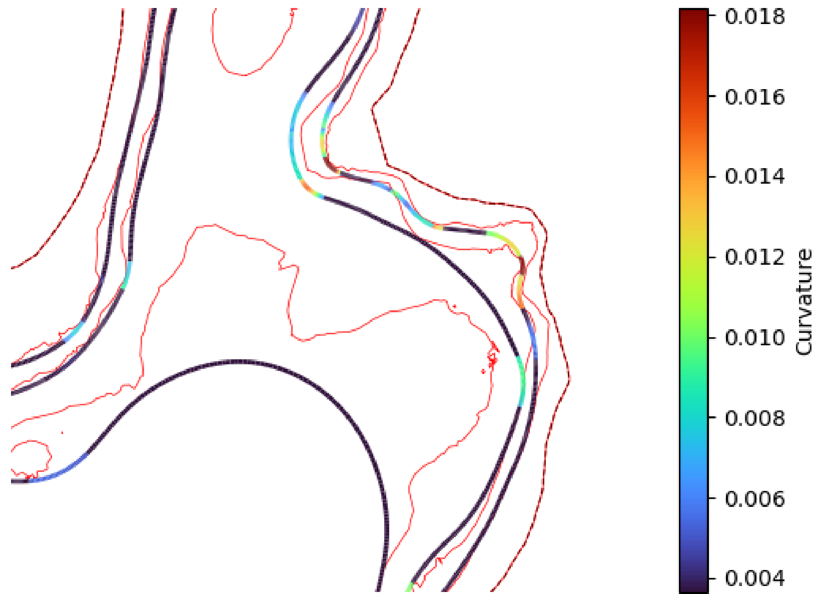

Figure 7.6: Part of a solution of 10 000 optimization steps using FSSRS with chart scale 2cm^2. The colors indicate curvature higher than the target of 0.0036

If the optimization were run for a longer time, it would eventually conquer this area, as can be seen when comparing to Figure 6.3, where the optimization is run 10 times longer for a total of 100 000 optimization steps. This indicates that the FSSRS configuration quickly finds a reasonable solution, but it is slow at getting the fine details correct. Our guess as to why this happens is because there are fewer ways to improve the search directions as better solutions are found. For example, the initial contour can be improved in many ways, making it more likely to randomly choose a search direction which results in lower loss. However, with an optimized solution that has only a few possible improvements left, the chances of randomly finding an improving search direction are low.

If more emphasis on area is desired, a change of area score is sufficient. We have run the optimization with other area scores, the result of which can be found in Figure B.1 and Figure B.2 in the appendix.

7.3 Hyperparameter Optimization

In Chapter 4 we presented three optimization methods for black box optimization used for this project: Compass Search, Combined Compass Search, and FSSRS. Common is that all require some initial step size parameter. This section describes the process of determining this parameter for the FSSRS configuration. We choose to focus on FSSRS as this seems to perform best from the results in Section 7.1. Determining such parameters for optimization is often referred to as *hyperparameter optimization*.

For our problem, an optimization step changes one or multiple properties of a spline point. As described in Section 6.3 our algorithm uses a vector of step sizes $\alpha \in \mathbb{R}^5$. Recall the spline

point representation

$$p = (pos_x, pos_y, angle', out, in) .$$

Each entry of α corresponds to a step size for each entry in p. pos_x and pos_y are closely coupled as well as *out* and *in*, thus we use the same scalar values for these. For the scalar values *position*, *scale* and *angle* we have

$$\alpha = (position, position, angle, scale, scale).$$

In order to discover as suitable α, we deploy a simple random search approach. This time we use a model-based approach as opposed to the instance-based approaches used in the context of the actual optimization algorithm. For the difference between model-based and instance-based, we refer to Section 4.2. The model-based random search simply samples random step sizes in some distribution and computes the loss for each. Inspection of the results hopefully yields greater insight into how different step sizes suit the problem. *position* and *scale* are chosen uniformly at random in the interval $[0\text{mm}, 1\text{mm}]$ in chart scale and the angle step size in $[0, 2\pi]$. Using this distribution, the optimization was run on D1 752 times with 10 000 optimization steps. Recall that D1 has a chart scale 1 : 50000, thus 1 millimeter corresponds to 50 meters. The result is illustrated in a parallel coordinate plot in Figure 7.7.

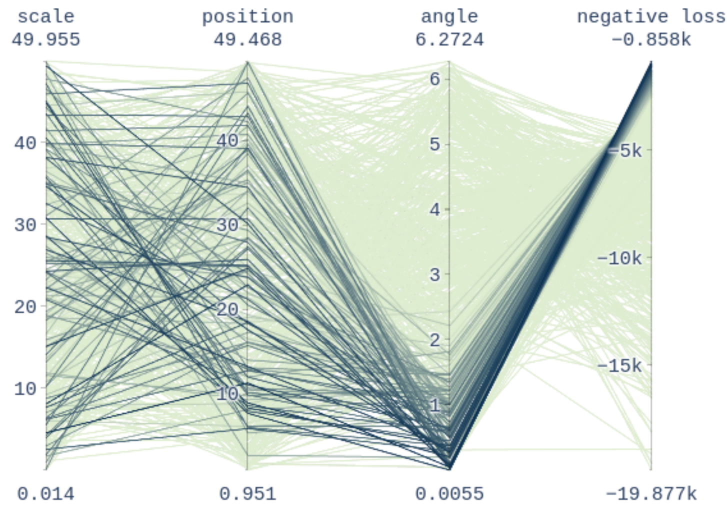

Figure 7.7: A parallel coordinate plot combining randomly chosen hyperparameters for the FSSRS R-tree configuration and their resulting loss on dataset D1. Dark green lines represent the best combinations, as they result in a lower loss (the loss is negated thus larger values are better).

The parallel coordinate plot connects a combination of hyperparameters by lines. This line is then colored based on the loss that the combination of hyperparameters provided. The combinations resulting in a low loss are illustrated with a dark green color. From the illustration, it is clear that lower values of *angle* provide better results. It is not as clear for *position* and *scale*. Many different combinations seem to work, but no certain conclusion is made. Thus we investigate this further. From the precise data, an *angle* of around 0.05 provided the best results. Fixing the *angle* to 0.05 we ran the optimization again. The result is illustrated in the scatter plot in Figure 7.8. A *position* step size between 10 and 20 meters ($[0.2\text{mm}, 0.4\text{mm}]$ chart scale) yields best results. The *scale* step size seems to have no significant effect. As such, we choose the following parameters: $scale = 0.1\text{mm}$, $position = 0.3\text{mm}$, and $angle = 0.05\text{rad}$.

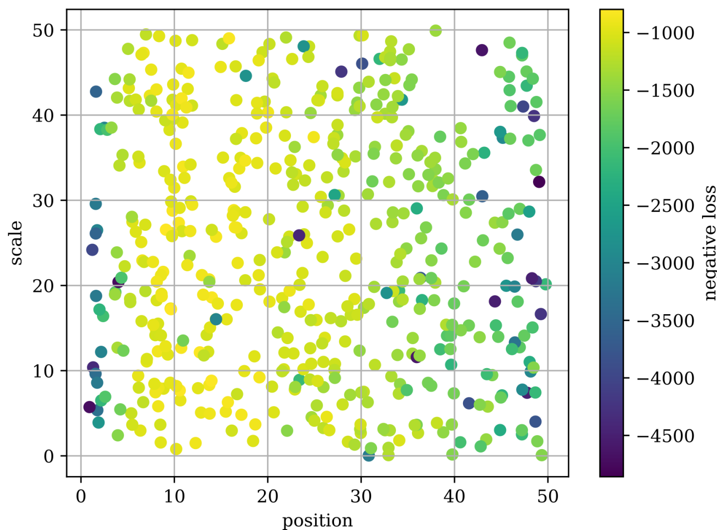

Figure 7.8: Random combinations of position step size and scale size step size given a fixed angle step size of 0.05. Points colors of negated loss thus yellow colors represent the best combinations.

7.4 R-tree Performance

In this section we present the benchmarks done of our Hilbert R-trees usage for improving constraint checking efficiency. The results are most interesting when presented in relation to the size of the different datasets. As such, Table 7.1 shows how many line segments and Bézier curves the datasets contain. Note that line segments are solely used for accurate contours, and Bézier curves for the optimized contours.

	No. Line Segments	No. Bézier Curves
D1	7 885	360
D2	394 004	15 924
D3	418 336	30 514

Table 7.1: Amount of line segments in the accurate contour and Bézier curves in the optimized contours of each dataset D1, D2 and D3.

Naive: Recall that, in this project, R-trees are used as a filtering mechanism. That is, given the MBR of a Bézier curve as query input, the R-tree will return all Bézier curves whose MBR intersect the query rectangle, i.e., the curves that the query curve *potentially* intersects with. To get a meaningful baseline to compare the R-tree implementation with, consider a *naive query* technique for constraint checking in which the MBR of a Bézier curve from the optimized contour is checked against the MBR of all other optimized contours to get the same set of potential intersecting curves. This approach to constraint checking is what we have benchmarked in Table 7.2, by performing 100 queries on randomly sampled curves from the respective dataset. Unsurprisingly, the number of MBR comparisons need is exactly the amount of line segment or curves in the dataset, and the query time is substantial.

	Accurate Contour Naive		Optimized Contour Naive	
	query time	comparisons	query time	comparisons
D1	2.16 ms	7 885	0.14 ms	360
D2	102.77 ms	394 004	5.77 ms	15 924
D3	124.92 ms	418 336	8.49 ms	30 514

Table 7.2: Benchmark data of the naive (baseline) approach for constraint checking. Values are averaged over 100 random queries.

R-tree: Our optimization algorithm stores two trees, one for the accurate contours and one for the optimized. As with the naive query approach, we benchmark by performing multiple queries and averaging. In addition to query time and comparisons, we also measured the average construction time of the R-tree. The construction time is an average over 100 constructions. The query time is an average over one query for each Bézier curve of the initial solution. Each query uses the MBR of the corresponding Bézier curve. The results are shown in Table 7.3.

	Accurate Contour R-tree			Optimized Contour R-tree		
	construction time	query time	comparisons	construction time	query time	comparisons
D1	13.9 ms	0.042 ms	374.5	0.5 ms	0.003 ms	56.7
D2	959.8 ms	0.24 ms	11 157.6	31.8 ms	0.010 ms	860.1
D3	1042.7 ms	0.16 ms	7 596.8	67.9 ms	0.015 ms	1 098.3

Table 7.3: Benchmark data of the Hilbert R-tree data structure for constraint checking.

As expected, the query time and comparison count seem to be correlated. Now we can compare the data from Table 7.2 and Table 7.3. Again as expected, using an R-tree greatly reduces the number of comparisons when compared to checking against all line segments or Bézier curves, which has an impact on the query time. Looking at querying accurate contours in D3 we see that using an R-tree provides a 775 times speedup. For the accurate contours, the slowest construction time is for D3 of approximately one second. As this construction is only run once, this is very efficient. For optimized contours, the construction is even faster, with the slowest being just 67.9ms.

We now turn our attention to the LBUPHR-tree described in Section 5.3. As described in that section multiple reconstructions may be applied to the optimized contour tree while the optimization is running. With a construction time of just 67.9ms, numerous reconstructions are possible in a short amount of time. Table 7.4 shows results of experiments with different sizes of bounding boxes for the LBUPHR-tree storing D1. As ε increases no significant change is visible in query time. The query result size indicates how many geometries the query finds and not surprisingly, this does not change much. However, the amount of reconstruction drops fast. With $\varepsilon = 0.6$ no reconstructions are performed. As reconstruction time is fast a lower ε also suffice.

ε (mm)	0.12	0.24	0.36	0.48	0.6	0.72	0.84	0.96
reconstructions	1125	473	30	2	0	0	0	0
query time (ns)	8556	7764	8301	8424	8517	8376	8462	8193
query result size	3.01	3.04	3.07	3.10	3.16	3.22	3.28	3.36

Table 7.4: Number of reconstructions, query time and query size for different size extensions of bounding rectangles ε in chart scale on dataset D1 with 10 000 optimization steps. Bounding rectangles are increased by ε in each direction. Query time and query size is an average of one query for each Bézier curve in the dataset.

Chapter 8

Conclusion

In this thesis, we set out to explore the problem of optimizing contour lines on bathymetric charts. Concluding this work, we have come to gain an understanding of the context in which the problem arises, as well as extensive knowledge about the problem itself. In order to successfully model the problem as a precise mathematical optimization problem, we looked into a fundamental way of expressing curves digitally, namely using Bézier curves (Chapter 2). We have provided intuition as well as proofs for the most important properties of Bézier curves. Furthermore, we have presented descriptions of algorithms for computing roots, bounding boxes, curvature, area, and intersections of Bézier curves.

Using this knowledge we have modeled a loss function that is used to compare different solutions to each other (Section 3.5). The loss function takes into account the maximum curvature of Bézier curves forming the optimized contour lines, as well as the area of the relatively deeper parts of the map. The optimization algorithm aims to minimize the maximum curvature metric while maximizing the area. We have defined feasible solutions to the problem by use of three constraints namely the depth, intersection, and topology constraints (Section 3.6). We have argued how these, in conjunction, ensure that feasible solutions are true to the underlying data. Finally, we have modeled the problem in standard form by combining the loss function and constraints (Section 3.7).

We have implemented an optimization algorithm for the stated optimization problem. Specifically, given a feasible initial contour map as input the algorithm will produce an optimized (but not necessarily *optimal*) feasible contour chart as output. The algorithm treats the loss function as a black box, and as such we have explored several search heuristics, namely direct search and random search for traversing the solution space (Chapter 4). Furthermore, we have used the R-tree data structure to improve the performance of constraint checking, allowing us to process larger datasets (Chapter 5).

The optimization algorithm can be run in different configurations: with or without the use of R-trees, with direct search or with random search. These configurations are the primary subjects of our benchmarks, which show random search to perform the best by a large margin (Section 7.1). R-trees significantly improved the efficiency of constraint checking and experiments showed the uses of lazy update R-trees to be very efficient (Section 7.4). Visualization of the output of the optimization algorithm as seen in Section 6.5 suggests that the constructed loss function models the goals of the problem description well.

Bibliography

[1] Pankaj Agarwal, Lars Arge, Thomas Mølhave, and Babak Sadri. I/o-efficient algorithms for computing contours on a terrain. In *Proceedings of the Symposium on Computational Geometry*, pages 129–138, 2008.

[2] Pankaj K. Agarwal, Lars Arge, T. M. Murali, Kasturi R. Varadarajan, and Jeffrey Scott Vitter. I/o-efficient algorithms for contour line extraction and planar graph blocking. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 117–126, 1998.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, may 1990.

[4] Wolfgang Boehm and Andreas Müller. On de casteljau's algorithm. *Computer Aided Geometric Design*, 16(7):587–605, 1999.

[5] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. Brooks/Cole, Cengage Learning, Boston, MA, 9 edition, 2010.

[6] M. Daniel and J.C. Daubisse. The numerical problem of using be´zier curves and surfaces in the power basis. *Computer Aided Geometric Design*, 6(2):121–128, 1989.

[7] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, third edition, 2008.

[8] Anthony D. DeRose. *Geometric Continuity: A Parametrization Independent Measure of Continuity for Computer Aided Geometric Design*. PhD thesis, EECS Department, University of California, Berkeley, Aug 1985.

[9] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '89, page 247–252, New York, NY, USA, 1989. Association for Computing Machinery.

[10] Gerald E. Farin. *Curves and Surfaces for Computer-Aided Geometric Design*. Academic Press, 5th edition, 2002.

[11] Peter L. Guth, Adriaan Van Niekerk, Carlos H. Grohmann, Jean-Paul Muller, Lewis Hawker, Igor V. Florinsky, Dean Gesch, Heinrich I. Reuter, Victor Herrera-Cruz, Serge Riazanoff, et al. Digital elevation models: Terminology and definitions. *Remote Sensing*, 13(18), 2021.

[12] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, page 47–57, New York, NY, USA, 1984. Association for Computing Machinery.

[13] Robert Hooke and T. A. Jeeves. " direct search" solution of numerical and statistical problems. *J. ACM*, 8:212–229, 1961.

[14] Boguslaw Jackowski. Computing the area and winding number for a bézier curve. *TUGboat*, 33, 2012.

[15] Camille Jordan. *Cours d'analyse*. 1887.

[16] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. In *Proceedings of the Second International Conference on Information and Knowledge Management*, CIKM '93, page 490–499, New York, NY, USA, 1993. Association for Computing Machinery.

[17] Deok-Soo Kim, Soon-Woong Lee, and Hayong Shin. A cocktail algorithm for planar bézier curve intersections. *Computer-Aided Design*, 30(13):1047–1051, 1998.

[18] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[19] Tamara G. Kolda, Robert Michael Lewis, and Virginia Torczon. Optimization by direct search: New perspectives on some classical and modern methods. *SIAM Review*, 45(3):385–482, 2003.

[20] P.A. Koparkar and S.P. Mudur. A new class of algorithms for the processing of parametric curves. *Computer-Aided Design*, 15(1):41–45, 1983.

[21] Dongseop Kwon, Sangjun Lee, and Sukho Lee. Indexing the current positions of moving objects using the lazy update r-tree. In *Proceedings Third International Conference on Mobile Data Management MDM 2002*, pages 113–120, 2002.

[22] Jeffrey M. Lane and Richard F. Riesenfeld. A theoretical development for the computer generation and display of piecewise polynomial surfaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-2(1):35–46, 1980.

[23] Scott Leutenegger, Mario Lopez, and Jeffrey Edgington. Str: A simple and efficient algorithm for r-tree packing. pages 497–506, 05 1997.

[24] Qi Lou and Ligang Liu. Curve intersection using hybrid clipping. *Computers Graphics*, 36(5):309–320, 2012. Shape Modeling International (SMI) Conference 2012.

[25] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Yannis Theodoridis. *R-Trees: Theory and Applications*. Advanced Information and Knowledge Processing. Springer London, first edition, 2006.

[26] A. Mutseniyeks and L. A. Rastrigin. Extremal control of continuous multi-parameter systems by the method of random search. *Engineering Cybernetics*, 1:82–90, 1964.

[27] Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 1999.

[28] L. A. Rastrigin. The convergence of the random search method in the extremal control of a many parameter system. *Automation and Remote Control*, 24(11):1337–1342, 1963. 1964 translation of Russian Avtomat. i Telemekh pages 1467–1473.

[29] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed r-trees. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, SIGMOD '85, page 17–31, New York, NY, USA, 1985. Association for Computing Machinery.

[30] Hans Sagan. *Space-Filling Curves*. Universitext. Springer New York, NY, 1 edition, 1994. Published: 02 September 1994 (softcover), 06 December 2012 (eBook).

[31] MA SCHUMER and Kenneth Steiglitz. Adaptive step size random search. *Automatic Control, IEEE Transactions on*, AC13:270 – 276, 07 1968.

[32] Thomas W. Sederberg. *Computer Aided Geometric Design*. January 2012.

[33] Thomas W Sederberg and Scott R Parry. Comparison of three curve intersection algorithms. *Computer-Aided Design*, 18(1):58–63, 1986.

[34] T.W. Sederberg and T. Nishita. Curve intersection using bézier clipping. *Computer-Aided Design*, 22(9):538–549, 1990.

[35] M. Shimrat. Algorithm 112: Position of point relative to polygon. *Commun. ACM*, 5(8):434, aug 1962.

[36] M.R. Spiegel. *Mathematical Handbook of Formulas and Tables*. Schaum's outline series. McGraw-Hill, 1990.

[37] Oswald Veblen. Theory on plane curves in non-metrical analysis situs. *Transactions of the American Mathematical Society*, 6(1):83–98, 1905.

[38] G Wang. The subdivision method for finding the intersection between two bézier curves or surfaces. *Zhejiang University Journal, Special Issue on Computational Geometry (in Chinese)*, 1984.

[39] Zelda B. Zabinsky. *Random Search Algorithms*. John Wiley Sons, Ltd, 2011.

[40] M. Zlochin, M. Birattari, N. Meuleau, and M. Dorigo. Model-based search for combinatorial optimization: A critical survey. *Annals of Operations Research*, 131:373–395, 2004.

Appendix A

Loss over 10 000 Steps

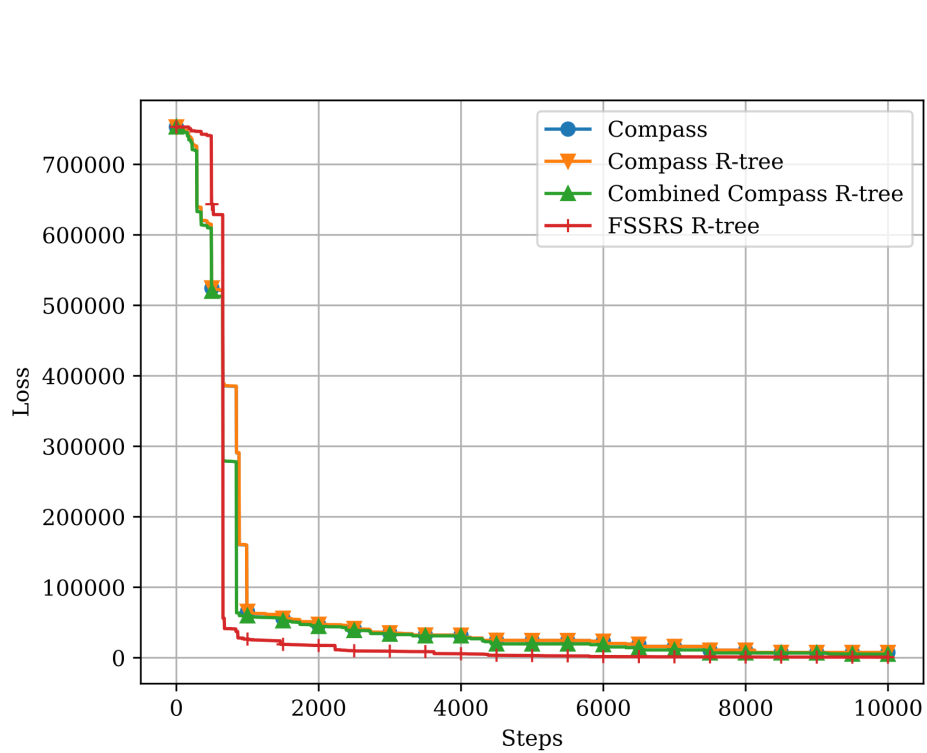

Figure A.1: All configurations loss as a function of optimization steps on D1.

Appendix B

Curvature vs. Area

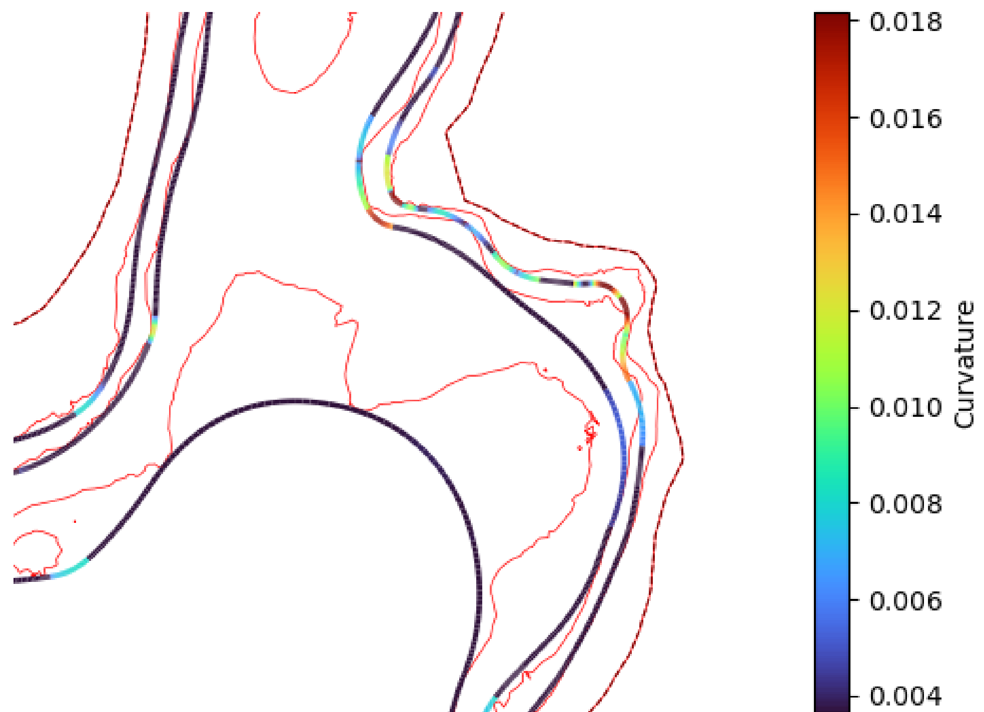

Figure B.1: Part of a solution of 10 000 optimization steps using FSSRS with area score $\frac{2}{30}\text{cm}^2$. The colors indicate curvature higher than the target.

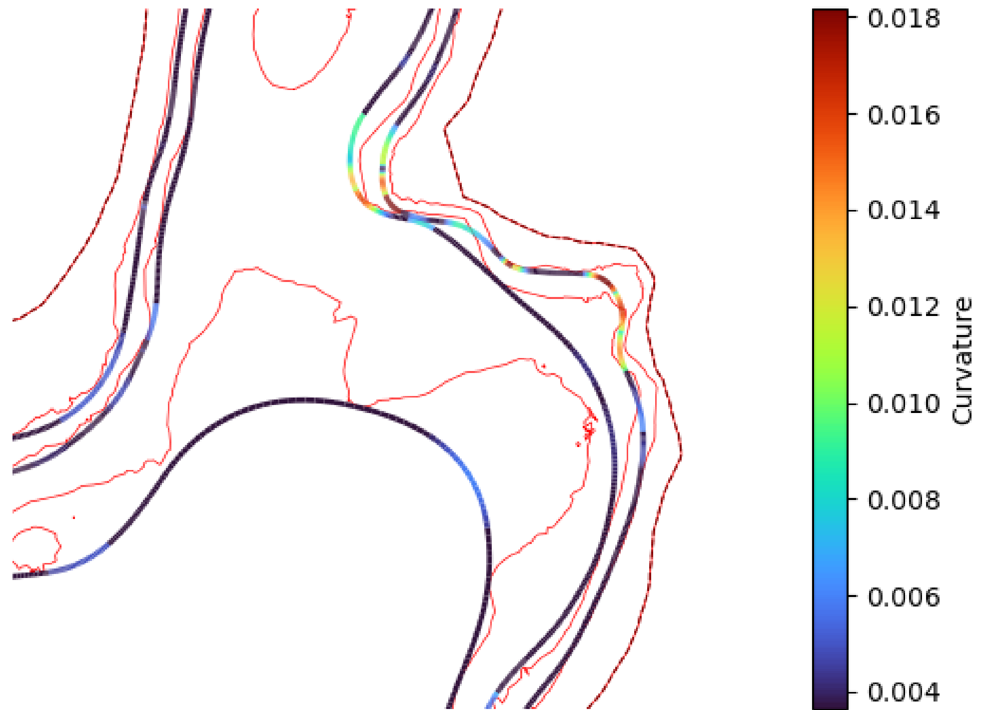

Figure B.2: Part of a solution of 10 000 optimization steps using FSSRS with area score $\frac{2}{60}\text{cm}^2$. The colors indicate curvature higher than the target.

Appendix C

Visualization of Datasets D2 and D3

Figure C.1: D2 after 150 000 optimization steps. Note that it is not displayed in its true chart scale as it would otherwise not fit the page. This makes contours appear with higher curvature.

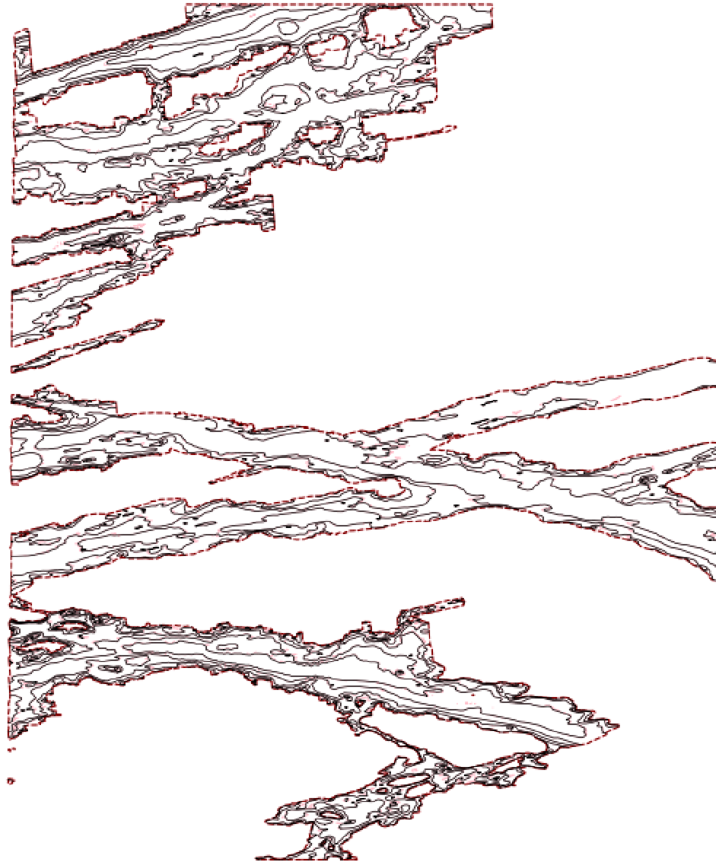

Figure C.2: D3 after 150 000 optimization steps. Note that it is not displayed in its true chart scale as it would otherwise not fit the page. This makes contours appear with higher curvature.

Figure C.3: Part of D3 after 150 000 optimization steps.