DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Robotics, Cognition, Intelligence

Implementation of an Iterative Sampling-Based Local Path-Planner

Roman C. Podolski

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Robotics, Cognition, Intelligence

Implementation of an Iterative Sampling-Based Local Path-Planner

Implementierung eines iterativen sampling-basierten lokalen Pfad-Planners

Author:	Roman C. Podolski
Supervisor:	Prof. Dr.-Ing. Klaus Diepold
Advisor:	Prof. Dr. rer. nat. habil. Thomas Bräunl
Submission Date:	December 15, 2017

I confirm that this master's thesis in robotics, cognition, intelligence is my own work and I have documented all sources and material used.

Munich, December 15, 2017 Roman C. Podolski

Acknowledgments

I want to thank my advisor and mentor Prof. Dr. rer. nat. habil. Thomas Bräunl and my supervisor Prof. Dr.-Ing. Klaus Diepold for their ongoing support and guidance. They deserve my special thanks for providing me with the opportunity to conduct this research at the University of Western Australia (UWA).

I want to thank the REV-Team at UWA for their splendid collaboration. Their imperturbable motivation in the face of challenging problems contributed to the success of this work.

This work would not have been possible without the financial support of the Deutscher Akademische Austauschdienst e. V. (DAAD) and the FITweltweit Scholarship program. I feel honored to hold this scholarship and hope that I can return some of its worth with this work.

I thank Ms. Simone Wenig and the team of the TUM-Mentoring program for bringing people together. I was not expecting that I would travel to Australia to write my thesis when I signed up for this program.

Thanks go to Chris Kahlefendt for taking the time to discuss my work and sharing his knowledge.

I would like to thank my friends Lauren Smith and Sam Evans-Thomson for allowing me into their home and enduring me during writing. Thanks for being friends when I needed some.

I must express my profound gratitude and respect for my parents for providing me with the unfailing support and continuous encouragement through years of study. I thank my mother for teaching me to never back down in the face of difficulties - you gave me the power to endure everything by merely believing in me. I thank my father for teaching me that no dream is too big or complicated to be pursued if one approaches a problem a single step at a time - you led by example and showed me that I could be whatever I want. This accomplishment would not have been possible without them. Thank you.

Finally, I would like to give my thanks to my partner, Christin Caliebe, who accompanied me in this undertaking: You traveled half the world with me so that I could pursue my dream. I'm genuinely grateful for your support, loyalty, and company. Thank you for being with me all this time.

Abstract

This work presents an implementation of an iterative sampling-based local path planner in the object-oriented programming language C++ with the modern ISO standard of 2014 (C++14). It aims to solve the planning problem for an application in autonomous racing. The targeted hardware is an electronic racecar built at the University of Western Australia which is intended to participate in a race similar to the Formula Student Driverless. Since the planner must run efficiently on embedded hardware, this work favors a sampling-based near-optimal solution over a computationally more intensive optimal solution. Further, this thesis introduces resource economic approach to collision checking. The implementation of the algorithms is required to be perormant and easily integrable in the robotic framework of the racecar. Therefore, C++14 is used to implement the algorithms, since it provides high-performance and modern language features that integrate well with existing software. Simulation verifies the correct workings and performance of the implementation. The results of the simulation prove that the chosen algorithms find a near-optimal solution to the path planning problem for a scenario like the Formula Student Driverless, while allowing update rates as high as 20 Hz. This work provides the base for a practical application of the provided planner in a robotics framework and integration testing in a real-world scenario.

Contents

1. Introduction

This chapter outlines the motivation for this work and discusses the scientific context. It provides a specific and measurable description of the goal that this work achieves and formally describes the problem at hand. The chapter discusses the methods used to attack the problem and concludes with a structural overview of the contents of this work.

1.1. Motivation

In 2015 the Formula-E electric motor racing series announced that it is adding a robotic event as an opening act. Autonomous cars will race before the human competition begins [Lau15]. This ROBORACE[1] aims to be the first autonomous racing series in the world. The vast scientific interest in autonomous driving over the past 30 years [Pom89; DMC90] mainly focused on applications in an urban environment. The hopes associated with the topic often focus on the potential of autonomous vehicles to make traffic safer, while being more comfortable for the driver and passengers [Fis+16]. ROBORACE shows that there is also a possible entertainment aspect to autonomous driving. The nature of motorsport inspires technological advances to set teams ahead of their competition. It allows testing of new technologies in a safe environment while pushing the boundaries of the possible. Findings produced from sports often make their way back into the casual driving domain. Technological standards in motorsport define the future of transportation. For example, the rear spoiler was introduced in 1960 in the racing domain and can be found on many consumer cars today. The DARPA grand challenge 2006 has shown the accelerative impact of a competitive event on science [Thr+06; Chr+08] and inspired other events of its kind. The Formula Student Germany (FSG) and Formula Society of Automotive Engineers (SAE) are international design competitions for students, where the teams develop a business plan and prototype for a Formula-style race car. FSG and Formula-SAE provide the opportunity for students to apply their knowledge in a real-world scenario and for companies to find a new generation of motivated and skilled automotive designers and engineers. In August 2017, for the first time, the FSG held a new competition class: Formula Student Driverless (FSD) [Lau17a]. This future-oriented event confronts students with the challenge to develop a race car that can run without a driver in autonomous mode, or with a driver in manual mode. Akademischer Motorsportverein Zürich (Academic Motorsport Society Zurich) (AMZ), the winning team of this event,

[1] https://roborace.com/

proved that such a task is feasible for teams of students [Lau17b]. The car developed by this team was the only car to finish the race.

The FSD inspired members of the Renewable Energy Vehicle (REV) Project at UWA to convert a racecar that was built for the Formula-SAE competition to perform a task similar to that required by the rules of the FSD: The car should autonomously maneuver a track delimited by traffic cones while avoiding collisions with static obstacles. This work modifies the rules of the FSD to an extent and assumes that the racetrack may be driven once manually, before performing the autonomous lap (see section 1.3).

The vehicle that served as research platform was converted by former members of the REV-Project, first to drive-by-wire navigation [Dra13] and later extended to incorporate static obstacle avoidance [Fre14; Chu15]. While those works provide a base for development, the implemented software lacked documentation and flexibility. Nor was it possible to perform software in the loop (SIL) or hardware in the loop (HIL) testing and simulation. The software is implemented in a very procedural fashion and lacks an extendible, object-orientated and modular architecture. The approach to software design is a monolithic one, where one class monopolizes the processing, and other classes primarily encapsulate data. Monolithic design is a known software-development anti-pattern that results in a limited ability to modify parts of the system without affecting the functionality of other encapsulated objects [SFP17]. Modifications on other, logically unrelated modules are likely to have an impact on the central processing monolith of the software. The outcome of this architecture and the lack of documentation or examples is that one needs to undertake modifications on the existing software with excessive caution, but is likely to nevertheless cause hard to trace errors in the system. The implementation of the path planning was one of the modules that suffered severely from this. It was not possible to deduce from the existing software architecture which parts of the program are related to path planning, and which perform separated tasks. A cause of this problem was that former work used MATLAB[2] to prototype the implementation of the path planner. The MATLAB-Coder was used to generate C/C++-sources from the prototype. While this made it possible to integrate the prototype with the existing software, it added the complexity of another programming language. The existing software provides a solid base, but requires much attention and rework of crucial parts to provide an extensible research platform for students that wish to focus on autonomous vehicles.

1.2. Problem Description

The problem at hand is defined by the rules of the FSD from 2017: A vehicle must find its way through a racetrack autonomously. Traffic cones delineate the road. Blue cones delimit the left side, yellow ones the right side (see fig. 1.1). The maximum distance between two cones in driving direction will be 5 m, the minimal width of the road is 3.5 m.

[2] https://mathworks.com/products/matlab.html

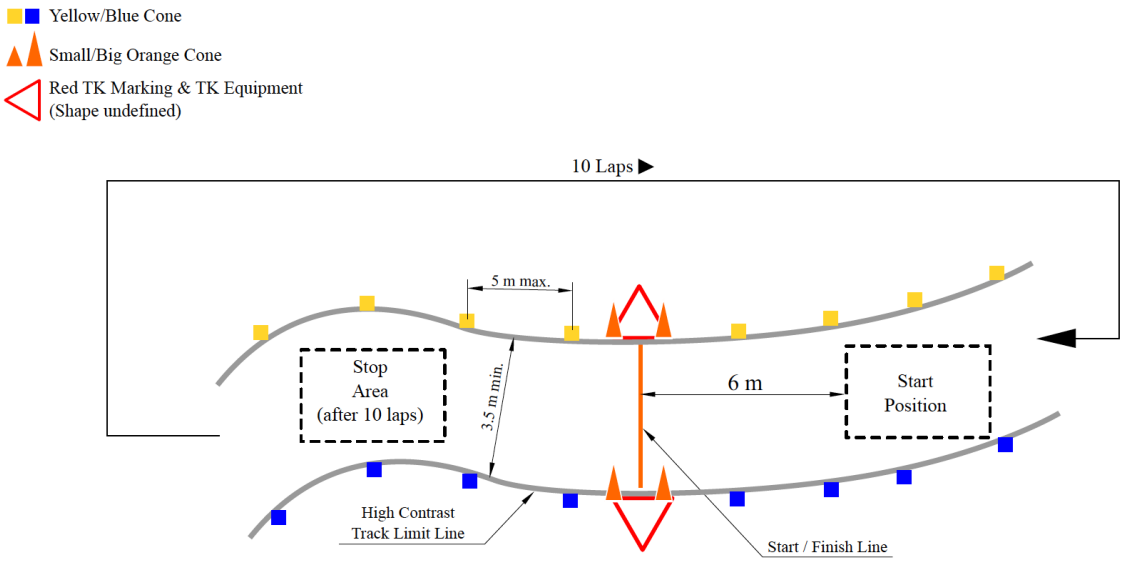


Figure 1.1.: An example scenario for a track drive from the FSD-rules. This illustration shows a representative piece of track that an autonomous vehicle has to follow during the FSD competition. Small traffic cones delimit lane markings: yellow ones for the right side, blue ones on the left side. The cones a maximum of 5 m apart in the driving direction and the track is a minimum of 3.5 m wide at any point. Larger orange cones mark the start- and stop-zones of the racetrack. Participants of FSD have to accomplish ten laps around such a track. (Figure taken from [For17, p. 10])

The Performance, Environment, Actuators, Sensors (PEAS) Description [RN09, Section 2.3.1] provides a way to analyze this problem concerning its environment and the agent. First, one considers the performance measure of the task. The vehicle should make its way around without colliding with obstacles. Collisions with delimiters will result in negative points. Therefore the safety of the path is a measurement of the performance of the agent. The next objective of the vehicle is to make its way around the track and continue to do so for ten laps, without leaving the road. Last, with safety and continuity in mind, the laptime has to be considered. To allow high velocity the car agent needs to drive a smooth line around the track. Therefore, smoothness is the last performance measure of the problem. The environment in which the car will perform its task is layout by the FSD-rules: It is partially observable, since the sensory perception of the surrounding world may be imperfect. There are no other participants on the racetrack; Therefore the environment is a single agent scenario. It is stochastic and sequential because the environment may change during operation (a cone may move or get removed) and actions of the agent are dependent on each other. Finally, the environment is continuous since the possible states of the environment are infinite. The racecar can influence its configuration using actuators on steering, throttle, and breaks. The sensory body contains light detection and ranging (Lidar), a monocular camera, an inertial measurement unit (IMU) and wheel-speed odometer. Lidar and camera can be used to observe the geometrical shape and relative position of obstacles in the workspace. IMU, global positioning system (GPS) and odometer provide information about the absolute position and orientation of the vehicle and obstacles in the workspace. Table 1.1 summarizes the PEAS description of the vehicle and its environment.

Performance	Environment	Actuators	Sensors
safety	partially observable	steering	Lidar
smoothness	single agent	throttle	camera
continuity	stochastic	breaks	GPS
	sequential		odometer
	continuous		IMU

Table 1.1.: PEAS-Description (Performance, Environment, Actuators, Sensors) of the FSD[RN09, p. 2.3.1].

An agent that aims to achieve this task needs to perform several steps as a hierarchy of its decision-making process. First, finding a global route through the track. A precondition for this step is the accumulation of map data. A global map may be provided by manual measuring or simultaneous localisation and mapping (SLAM)[Thr02, Chaper 10] during an exploration drive around the racetrack. However, SLAM in this scenario

Figure 1.2.: Photo of the electric race car built by the REV-Project. The chassis of the car is in line with the rules of the Formula-SAE [SAE17]. Former members of the team converted the motors from combustion units to electrical powered, implemented drive-by-wire functionality, and way-point navigation [Dra13]. An Ibeo Lux automotive Lidar, IMU, GPS, monocular camera and wheelspeed odometry serve as the sensory body of the vehicle. Electronic actuators control steering, breaks, and acceleration.

is out of the scope of this work and maybe the subject of future publications. The next layer in the decision-making hierarchy produces behavior, based on the perception of the environment. At the behavioral layer, the car decides if it should drive because it is performing a lap or if it should stop because it achieved its objective. The behavioral layer for the task at hand is trivial, and not subject of this work. The third stage of the hierarchy plans the local motion for the behavior from the top layer. A feedback controller executes the output of this layer, by estimating the vehicle state and correct throttle, steering and breaking to follow the path from the motion planner. This work primarily focuses on motion planning.

The problem of motion (or path-) planning in robotics can be expressed as finding a *feasible* path $\boldsymbol{\tau}(\alpha) : [0, L] \to \mathcal{C}$ in the configuration space from an initial configuration $\boldsymbol{x}_{\text{init}}$ to a goal section of the configuration space $\mathcal{C}_{\text{goal}}$. Feasibility in this context means that the robot can dynamically perform the motions to follow the path. It does not focus on the quality of the solution but expresses that a path satisfies some given problem constraint. An infeasible path for a car-like robot, for example, would be to move sideways without any forward motion. Robots that are constrained in this way are referred to as *non-holonomic*. *Optimality* in motion planning refers to the problem of finding a path that optimizes some quality criterion with a subject to given constraints. Therefore optimality includes feasibility, but not vice versa.

The problem of optimal path planning can be formally stated: Let $\mathcal{C}$ be the configuration space of the vehicle and let $\Sigma(\mathcal{C})$ be the set of all continuous functions $[0, L] \to \mathcal{C}$ in the configuration space of the vehicle (or robot). The initial configuration of the robot is $\boldsymbol{x}_{\text{init}} \in \mathcal{C}$. The path is required to end in a goal region $\mathcal{C}_{\text{goal}} \subset \mathcal{C}$. The set of all allowed configuration of the vehicle is called the free configuration space $\mathcal{C}_{\text{free}}$. Free configurations are those that do not result in collisions with obstacles and meet the holonomic constraints on the path. Differential constraints on the path are expressed by the predicate $D(\boldsymbol{x}, \boldsymbol{x}', \boldsymbol{x}'', \dots)$. Further, let $J(\boldsymbol{\tau}) : \Sigma(\mathcal{C}) \to \mathbb{R}$ be the cost (or objective) function. An optimal path minimizes the cost. Now, the problem of optimal path planning can be stated as follows:

Problem definition (Optimal path planning):
Given a 5-tuple $(\mathcal{C}_{\text{free}}, \boldsymbol{x}_{\text{init}}, \mathcal{C}_{\text{goal}}, D, J)$ find $\boldsymbol{\tau}* =$

$$\begin{aligned}
&\underset{\boldsymbol{\tau} \in \Sigma(\mathcal{C})}{\arg\min} && J(\boldsymbol{\tau}) && && (1.1)\\
&\text{subj. to} && \boldsymbol{\tau}(0) = \boldsymbol{x}_{\text{init}} \text{ and } \boldsymbol{\tau}(L) \in \mathcal{C}_{\text{goal}} && && (1.2)\\
& && \boldsymbol{\tau}(\alpha) \in \mathcal{C}_{\text{free}} && \forall \alpha \in [0, L] && (1.3)\\
& && D(\boldsymbol{\tau}(\alpha), \boldsymbol{\tau}'(\alpha), \boldsymbol{\tau}''(\alpha), \dots) && \forall \alpha \in [0, L] && (1.4)
\end{aligned}$$

This problem is known to be PSPACE-hard, which means that it is at least as hard as solving any NP-complete problem and thus, assuming $P \neq NP$, there is no efficient (polynomial-time) algorithm that can solve all instances of the problem[Pad+16]. A

near-optimal solution to the problem reduces the computational load drastically.

1.3. Goal

The goal of this work is to implement a local path planner that finds a near-optimal solution for the path planning problem under the rules of the FSD [FS 17]. A map can be measured beforehand, manually or through SLAM during an exploration lap.

The target hardware that should perform this task is an electric race car built by members of the REV-Project (see section 1.2). The implementation will be designed with extensions and integration into a generic software framework in mind, but decoupled from the surrounding software architecture. The aim is to provide a well designed software module which can be easily integrated in a framework and fulfills the standards of modern software development. Documentation and example code will be provided along with the implementation. This allows students that wish focus their effort on motion planning for autonomous racing, to use this work as a basis or starting point for their research. SIL-Simulation will serve as a form of measurement for the performance of the provided implementation.

1.4. Methodology

Several methods can be used to attack the path planning problem. [Pad+16, Table 1] provides a rich overview and analysis of popular motion planning methods. Unfortunately, an exact algorithm with practical computational complexity is unavailable. Therefore, one has to resort back to numerical methods. Those methods are not exact but attest to find a satisfactory solution, or a sequence of feasible ones that converge to the optimal path. One can broadly divide those numerical methods into three categories:

Variational methods view path planning as a non-linear continuous optimization problem. A function parametrized by a finite-dimensional vector represents the path, and the solution is found by non-linear optimization. If an analytical solution of the problem can be obtained, it is optimal. However, more often only a numerical solution for the problem is available. Those methods are attractive because of they converge fast, but tend to get stuck in local minima for poor initial guesses. Pontryagin's maximum (or minimum) principle (PMP) is an example of a variational method.

Graph-search methods discretize the configuration space of the vehicle in the form of a graph. A finite set of vertices represent a collection of vehicle configurations, the edges represent the transitions between the configurations. Discrete search (e.g., with Dijkstra's algorithm) can find the desired path for a minimum cost. Graph-search methods require a construction of obstacles in the configuration space, which is a costly operation and not always possible. Probabilistic Roadmaps (PRMs) are a classic example of graph-search methods.

Incremental-search methods probe the configuration space with discrete samples and incrementally build a reachability graph. The graph maintains a discrete set of configurations and possible transitions between them. When the graph is big enough to contain at least one node in the goal-region, the desired path is obtained by backtracking. In contrast to incremental search methods, sampling-based methods incrementally increase the size of the graph until finding a satisfactory solution. Sampling-based incremental search methods allow online planning, while the vehicle executes parts of the solution that the method obtained during previous iterations. It is therefore very well suited for partially observable environments and time-critical applications, like racing. Rapidly Expanding Random Trees (RRTs) are an example of an incremental-search method.

While it is possible to exploit the advantages of those methods by combining them, this work will use a sampling-based incremental search approach [Lav06, Chapter 5]. Sampling-based planning algorithms avoid the explicit construction of $\mathcal{C}_{\text{obs}}$. Instead, a search probes the configuration space with a sampling-scheme. The probing is enabled by a collision-detection module that the motion planner can treat as a *black-box* between $\mathcal{W}$ and $\mathcal{C}$. Figure 1.3 illustrates this philosophy. RRTs are a well known sampling-based incremental-search method. However, RRTs are computationally expensive by their nature. Therefore they are neglected in favor of a more efficient geometrical solution. The assumption that the global route is provided as a set of discrete waypoints allows using an approach similar to [CLS12].

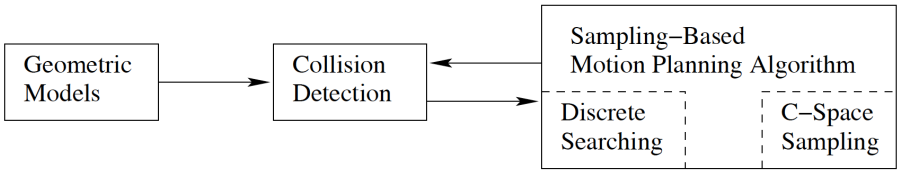


Figure 1.3.: Sampling-based planning uses collision detection as a *black box* that separates motion planning from the geometric and kinematic models. $\mathcal{C}$-sampling and discrete searching are performed. (Figure taken from [Lav06, p. 185])

With the method chosen, one has to decide what kind of tools to use to implement it. Section 1.3 declares that one of the goals of the implementation is the easy integration in a robotic software framework that drives the REV-racecar. The existing software framework is implemented using the programming languages Python, C, C++, Perl, and Bash. The scripting languages Perl and Bash are inappropriate for numerical programming, and therefore they are unsuited to implement the motion planning routines. The implementation should meet modern standards of software development. It should be extensible, well designed and performant. While C, C++, and Python are languages that are widely used in robotics, they serve different purposes. C serves a high-performance domain but lacks object-orientation. The software design in C tends to be clunky, unclear and often hard to support or to extend. Python is very convenient for mathematical programming but is an interpreted language. The price for

the convenience that Python offers therefore is performance. The C++ ISO standards of 2014 offer a language that is very high performing (in some cases even more than C) but supports many of the language extensions that make Python so convenient to use. Further, the language is object-oriented and allows sophisticated software design which supports flexibility and readability. This work uses C++ for non-linear numerical object-oriented programming and iterative discrete sampling-based motion planning to solve the path planning problem described in 1.2.

1.5. Structure

The remaining part of this thesis is structured as follows: Chapter 2 explains the fundamental algorithmic and mathematical basis of the presented path-planner. It introduces and explains a novel and resource-economic approach to collision checking. Chapter 3 discusses the implementation. It explains the design rationale, introduces software libraries and tools that were used to implement the planner. Further, it discusses specific algorithmic characteristics and code-fragments that illustrate the use of unique language features and software libraries. Chapter 4 presents the result that where achieved with the provided implementation. It introduces the simulation which was used to produce the results, discusses the influence of the objective function and design parameters on the solution and benchmarks the runtime performance of the program. Further, the achieved results are discussed and compared to other results from the literature. Chapter 5 concludes this thesis. It discusses possible future work, and finally summarizes the thesis.

2. Basics

This chapter explains the fundamental basics of the provided local path-planner. The underlying mathematical principles are introduced and discussed. Section 2.1 gives a brief overview of the used algorithm. The following sections 2.2 to 2.7 are dedicated to the intermediate steps of the planning process. After all principles have been explained in detail, section 2.9 formally defines the algorithm.

2.1. Overview

The presented algorithm takes an iterative sampling-based approach to the problem of motion planning [Lav06, Chaper 5.]. It is based on the works of [Fre14; CLS12; Li+16]. The idea presented in those works is extended by a novel approach to efficient collision checking in the $\mathcal{W}$-space.

As described in Section 1.3 the objective of the presented path planner is to find a feasible and near-optimal path τ from a starting configuration $\boldsymbol{x}_{\text{init}}$ to the goal-region $\mathcal{C}_{\text{goal}}$. The notion of feasibility in this context means that the path τ has to lie entirely in $\mathcal{C}_{\text{free}}$, so it must not collide with any obstacles in $\mathcal{W}$ and satisfy differential constraints D of a non-holonomic robot. Optimality expresses that the path provides the best solution concerning an objective function $J : \Sigma(\mathcal{C}) \to \mathbb{R}$.

The intermediate steps that are taken by the motion planner to achieve this goal are as follows:

1. Construction of a spline-curve to connect given waypoints $\boldsymbol{p}_i$ and find a continuous path from $\boldsymbol{x}_{\text{init}}$ to $\mathcal{C}_{\text{goal}}$.

2. Construction of a curvilinear coordinate system for which the spline curve serves as a baseframe.

3. Localisation of the vehicles position x_{init} in the curvilinear coordinate system.

4. Generation of N path-samples as cubic-polynomials in the curvilinear coordinate system.

5. Transformation of the paths from step 3 to the local Cartesian coordinate system by forward-integration of a kinematic model.

6. Checking the paths for feasibility and discarding paths that collide with obstacles or violate non-holonomic constraints.

7. From the remaining paths returning the one that minimizes the objective function J. If no feasible path could be found return the path that travels the longest distance without collision.

8. Repeat steps 3-7 while the vehicle moves until the goal-region is reached.

From those intermediate steps one can derive the requirements of the algorithm:

1. The map of the track has to be provided beforehand, as a set of 2D-waypoints in local Cartesian coordinates.

2. The environment in which the vehicle is operated must be presentable as a planar projection to local Cartesian coordinates.

3. The geometrical shape and position of obstacles must be measurable by higher layers of the decision making hierarchy of the vehicle. Therefore a module to sense obstacles and project them in local carthesian coordinates must be present.

4. The position and orientation of the vehicle must be measurable. This is also a requirement for the feedback controller of the vehicle. Therefore one can assume that this is present in a real world scenario.

2.2. Arc-Length Parametrization of Cubic B-splines

The global map is provided as a set of n discrete waypoints $\boldsymbol{p}_i \in \mathbb{R}^2$ with $i = 0, 1, \ldots, n$. One needs to find a continuous connection between those waypoints. A naïve solution to this problem would be to find a linear interpolation in between the points $\boldsymbol{p}_i$ and $\boldsymbol{p}_{i+1}$. While this solution encodes usable directional information and is continuous, the resulting curve is not continuously differentiable. As shown in [Dra13, section 5.2] the abrupt change in tangent angle of the curve results in jerk and unnatural steering maneuvers in a vehicle that attempts to complete such a path. It can be assumed that a smooth curve yields more natural driving and steering behavior. The first and second derivatives of a smooth curve in this sense are continuous. Per definition the required smooth curve, therefore, needs to be at least of class C^2. A parametric cubic spline provides a natural basis to model a drivable path, i.e., a road surface of a racetrack. To model the racetrack this work uses a piece-wise cubic B-spline, where the waypoints serve as control points (also called knots) for the spline. Equation (2.1) gives the used spline as a linear system, where $G_{i,3}$ is the basis function of the cubic B-spline.

$$Q(u) = \sum_{i=0}^{n} \boldsymbol{p}_i G_{i,3}(u) = \frac{1}{6} \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{p}_i \\ \boldsymbol{p}_{i+1} \\ \boldsymbol{p}_{i+2} \\ \boldsymbol{p}_{i+3} \end{bmatrix}, \; u \in [0, 1] \quad (2.1)$$

Equation (2.1) shows several properties of the resulting spline-curve. Given that the spline is cubic and therefore a section is defined by four control points, it becomes evident

that at least four waypoints are required for this approach. Further, because function $G_{i,3}$ is scalar, the dimension of the resulting curve is determined by the dimension of the knots. The resulting curve in the given planar workspace will be two dimensional ($\mathcal{W} \subset \mathbb{R}^2$ and therefore $P_i \in \mathbb{R}^2$). This 2D-cubic-B spline can be written as a function $\boldsymbol{Q}(t) : [0, n] \to \mathbb{R}^2$. $\boldsymbol{Q}$ can be viewed as a vector of two one-dimensional spline functions parametrized by t.

$$\boldsymbol{Q}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}, \; t \in [0, n] \tag{2.2}$$

The functions $x(t), y(t)$ in eq. (2.2) are piece-wise defined cubic-polynomials. For the interval $i \leq t \leq i+1$ they are defined as:

$$\boldsymbol{Q}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} = \begin{bmatrix} a_{x,i}(t-i)^3 + b_{x,i}(t-i)^2 + c_{x,i}(t-i)^2 + d_{y,i} \\ a_{y,i}(t-i)^3 + b_{y,i}(t-i)^2 + c_{y,i}(t-i)^2 + d_{y,i} \end{bmatrix}, \; t \in [i, i+1] \tag{2.3}$$

Equation (2.1) calculates the coefficients of those polynomials.

So far the continuous domain of control point indices is used to parametrize the spline-curve. This allows a convenient and compact definition of the spline, but has no actual geometrical meaning and therefore no equivalent that can be measured or observed by a mechatronic system. Therefore using $\boldsymbol{Q}$ in the context of a motion control is a non-trivial and unnecessary complex task. It is common practice in this context to use spline-curves that are parametrized by the arc-length $s \in [0, L]$ where L is the total arc-length of the curve. [WKA03] gives a simple two-stage process to achieve the arc-length parametrization of a spline-curve from any differentiable parametrization:

- Compute the arc-length s as a function of the parameter t: $A(t) = s$. Then s is a strictly increasing function of t, and there is a one-to-one correspondence between s and t.
- Find the inverse function $A^{-1}(s) = t$. This is a well defined and monotonically increasing function. Substitution of $t = A^{-1}(s)$ in $Q(t)$ yields a curve parametrized by arc-length s.

The mapping $A(t) : [0, n] \to [0, L]$ can easily be defined by geometric integration.

$$s = A(t) = \int_{t_0}^{t} \sqrt{(x'(t))^2 + (y'(t))^2} \, \mathrm{d}t \tag{2.4}$$

Where $x'(t)$ and $y'(t)$ are the first derivatives of $x(t), y(t)$ w.r.t. t. However, in general, the integral eq. (2.4) cannot be computed analytically. Therefore the arc-length parametrization cannot be expressed as a combination of elementary functions, but needs to be solved numerically. Since eq. (2.4) cannot be computed analytically, the same has to be right for the inverse function in eq. (2.5).

$$t = A^{-1}(s) \tag{2.5}$$

Possible numerical solutions for this problem will be discussed in section 3.4. As stated before, substituting the inverse mapping in eq. (2.5) into eq. (2.2) yields the desired arc-length parametrized spline-curve $P(s) : [0, L] \rightarrow \mathbb{R}^2$ in eq. (2.6).

$$\boldsymbol{P}(s) = \boldsymbol{Q}(A^{-1}(s)) = \begin{bmatrix} x(A^{-1}(s)) \\ y(A^{-1}(s)) \end{bmatrix}, \; s \in [0, L] \tag{2.6}$$

Therefore the arc-length parametrization can be viewed as a non-linear transformation of the parameter domain. It should be evident that this transformation does not affect the coefficients of the polynomial-sections of the spline. Because of this it is possible to express the sections of the spline as cubic polynomials of the parameter s in the interval $s_i \leq s \leq s_{i+1}$ using the same coefficients.

$$\boldsymbol{P}(s) = \begin{bmatrix} x(A^{-1}(s)) \\ y(A^{-1}(s)) \end{bmatrix} = \begin{bmatrix} a_{x,i}(s-s_i)^3 + b_{x,i}(s-s_i)^2 + c_{x,i}(s-s_i) + d_{x,i} \\ a_{y,i}(s-s_i)^3 + b_{y,i}(s-s_i)^2 + c_{y,i}(s-s_i) + d_{y,i} \end{bmatrix}, \; s \in [s_i, s_{i+1}] \tag{2.7}$$

where s_i is the arc-length at the control point $\boldsymbol{p}_i$.

Besides the continuity of a path, the drivability is highly dependent on the curvature. If the curvature exceeds the physical limitations of the robot, it is not possible anymore to follow this path. The curvature κ_b can be calculated given the first and second derivatives of x, y. Since x and y can be described as functions of either t or s, the derivatives can be calculated w.r.t. both parameters. Knowing x' and y' one can calculate the tangent angle of the spline-curve.

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \frac{\mathrm{d}x}{\mathrm{d}s} = x' = \cos\theta_b \qquad \frac{\mathrm{d}y}{\mathrm{d}t} = \frac{\mathrm{d}y}{\mathrm{d}s} = y' = \sin\theta_b \tag{2.8}$$

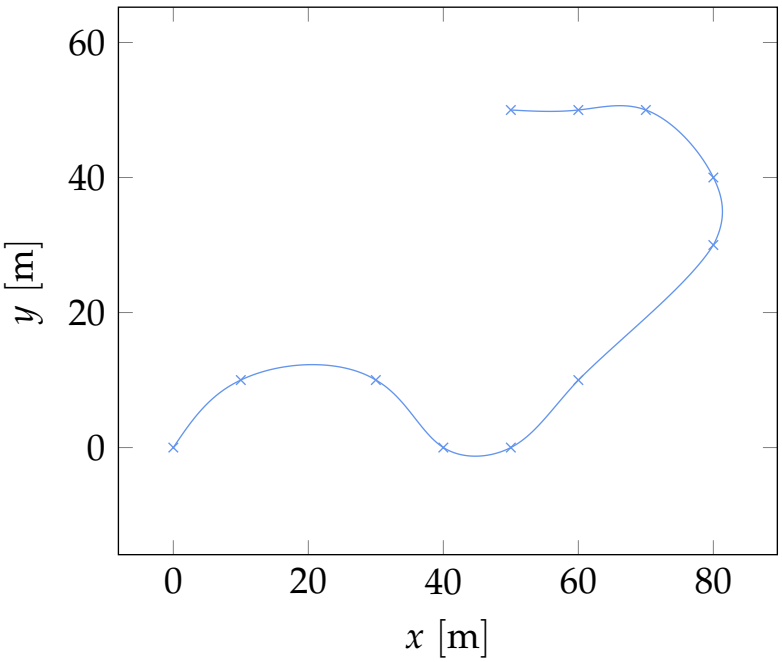
$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = \frac{\mathrm{d}^2x}{\mathrm{d}s^2} = x'' \qquad \frac{\mathrm{d}^2y}{\mathrm{d}t^2} = \frac{\mathrm{d}^2y}{\mathrm{d}s^2} = y'' \tag{2.9}$$

$$\kappa_b = \frac{x_b' y_b'' - x_b'' y_b'}{(x_b'^2 - y_b'^2)^{\frac{3}{2}}} \qquad \theta_b = \tan^{-1}\left(\frac{y'}{x'}\right) \tag{2.10}$$

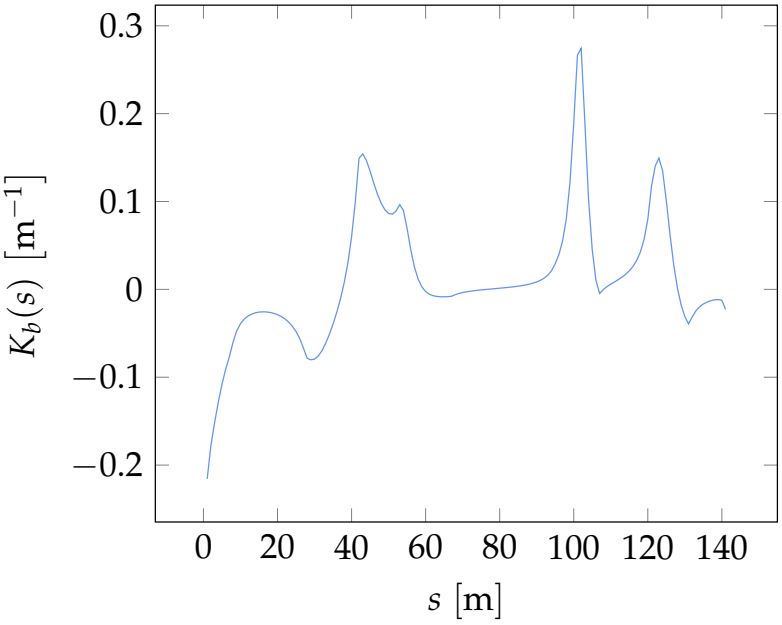
An example for the continuous connection of waypoints with an arc-length parametrized cubic B-spline is shown in fig. 2.1.

2.3. Curvilinear Coordinates

The spline curve that was defined in section 2.2 serves as the baseframe for a two-dimensional curvilinear coordinate-system. The base of this coordinate-system contains directional information and is directed to the goal-region. Coordinates with larger positive values are therefore closer to the goal-region. It is important to note that the arc-length parametrized spline curve that provides the baseframe is only defined for values that are within the range $[0, L]$, where L is the total arc-length of the curve and lies within the goal-region $\mathcal{C}_{goal}$. Further, the curvature of the road is modeled within this coordinate-system. The spline-curve can be viewed as the centerline of the



(a) Spline-curve for an example scenario



(b) Curvature of spline

Figure 2.1.: This figure illustrates how a cubic B-spline-curve can be used to model a continuous path through some waypoints. Figure 2.1a illustrates a chosen example scenario with 11 waypoints that might represent a section within a racetrack. The used spline-curve is an arc-length parametrized as it is described in section 2.2. Figure 2.1b shows the curvature of the spline as a function of the arc-length.

road or (if the waypoints represent an optimized line) as the optimal path through a racetrack. The orthogonal offset q from the centerline is plotted by the second axis of the curvilinear coordinate-system. This axis is not curved, but linear and orthogonal to the baseframe. This curvilinear coordinate-system has a one-to-one relation to a local Cartesian coordinate-system and coordinates can be transformed between the coordinate-systems with a nonlinear transformation.

$$x = \left(\frac{1}{\kappa_b} - q\right) \sin(s\kappa_b) \tag{2.11}$$

$$y = \frac{1}{\kappa_b} - \left(\frac{1}{\kappa_b} - q\right) \cos(s\kappa_b) \tag{2.12}$$

This relation conveniently describes paths as continuous functions of class C^2 with desired properties in the curvilinear coordinate-system. Those paths will naturally follow the shape of the road in Cartesian space. Figure 2.2 illustrates the axis of the curvilinear coordinate-system and the relation between a path in curvilinear coordinates and Cartesian space.

2.4. Coordinate Transformation

The position of the vehicle is obtained in planar Cartesian space. The path candidates will be generated in the curvilinear coordinate system (see section 2.5). Therefore it is necessary to obtain the position of the vehicle in curvilinear coordinates. One needs to find a transformation $\rho : \mathbb{R}^2 \to \mathbb{R}^2$ from Cartesian space to the curvilinear coordinate system. This is equal to finding the closest point to the vehicle on the spline curve by minimizing the squared Euclidean distance between the position of the vehicle and the baseframe.

$$\underset{s\in[0,L]}{\arg\min}\, d(\boldsymbol{P}(s), \boldsymbol{x})^2 = \underset{s\in[0,L]}{\arg\min}\, |\boldsymbol{x} - \boldsymbol{P}(s)|^2 = s_{\min} \tag{2.13}$$

A numerical minimization of eq. (2.13) allows to find the argument $s_{\min}$ that minimizes the lateral distance from the baseframe. Section 3.5 points out various suitable numerical minimization methods and discusses the method used in this work in depth. The minimal distance between a curve in Euclidean space and a point is a line. This line is orthogonal to the curve. Therefore the offset q from the base frame can be computed by $d(s_{\min}) = q_{\min}$. The position in curvilinear coordinates is therefore given by $\begin{bmatrix} s_{\min} & q_{\min} \end{bmatrix}^T$. The geometrical relations of this process are illustrated in fig. 2.3.

$$\boldsymbol{\rho}(\boldsymbol{x}) = \begin{bmatrix} \arg\min_{s\in[0,L]} d^2(\boldsymbol{P}(s), \boldsymbol{x}) \\ \min_{s\in[0,L]} d(\boldsymbol{P}(s), \boldsymbol{x}) \end{bmatrix} = \begin{bmatrix} s_{\min} \\ q_{\min} \end{bmatrix} \tag{2.14}$$

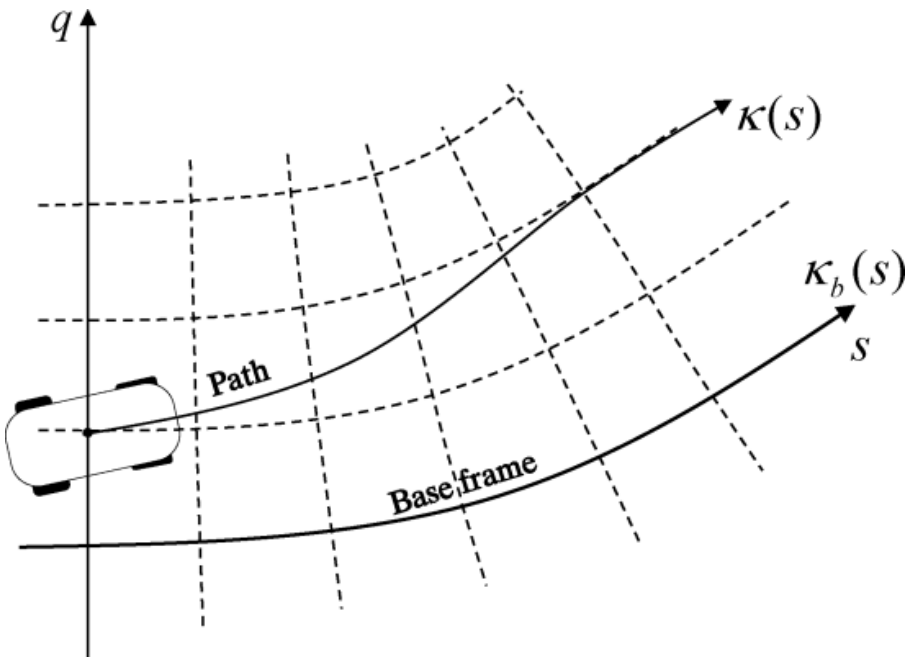


Figure 2.2.: This figure illustrates the layout of the curvilinear coordinate-system used for path generation. The s-axis is curved and models the centerline of the racetrack, where s describes the length along the track. $\kappa_b(s)$ denotes the curvature of the baseframe at arc-length s. An arc-length parametrized spline (see Section 2.2) serves as the baseframe for this axis. The orthogonal lateral offset q from the centerline is plotted along the second axis of this coordinate-system. An example path in curvilinear coordinates is plotted to illustrate the relation between curvilinear and Cartesian coordinates. $\kappa(s)$ denotes the curvature of the example path at arc-length s. (Figure from [CLS12])

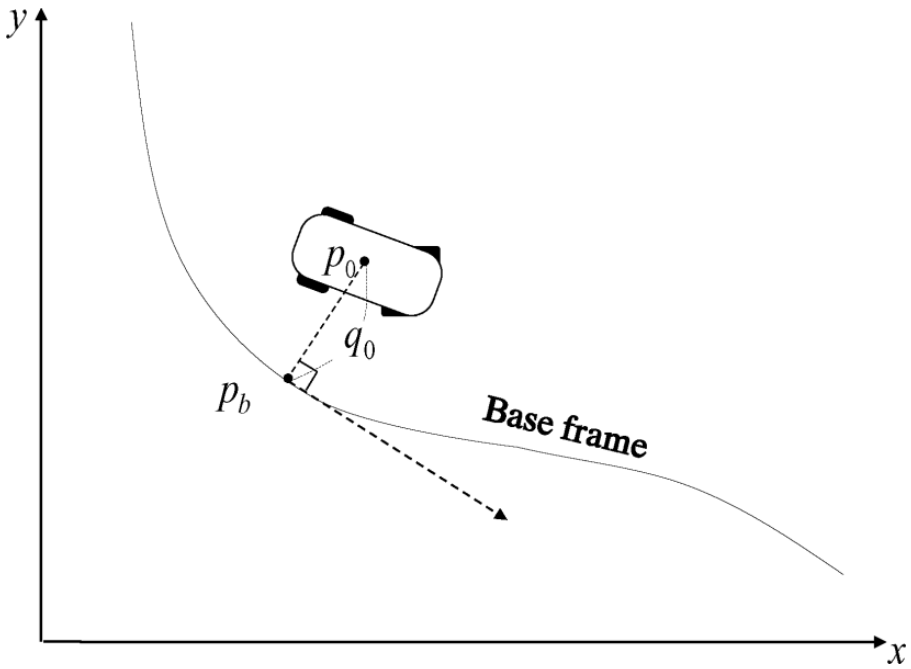


Figure 2.3.: This figure illustrates the geometric relations during localization of the vehicle in the curvilinear coordinate system. p_0 denotes the vehicle position in Cartesian space, p_b is the corresponding condition on the baseframe. The vector that connects this two points is orthogonal to the baseframe. The shortest distance between the points is given by q_0. The position in curvilinear coordinates is consequently given by the arc-length $s_{\min}$ at p_b and the shortest distance between the vehicle and the baseframe q_0. (Figure from [CLS12]).

2.5. Maneuver Definition

The cubic B-spline curve which was defined in section 2.2 provides a path from x_{init} to $\mathcal{C}_{\text{goal}}$. The question arises why one does not choose the spline as a local path for the vehicle. Unfortunately, it is not guaranteed that this path would be *feasible* nor *optimal* in the sense that was defined by section 1.2. The spline may be infeasible if it lies not entirely in $\mathcal{C}_{\text{free}}$. The optimality of the path depends on the designed objective function $J : \Sigma(\mathcal{C}) \to \mathbb{R}$ (see section 2.8). The spline would therefore only be guaranteed to be the optimal path if it was the only feasible path. To find a feasible and optimal path the planner consequently evaluates many candidate paths (also called maneuvers) in the observable workspace. Before such an optimal path, out of a set of maneuvers, can be chosen, one needs to define the maneuvers concerning their feasibility. Section 2.2 argued that smoothness is desired quality for a feasible path. Cubic-polynomials are of class C^2 and fulfill this requirement, which is why there are utilized in this work. A path candidate is defined by a smooth lateral offset function in the curvilinear coordinate system. Equation (2.15) defines such a smooth horizontal offset as cubic polynomial functions of the arc-length s. Equations (2.16) and (2.17) provide the first and second derivatives of eq. (2.15).

$$q(s) = \begin{cases} a(s-s_i)^3 + b(s-s_i)^2 + c(s-s_i) + d & \text{if } s_i \leq s < s_f \\ q_f, & \text{if } s_f \leq s \leq L \end{cases} \tag{2.15}$$

$$\frac{\mathrm{d}q}{\mathrm{d}s}(s) = \begin{cases} 3a(s-s_i)^2 + 2b(s-s_i) + c & \text{if } s_i \leq s < s_f \\ 0, & \text{if } s_f \leq s \leq L \end{cases} \tag{2.16}$$

$$\frac{\mathrm{d}^2q}{\mathrm{d}s^2}(s) = \begin{cases} 6a(s-s_i) + 2b & \text{if } s_i \leq s < s_f \\ 0, & \text{if } s_f \leq s \leq L \end{cases} \tag{2.17}$$

Figure 2.4 illustrates the geometrical relations of a maneuver. The initial arc-length s_i corresponds to $s_{\min}$ that was obtained in eq. (2.13) during the localization step. The initial lateral offset of the maneuver q_i corresponds to $q_{\min}$. The angular difference $\Delta\theta$ is defined by the difference between the tangent angle of the baseframe at s_i and the current vehicle heading. The shape of the maneuver can be influenced by the final arc-length of the maneuver s_f and the final lateral offset q_f. The length of the maneuver $\Delta s = s_f - s_i$ is the distance over which the lateral offset monotonically increases or decreases. Maneuvers are defined for values greater than s_f, but with a constant value q_f. With this information, one can define the following boundary conditions eqs. (2.18) to (2.21) which enforce the desired properties of the maneuver.

$$q(s_i) = q_i \tag{2.18}$$

$$q(s_f) = q_f \tag{2.19}$$

$$\frac{\mathrm{d}q}{\mathrm{d}s}(s_i) = \tan\Delta\theta \tag{2.20}$$

$$\frac{\mathrm{d}q}{\mathrm{d}s}(s_i) = 0 \tag{2.21}$$

Given those boundary conditions, the coefficients of the maneuver can be determined by solving the corresponding linear system. In section 3.6 this linear system is constructed and solved. The curvature of the maneuver in Cartesian coordinates can be calculated with eq. (2.22). For a derivation of eq. (2.22) the interested reader is referred to [Bar+02; Wer+10].

$$\kappa = \frac{S}{Q}\left(\kappa_b + \frac{(1-q\kappa_b)(\mathrm{d}^2q/\mathrm{d}s) + \kappa_b(\mathrm{d}q/\mathrm{d}s)^2}{Q^2}\right) \tag{2.22}$$

The values S and Q in eq. (2.22) are given in eq. (2.23).

$$S = \operatorname{sgn}(1-q\kappa_b), \qquad Q = \sqrt{\left(\frac{\mathrm{d}q}{\mathrm{d}s}\right)^2 + (1-q\kappa_b)^2} \tag{2.23}$$

The length of the maneuver Δs_f can be determined for the velocity of the vehicle. This enables a dynamic driving style, by taking into account the centripetal forces that may appear during driving. A higher velocity, therefore, requires the lateral change of the maneuver to be distributed over a longer distance. This requirement can be modeled by eq. (2.24).

$$\Delta s_f = k_v s + \Delta s_{\min} \tag{2.24}$$

Where k_v denotes a constant velocity gain factor and $\Delta s_{\min}$ denotes the minimal length of a maneuver. These parameters influence the length of a maneuver and can be chosen by design, or in future by a machine learning technique.

2.6. Kinematic Single Track Model

To be able to describe the motion of the vehicle in the workspace, a suitable kinematic model of vehicle motion is required. The kinematic single-track model [Pad+16; MS93; Cou92; Lav06] is a commonly used model of vehicle motion. It is the most basic model of practical use. The car is modeled as two wheels connected by a rigid link that moves in a plane. It is assumed that the wheels do not slip at their contact points with the ground. The rotation of the wheels is assumed to be free about their axis of rotation. To model steering, the front wheel has an additional degree of freedom, where it is allowed to rotate about an axis normal to the plane.

While this basic model is simple enough to be conveniently expressed as a three-dimensional non-linear system of ordinary differential equations (ODEs) of first order, it is yet sophisticated enough to model the limited maneuverability of a vehicle. Those limitations are referred to as non-holonomic constraints and can be expressed as differential constraints of the motion of the car.

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x},\boldsymbol{u}) = \begin{bmatrix} v\cos(\theta) \\ v\sin(\theta) \\ \frac{v}{l}\tan(\delta) \end{bmatrix}, \ \boldsymbol{x} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}, \ \boldsymbol{u} = \begin{bmatrix} v \\ \delta \end{bmatrix} \tag{2.25}$$

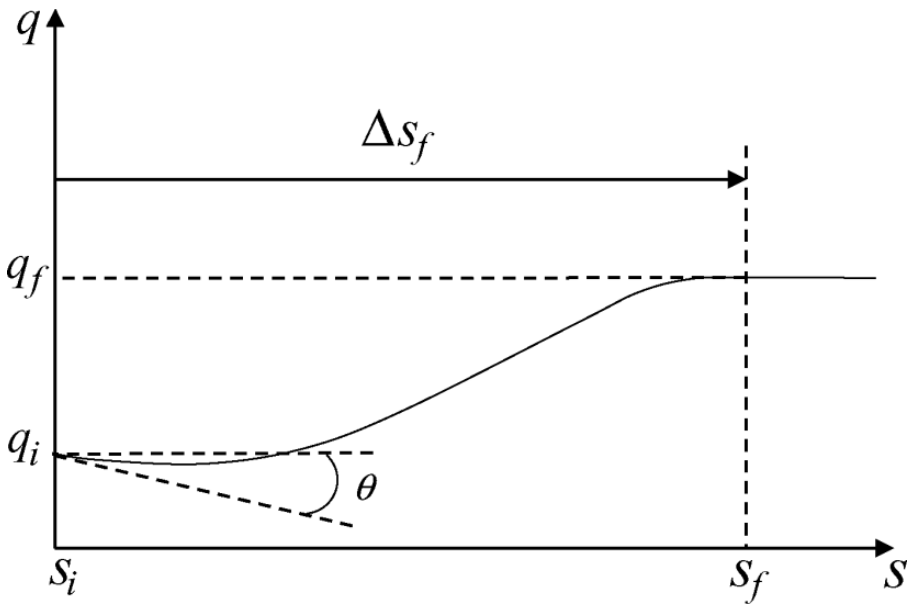


Figure 2.4.: This figure illustrates the geometrical relations of a maneuver in the curvilinear coordinate system (see section 2.3). The initial arc-length is denoted with s_i, the initial lateral offset with q_i. The final arc-length and lateral offset are denoted with s_f and q_f. The maneuver length Δs_f is the distance over which the lateral offset monotonically increases or decreases. The difference between the steering angle of the vehicle and the tangent angle of the baseframe is given by θ. (Figure from [CLS12])

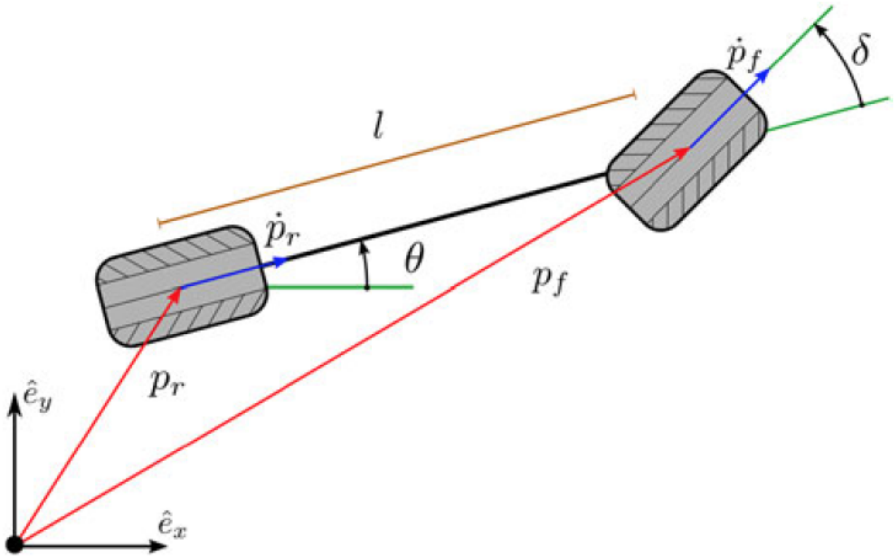


Figure 2.5.: Kinematics of the single-track model. p_r and p_l are the ground contact points of the rear and front wheel respectively. θ denotes the vehicle heading. The time derivatives of p_r, and p_f are restricted by the non-holonomic constraints to the direction indicated by the blue arrows. The steering angle of the front wheel is denoted by δ. (Figure taken from [Pad+16])

The single track model serves as a basis to express the vehicle movement within the planning algorithm used in this work. However, the model needs to be adapted to represent motion in available terms. The first modification to this model is to express the change in heading, not regarding the steering angle δ, but with the curvature κ of the path that the vehicle travels. By substituting $l \tan(\delta) = \kappa$ in eq. (2.25), one can derive the left-hand side of eq. (2.26). It is not required to express the vehicle motion regarding velocity in this context, since the planner does not consider it at this stage. Therefore we substitute $v = SQ\frac{\mathrm{d}s}{\mathrm{d}s}$. This yields a system of ODEs that is not expressed with respect to time t, but with respect to the traveled arc-length s. We arrive at the system at the right-hand side of eq. (2.26).

$$\dot{\boldsymbol{x}} = \begin{bmatrix} v\cos(\theta) \\ v\sin(\theta) \\ \frac{v}{l}\tan(\delta) \end{bmatrix} \xrightarrow{l\tan\delta=\kappa} \dot{\boldsymbol{x}} = \begin{bmatrix} v\cos(\theta) \\ v\sin(\theta) \\ v\kappa \end{bmatrix} \xrightarrow{v=SQ\frac{\mathrm{d}s}{\mathrm{d}t}} \frac{\mathrm{d}}{\mathrm{d}s}\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x}, s) = \begin{bmatrix} Q\cos\theta \\ Q\sin\theta \\ Q\kappa \end{bmatrix} \tag{2.26}$$

This representation allows solving the system of ODEs by integration over the traveled distance along the baseframe s. Finally, this allows us to define $\tau(s) : [0, L] \to \mathcal{C}$.

$$\tau(s) = \int_{s_i}^{s} \boldsymbol{f}(x, s)\, \mathrm{d}s \tag{2.27}$$

Section 3.6 discusses numerical solutions for this system.

2.7. Collision Detection

A feasible path must not collide with obstacles in the workspace. Therefore, for a feasible path, the relation in eq. (2.28) must hold.

$$\tau(\alpha) \in \mathcal{C}_{\text{free}}, \qquad \forall \alpha \in [0,1] \tag{2.28}$$

We can define a logical predicate that checks whether this relation holds for a continuous section of the configuration space.

$$\Phi : \Sigma(\mathcal{C}) \to \{0,1\} \tag{2.29}$$

Collision detection might appear as an effortless operation, but in fact, it is a non-trivial task both in time and memory complexity. According to [Lav06, Sections 5.3] in most motion planning applications, the majority of time is spent on collision checking. As a result, an efficient collision detection algorithm is crucial for online operation of a motion planner.

The essential operation of Φ checks wheter the robot $\mathcal{A}(\boldsymbol{x}) : \mathcal{C} \to \mathcal{W}$ is in collision with an obstacle in the workspace. If the complexity of this operation can be reduced, it has a significant impact on the runtime of a motion-planning program. A commonly used approach to this problem is to approximate the shape of $\mathcal{A}$ and $\mathcal{W}_{\text{obs}}$ using geometrical primitives as bounding boxes (see fig. 2.6). Usually, bounding boxes that provide a more tight approximation of the robot are more expensive to test for overlapping regions. Within this work, an oriented bounding box (OBB) is used to approximate the shape of the robot.

The complexity of Φ can be further reduced by only performing the collision check on a subset of $\mathcal{W}_{\text{obs}}$. A naïve approach might be to check only the section of $\mathcal{W}_{\text{obs}}$ that was observed during the current sensing interval. While this approach might result in considerable reduction in complexity, its solution is unreliable. Obstacles that were sensed during former intervals might have gone out of the field of view, but still lie in the reachable configuration space of the robot. In general, unprocessed environment sensing provides an imperfect and noisy representation of the world. An obstacle that was reliably sensed before might have been missed in the current interval. Therefore all obstacles that have been sensed during operation have to be taken into account. This work proposes a novel solution to the problem of reducing the complexity of the collision checking task, while still retaining a collision check of the entire known $\mathcal{W}_{\text{obs}}$. An R-Tree [Gut84] is utilized as a spatial indexing structure for observed obstacles. A spatial index efficiently stores the spatial information of obstacles. Within the context of collision checking the spatial index can be used to retrieve obstacles with an axis aligned bounding box (AABB) that intersects with the AABB of the vehicle. The R-Tree can be queried with linear complexity $\mathcal{O}(n)$. If the query returns a set of obstacles that fulfill this criterion, the collision check is performed on the OBB of the obstacles is performed. This two-stage process reduces the complexity of the collision checking operation, while still providing an accurate and geometrically reasonable result. The author is not aware

of other work that uses a spatial index in the context of collision checking and believes that this is a novel approach. Algorithm 1 samples the given path $\tau(\alpha)$ with granularity Δs. The granularity is chosen empirically and is a design parameter of the algorithm. For all $\boldsymbol{x} = \tau(\{s_0, s_1, \ldots, s_n\})$ the R-Tree is queried for obstacles that intersect with the AABB of the robot $\mathcal{A}(\boldsymbol{x})$. If this query returns a non-empty set, a collision check is performed on the convex-hulls of the robot and obstacles. The algorithm exits with a positive result on the first collision which the second level of the checking detects. If no collision is detected for all $s \in \{s_0, s_1, \ldots, s_n\}$, then the algorithm returns with a negative result. One may criticise that discrete sampling is not a logically complete solution. [Lav06, Section 5.3.4] suggests an alternative solution to discrete sampling that is logically complete, but requires additional information. Such a solution might be implemented in future work.

input : $\tau : [0, L] \to \mathcal{C}, \mathcal{A} : \mathcal{C} \to \mathcal{W}$, R-Tree
output: 0 if $\tau([0, L]) \subset \mathcal{C}_{\text{free}}$, 1 otherwise

```
1  for i ← 0 to L by Δs do
2      x ← τ(i);
3      O ← R-Tree.query(AABB(A(x)) ∩ W_obs );
4      for o ∈ O do
5          if A(x) ∩ o ≠ ∅ then
6              return 1;
7          end
8      end
9  end
10 return 0;
```

Algorithm 1: Collision checking function $\Phi(\tau) : \Sigma(\mathcal{C}) \to \{0, 1\}$. This algorithm performs a collision check for a continuous path $\tau : [0, L] \to \mathcal{C}$ at discrete equidistant sampling points $s_0, s_1, \ldots, s_n$. The sampling interval Δs is empirically chosen. $\mathcal{A}(\boldsymbol{x})$ is the subset of workspace that is occupied by the robot at configuration $\boldsymbol{x}$.

2.8. Maneuver Selection

To find a feasible and near-optimal path the planning algorithm evaluates a set $\mathcal{M}$ of N possible candidate paths with distinctive lateral offsets q_f. The lateral offsets are in a manner to cover a broad section of the reachable configuration space. This is archived by distributing them equally over the width of the road. Information about the road width needs to be provided to the path planner, either by sensor information or as a design parameter. Since the optimal path is found by taking the path from the set that minimizes the cost function, it is required only to contain feasible paths. The set of

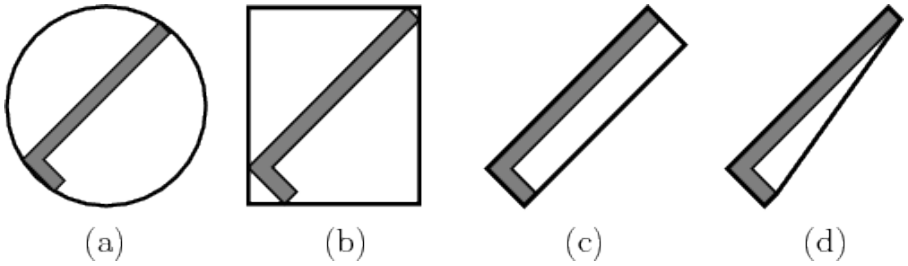


Figure 2.6.: Different kinds of bounding boxes for an L-shaped robot. (a) sphere, (b) AABB, (c) OBB and convex hull. Each provides a more tight approximation than the previous one but is more expensive to test for overlapping pairs. Within this work AABBs and OBBs are used. (Figure taken from [Lav06, p. 211])

possible maneuvers is defined as

$$\mathcal{M} \subseteq \Sigma(\mathcal{C}_{\text{free}}) \tag{2.30}$$

To enforce this relation the planner needs to only add paths to the set of candidates that lie within $\mathcal{C}_{\text{free}}$. Paths that violate the non-holonomic constraints of vehicle motion or collide with an obstacle are infeasible and need to be discarded. Weather a path lies within $\mathcal{C}_{\text{free}}$ can be derived from its geometric characteristics. If the curvature of a path exceeds the maximum curvature $\kappa_{\max}$ it contains configurations that can not be reached by the vehicle. If the lateral offset of a path q is larger than the radius of curvature $1/\kappa_b$, the curvature and the direction of the path will be opposite in sign to the baseframe. The resulting movement cannot be performed by a non-holonomic vehicle, and the corresponding path is to be discarded. A path that contain a lateral offset q that is equal to the radius of curvature $1/\kappa_b$ passes through the center of curvature of the baseframe. For a path to be feasible, the lateral offset needs to be less than the radius of curvature of the baseframe in the definition domain. The feasibility function $\zeta : \Sigma(\mathcal{C}) \to \{0, 1\}$ will return a positive result if relation 2.31 holds over the entire definition domain of a path.

$$\zeta(\tau) = \begin{cases} 1, & \text{if } |\kappa(s)| \leq \kappa_{\max} \wedge q(s) < \frac{1}{\kappa_b(s)},\ \forall s \in [s_i, s_f] \\ 0, & \text{otherwise} \end{cases} \tag{2.31}$$

To ensure that a path τ lies entirely in $\mathcal{C}_{\text{free}}$ a collision check needs to be performed. The collision checking function $\Phi : \Sigma(\mathcal{C}) \to \{0, 1\}$ is discussed in detail in section 2.7. This allows to define a set of path candidates $\mathcal{P}$.

$$\mathcal{P} = \{\tau \in \Sigma(\mathcal{C}) |\ \neg\Phi(\tau) \wedge \zeta(\tau)\} \tag{2.32}$$

The set $\mathcal{P}$ defines the set of feasible paths from which the planner will take the near-optimal one. In section 1.2 it was argued that the optimal solution of the given problem is the argument that minimizes the cost function $J : \Sigma(\mathcal{C}) \to \mathbb{R}$. The cost function is used

to model desired characteristics of the optimal solution. Such desired characteristics can be used to choose paths that are less likely to collide with obstacles in the workspace. In the context of the scenario defined by the FSG rules, a safe path is less likely to receive a penalty for the collision with a road delimiter. High velocities appear during a track drive. The centripetal acceleration of a vehicle is directly related to the velocity and the curvature of the driven paths. Smoother paths, therefore, allow higher velocities, which are a crucial component in a competitive race. The smoothness is another characteristic that can be modeled by the objective function. Lastly, the consistency of a path has to be considered. The consistency encodes the change in the current planning step in comparison to the path chosen in the previous planning step. Inconsistent path choices could lead to abrupt changes in heading, which require excessive energy and control effort. The result may be a degradation of the driving stability that might ultimately result in a collision. The objective function used in this work consequently takes safety, smoothness, and consistency of a path into account. For every desired characteristic a corresponding cost function is designed. Equation (2.33) shows that the objective function is a weighted linear combination of the safety cost C_s, smoothness cost C_κ and consistency cost C_c.

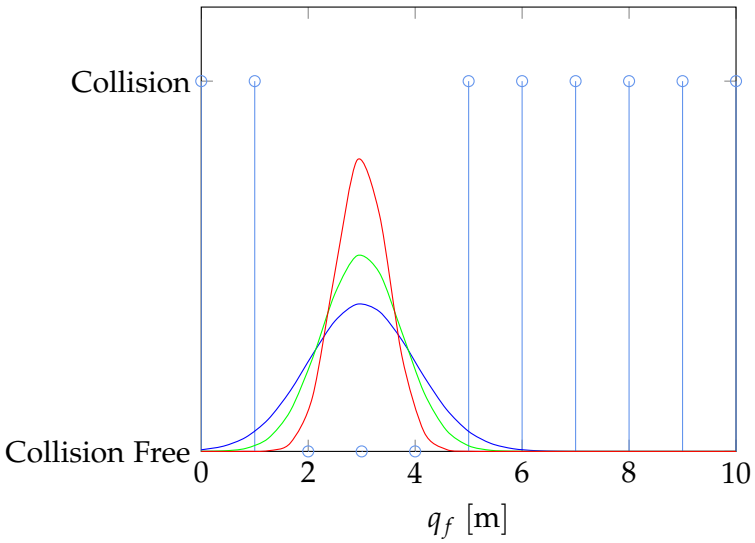
$$J(\tau) = w_s C_s(\tau) + w_k C_k(\tau) + w_c C_c(\tau) \tag{2.33}$$

In prevention of collisions, the planner performs a collision check. The collision checking function $\Phi : \Sigma(\mathcal{C}) \to \{0, 1\}$ returns a binary result and consequently does not allow an assumption on the likelihood of collision. Commonly the sensing of the vehicle state or the environment is imperfect and contains some random noise. As a consequence adjacent paths to collision paths are more likely to lead to collisions, even if the collision check returned a negative result. The distance from a collision path provides valuable information about the safety of the path. The safety cost function quantitatively evaluates the risk for each path by *blurring* the binary data for a collision. The risk of collision of a path is defined by a discrete Gaussian convolution.

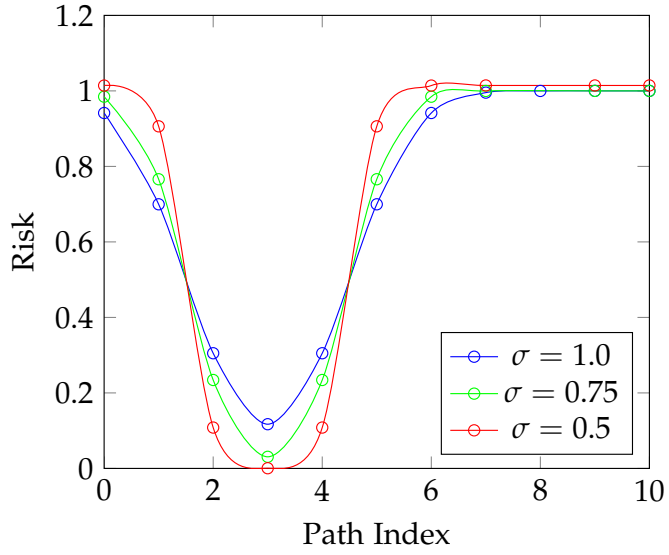
$$C_s(\tau) = \sum_{\tau_i \in \mathcal{M}} \Phi(\tau_i) g(q_{\tau_i,f}; q_f, \sigma_s), \qquad g(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{\sigma^2}\right) \tag{2.34}$$

where $\Phi : \Sigma(\mathcal{C}) \to \{0, 1\}$ is the collision checking function and $q_{\tau_i,f}$ denotes the final offset of the path τ_i. The safe distance to a collision path can be influenced by the standard deviation σ_s. A greater standard deviation σ_s will result in a broader Gaussian; consequently collision paths further away will have a higher impact on the risk. With a lower standard deviation the planner is more likely to choose paths that go through narrow gates. This principle is illustrated in fig. 2.7.

If one assumes a constant speed of a vehicle along a path, the instantaneous centripetal acceleration of the vehicle is proportional to the curvature of the path. Consequently, the integration of the squared curvature is equivalent to the integration of the squared acceleration. The integration of the curvature of the path κ in eq. (2.35) is performed



(a) Discrete convolution of three Gaussians with distinct σ_s.



(b) Risk blurrage of adjacent paths.

Figure 2.7.: This figure illustrates the definition of the safety cost function C_s as a Gaussian convolution and shows the effect of the standard deviation σ_s. Figure 2.7a shows a discrete convolution of three Gaussian with distinctive standard deviations. This example shows how the path cost for the path with the final lateral offset $q_f = 3$ would be performed. Figure 2.7b illustrates how the risk is blurred on adjacent paths. The path at index 3 is the furthest away from collisions. As expected the Gaussian convolution returns the lowest risk for this path. The influence of the standard deviation on the collision risk can be observed in this figure. A greater σ_s results in a broader Gaussian and therefore collisions further away will have a higher influence on the risk.

concerning the arc-length of the path s_p. This can be calculated by using the projection Q on the differential $\mathrm{d}s_p$ and integration over the arc-length of the baseframe s.

$$C_\kappa(\tau) = \int \kappa^2(s)\ \mathrm{d}s_p = \int \kappa^2(s) Q\ \mathrm{d}s \tag{2.35}$$

A higher weight on the curvature cost will lead to a planner that prefers paths that contain minimal curvature and allow high-speed dynamic driving.

The consistency cost considers the change of a path in comparison to a path from a former planning interval. The change is represented as the interval of the distance between the two paths for the overlapping section of them. This can be conveniently defined in the curvilinear coordinate system, since the distance for an equal value of the arc-length s is merely the absolute difference in lateral offset.

$$C_c(\tau) = \frac{1}{s_{f,\mathrm{last}} - s_i} \int_{s_i}^{s_{f,\mathrm{last}}} |q_{\mathrm{last}}(s) - q(s)|\ \mathrm{d}s \tag{2.36}$$

Equation (2.36) denotes the consistency cost as the described integral, where $s_{f,\mathrm{last}}$ denotes the final arc-length of the path that was returned from the former planning step τ^- and q_{last} consequently denotes the offset from the path for τ^-. The left symbols refer to the path τ and the known geometrical relations of a maneuver (see section 2.5).

2.9. Algorithm Description

Now, that fundamental principles and notation have been introduced; the algorithm can be formally described in pseudo code (see algorithm 2). The motion planner is updated when new sensor information is available, until $\boldsymbol{x}_{\mathrm{init}} \in \mathcal{C}_{\mathrm{init}}$, and the planning problem, therefore, is solved.

input : $\mathcal{W}_{\text{obs}}, \boldsymbol{x}_{\text{init}}, \mathcal{C}_{\text{goal}}, D, J$
output: $\tau = \arg\min_{\tau \in \Sigma(\mathcal{C})} J(\tau)$ such that
$\tau(0) = \boldsymbol{x}_{\text{init}} \wedge \tau(L) \in \mathcal{C}_{\text{goal}}$
$\tau(\alpha) \in \mathcal{C}_{\text{free}} \wedge D(\tau(\alpha), \tau'(\alpha), \tau''(\alpha), \dots) \qquad \forall \alpha \in [0, L]$

$[s_i, q_i] \leftarrow \rho(\boldsymbol{x}_{\textbf{init}})$;
$\mathcal{M} \leftarrow \emptyset$, $\mathcal{P} \leftarrow \emptyset$, $\Delta q = q_{\max}/2N$;
for $q_{i,f} \leftarrow -q_{\max}$ **to** $q_{\max}$ **by** Δq **do**
  $\tau \leftarrow$ make_path(s_i, q_i, θ, q_f);
  $\mathcal{M} \leftarrow \mathcal{M} + \{\tau\}$;
  if $\neg\Phi(\tau) \wedge \zeta(\tau)$ **then**
    $\mathcal{P} \leftarrow \mathcal{P} + \{\tau\}$;
  end
end
if $\mathcal{P} \neq \emptyset$ **then**
  $\tau_{\text{near-optimal}} \leftarrow \arg\min_{\tau \in \mathcal{P}} J(\tau)$;
  return $\tau_{\text{near-optimal}}$;
else
  $\tau_{\text{suboptimal}} \rightarrow \arg\min_{\tau \in \mathcal{M}} d(\tau)$;
  return $\tau_{\text{suboptimal}}$;
end

Algorithm 2: Pseudo code for the motion planning algorithm.

3. Implementation

This chapter discusses details on the implementation of the algorithm described in chapter 2. The described local-planning algorithm was implemented using the object-oriented programming language C++ with the ISO standard of 2014 [ISO14]. While this chapter provides an in-depth discussion of the design rationale of the software in section 3.1, it does not serve as documentation for the provided implementation and source. Instructions how to access the documentation and source codes is provided in appendix C. The implementation uses several third-party software libraries which are discussed in section 3.2. An in-depth discussion of the entire source code is not intended in this chapter. However, selected listings of interest that illustrate the functionality of the planner, the use of the software libraries or curious language features will be discussed.

3.1. Design Rationale

The provided software is designed with simplicity, usability, and extendability in mind. The software design aims to be simple enough to be easily understood by users. Therefore the design has to make a trade-off between flexibility and simplicity. Since the usecase of the software is quite specific (see section 1.3), one can sacrifice flexibility regarding a generic design that might be used in other domains. Further, the software provides interfaces for future extensions, in the from of object-oriented abstraction mechanisms. The module is provided as a standalone software module, that only depends on software libraries that it directly uses to implement its functionality (see section 3.2). An application that uses the provided module needs to include the sources, create a planner object, provide a map to it and update its state every time new sensor information is available. Since the goal domain for the implemented planner is autonomous racing, low latencies are crucial. The software should run with 20 Hz with near-optimal performance on the default set of design parameters. The user of the software can influence the performance of the planner, by choosing a different set of design parameters. This can be quickly done with command line flags (see section 3.10). Regarding reliability the software should only fail, if not a single feasible maneuver can be generated. If this happens, the software should fail controlled and provide as much insight into its internal state as necessary. The cause for this behavior is in general related to a set of invalid design parameters or an environment that is too crowded with obstacles.

On creation of the planning object, the planner generates a baseframe for the curvilinear coordinate system from the provided map. The baseframe is implemented as a one-to-one aggregation to the planner. Further, an instance of an R-tree is created to

store obstacles.

The planner defines a function `update` which is intended to be iteratively called. During an update-iteration, the planner creates a list of N instances of maneuvers with the given design parameters and adds newly observed obstacles to the R-tree. Maneuvers can access the R-tree over an observer pointer, so all maneuvers share the same instance of the spatial index. Those maneuvers are checked for collisions, and their objective functions are evaluated. During the collision checking step, the discrete representation of the maneuvers is generated. The results are stored in an associative ordered container. The keys of the container for the feasible and collision-free paths is their cost, that was returned from the objective function. For paths which contain collisions, their keys are the length along the baseframe until the collision occurs. The containers store their contents in ascending order by their keys. If the container for the feasible maneuvers is empty, the maneuver at the end of the container with the collision paths is returned (since it is the one with the longest distance until a collision occurs). Otherwise, the function returns the element at the beginning of the candidates-container (which is the element that minimizes the objective function). The discretization of those instances of the class `Maneuver` can be accessed by a member function called `Maneuver::path`. While only the discretization of the maneuver might be needed to run a controller which drives the vehicle to the desired pose, the full instance is provided, since more sophisticated controllers may require additional information.

Figure 3.1 illustrates the design rationale in a simplified Unified Modeling Language (UML).

3.2. Software Libraries

The language functionalities of C++ are powerful enough to achive virtually every programming task. However, often it is unnecessary to implement common functionalities. In most cases, it is more efficient just to use a software library to solve a task. Therefore, most serious software projects include third-party code. The C++ community supports a vast and ever-growing ecosystem of open source libraries, that one can freely use. Some are very well known and are likely to be found in almost every C++-Project. The remaining part of this section will introduce the third-party libraries, which presented implementation uses.

The standard template library (STL) [1] is defined by the ISO C++ standard and shipped with every C++-implementation. It contains four components called algorithms, containers, functors, and iterators. Algorithms provide solutions to common problems like sorting and searching. Containers provide implementations for common data structures like lists, maps, vectors, double-ended-queues, tuples and pairs. Functors provide an interface for functional programming. The implementation contains lambdas

[1] http://www.cplusplus.com/reference/stl

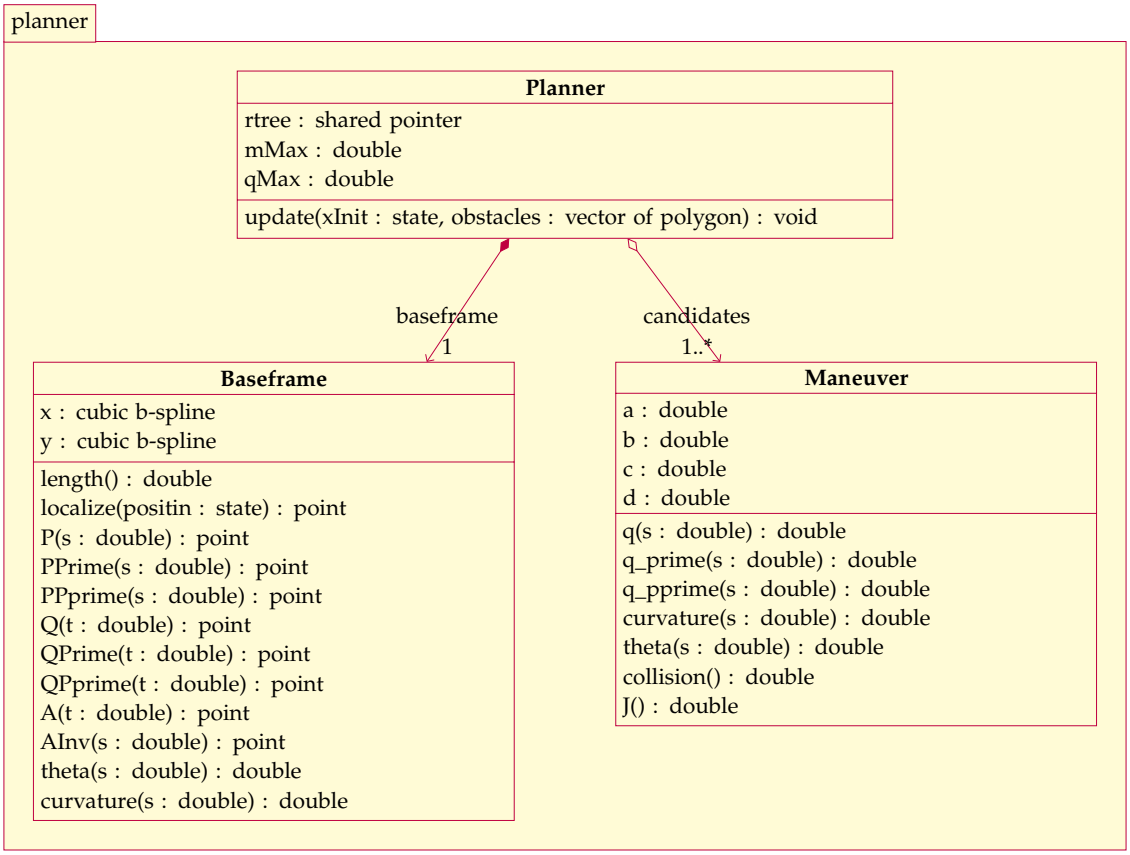


Figure 3.1.: Design rationale as simplified UML. The names of the functions and class members refer to mathematical symbols used in this work. Functions that are named *prime* or *pprime* return the first or second derivative of the base function. The core of the software consists of three classes: `Planner`, `Baseframe`, and `Maneuver`. To use the provided software the user creates an instance of a `Planner`. On creation, the `Planner` creates an instance of `Baseframe` which can be accessed by every instance of `Maneuver`. `Planner` defines a function update which is intended to be iteratively called to update the path plan. For more detail, on the available functions the interested reader is referred to the documentation (see appendix C.4).

and functor objects. Iterators provide, as the name suggests, functionality to iterate over a range of elements. Those components can be used with every build-type, and user-defined types that define some elemental operations such as compare and copy. Through the use of template-programming, the STL achives compile-time polymorphism which in general is more performant than run-time polymorphism. Since the STL is contained in the Standard, it is very well known within the C++-community. Maintainers do not have to learn alternative designs but can focus on the implementation. However, STL is not only a comfort-zone for C++-programmers. Compilers use the STL for optimization [God17]. This means that using the STL results in a program that performs better and is smaller than an equivalent self-made solution would be. Bjarne Stroustrup, one of the authors of the C++-Standard and the inventor of the language, strongly recommends in the book *the C++ Programming Language 4Th Edition* the use of the STL wherever feasible "[...] for portability and long-term maintainability". Consequently, the use of the STL results in good maintainability, performance, and design. Omitting the STL in a modern C++-Project is the textbook definition of "reinventing the wheel".

Boost [2] is a collection of widely useful libraries that complement the STL. Boost libraries are platform independent and support most operating systems. Since Boost libraries are based on the C++-Standard, they are implemented using state-of-the-art C++. "The Boost C++ Libraries enable you to boost your productivity as a C++ developer" [Sch17] and are known to be "[...] one of the most highly regarded and expertly designed C++ library projects in the world" [SA04]. Language extensions that have been introduced in Boost (i.e., smart-pointers, initializer lists or variants) have been picked up by the developers of the C++-Standard. Boost libraries are mostly lightweight header-only libraries. This keeps integration effort and increment of compile times to a minimum. Consequently it is widely used and can be found in many projects. In this work the following Boost libraries are used: As the name indicates, boost::math includes contributions to the domain of mathematics. Some of which are cubic B-splines, statistical distributions, mathematical constants, special functions and numerical integration, root finding and minimization algorithms. The numerical integration of boost::math is unsuited to solve initial value problems (IVPs) of ODEs (i.e., kinematic models for vehicle motion, see section 3.6). This gap is filled by the library boost::numerical::odeint, which contains adaptive and non-adaptive numerical integration steppers like the explicit Euler or Runge-Kutta method. boost::geometry provides concepts, primitives and algorithms to solve geometrical problems. The library is still under active development but contains promising features that this work utilizes for collision detection. The library uses template programming and a trait system to achive a high level of flexibility. Useful types that this library provides are points, polygons, boxes, lines and spatial indexes (i.e., R-Trees). The design of the library enables the use of those types with all standard types and a variety of coordinate systems.

[2] http://www.boost.org/

Eigen3 [3] is a high level header-only template library for numerical linear algebra. [GJa10] states that Eigen is versatile, fast, reliable and elegant. Eigen provides implementation of basic linear algebra subroutines (BLASs) level 1, 2 and 3 for dense and sparse matrices. Numerical linear algebra is known for being computational intensive [HS09]. Naïve implementations of BLAS can take up a significant part of the runtime of a program. Several libraries are available that solve this problem by offering an efficient implementation of BLAS, some of which are boost::uBLAS, cuBLAS, Armadillo, and Eigen. While boost::uBLAS might be the obvious choice, since boost is already available in the implementation, it is outdated. The authors ot the library deprecated the project and recommended to use one of the other available options [Wal+11]. cuBLAS offers high performance by utilizing massively parallelized computation on graphics processing units (GPUs), but is clunky to use and needs an extra compiler. Armadillo and Eigen are very similar, in both performance and usability. The decision for Eigen was made based on experience and personal taste.

gflags [4] implements command line flag processing with build in support for standard types. Command line flags enable the user to specify options or parameters of the program as successive lines of text to the program. Gflags offers an easy to use and lightweight implementation. This work utilizes gflags to implement design parameters and verbosity levels of logging for the planner.

glog [5] implements an expansive logging module. Logging is a way to communicate internal states of a program to the user. It can serve as an analytical tool or a simple human machine interface (HMI). Glog implements contextual logging, logging with different levels of verbosity, debug logging, conditional logging, and macros for run-time assertions.

3.3. Tools

Tools for modern software development do not add directly to the functionality or quality of software. However, by automating various everyday tasks like building or sanitizing source code, they increased productivity during developmen. Further they add to the good maintainability of the product. This section is dedicated to the tools that have been used to implement the provided path planner.

git [6] is the de facto standard tool for distributed source-code versioning and management. “Version control is a system that records changes to a file or set of files over time so that [one] can recall specific versions later”[CS14]. Distributed version control

[3] http://eigen.tuxfamily.org
[4] https://github.com/gflags/gflags
[5] https://github.com/google/glog
[6] https://git-scm.com

systems can be used with a cloud provider service (like GitHub[7] or Gitlab[8]) for data backup and code distribution. Git is known to be lightweight and fast which makes it a natural choice.

waf [9] is a Python-based meta-build system. Building a software project is a non-trivial task, that contains several steps. The program sources have to be compiled into binary code, the binary code then needs to be linked and packaged into a usable form. During software development, those steps are performed regularly to build the program-executable (i.e., for testing). While these steps can be performed manually, it is more practical to automate them with a build-tool. Most modern build-utilities can automate a variety of development tasks like building, testing, deployment and generating documentation. A variety of tools is available for C++-build automation: GNUMake, CMake, Gradle, Bazel, Bake are just a few. Waf is picked from the available choices because of several reasons: As a meta-build system, it is capable of building many different projects and languages. While the presented package is purely implemented in C++14, it is a sub-module of a bigger software project that contains several languages. Waf enables the REV-team to use one tool to build the entire polyglot project. Further waf is implemented as a single Python script that can be stored with the program sources. The setup progress for waf, therefore, is minimal, since every system that has a Python executable installed can run waf. The same reasoning holds for the issue of portability. Last factor in the decision is the learning effort: waf uses Python files to specify the rules how to build a project. Since Python is one of the official languages that are used within the REV-Project, the users can be expected to know the syntax already. Appendix C gives instructions how to use waf to build the sources.

cpplint [10] is a static code checker for C++. It checks C++ sources for violations of the Google C++ guidelines. The tool itself is implemented in Python, which results in high portability and usability. While code-style and sanitation do not directly contribute to functionality and quality of the implementation, they enforce common standards and add to the readability of the code. Source codes that follow known rules of style are easier to understand by maintainers. The provided sources follow the Google C++-style-guide.

clang++ [11] is a compiler front-end for C-style languages. Compared to other C++-compilers it is faster and uses less memory[12]. Clang is compliant with the C++-14 standard.

[7] https://github.com
[8] https://gitlab.com
[9] https://waf.io/
[10] https://github.com/cpplint/cpplint
[11] https://clang.llvm.org/
[12] https://clang.llvm.org/comparison.html

clang-format [13] is an automation tool for code formatting. As with code-style, formatting adds to the readability of the sources. Automation of formatting enforces the standards. Maintainers of the source codes do not need to test their implementations for violations of formatting-style, but can use the tool.

doxygen [14] is the de-facto standard tool for generating documentation from annotated C++ sources. Documentation is a crucial contribution to the usability of a program. Maintainers can consult the documentation for details on implementation and functionality. The profit from a tool that generates documentation from annotated sources is that the documentation is embedded in the sources. The documentation, therefore, is available as comments in the source code, as a handbook or online-resource. Appendix C gives instructions on how to access the documentation of the provided implementation.

3.4. Baseframe

The baseframe for the curvilinear coordinate system is an arc-length 2D cubic B-spline (see section 2.2). The implementation, therefore, has to solve two main problems: computing 2D cubic B-spline that connects a given set of points and find the transformation function of eq. (2.5) to represent the spline as a function of its arc-length. The remaining tasks for the baseframe (like computing it is total length L or the tangent angle θ) are trivial computations and do not need to be explained in detail.

Boost provides a numerically stable class for accurate and fast cubic b-spline interpolation of a function that is known at equally spaced points. Implementing the 2D cubic b-spline of eq. (2.2) is simply a matter of combining two instances of this class, one for the x-dimension and one for the y-dimension of the way-points. The baseframe is implemented as a class `Baseframe` that holds two private members of type `boost::math::cubic_b_spline`. Listing 3.1 shows the construction of the base frame from two `std::vector` that store the x and y coordinates of the way-points. The class uses constructor chaining to implement a set of constructors that initialize the underlying members from a variety of STL and `boost::geometry` types. The constructor in listing 3.1 is used as an endpoint for the chain of constructors.

```
#include <boost/math/interpolators/cubic_b_spline.hpp>
#include <vector>

using boost::math::cubic_b_spline;

void Baseframe::Baseframe(const std::vector<double> vx,
                          const std::vector<double> vy){
```

[13] https://clang.llvm.org/docs/ClangFormat.html
[14] http://www.stack.nl/~dimitri/doxygen/

```
  _x = cubic_b_spline<double>(v_x.begin(), v_x.end(), 0, 1.0);
  _y = cubic_b_spline<double>(v_y.begin(), v_y.end(), 0, 1.0);
}
```

Listing 3.1: The constructor of the Baseframe. The implementation uses two insances of `boost::math::cubic_b_spline<double>` to represent a 2D cubic B-spline. The instances are created from a range of values (as a begin-end iterator pair), a initial value and a step size. `_x` and `_y` are members of the class `Baseframe` and can be used within the scope of an instance of the class.

After the members `_x` and `_y` are initialized, eq. (2.2) can be implemented (see listing 3.2). This simple getter function queries the members `_x` and `_y` for their value at the given parameter packages them as a `boost::geometry::point` and returns a copy of the object.

```
#include <boost/geometry/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

namespace bg = boost::geometry;
typedef bg::model::d2::point_xy<double> point;

point Baseframe::Q(const double t) const {
       return point(_x(t), _y(t));
 }
```

Listing 3.2: The 2D cubic B-spline Q is implemented as a simple getter which packages the values of the members `_x` and `_y` at the parameter `t` in an instace of `boost::geometry::model::d2::point_xy<double>`.

With the generic 2D-spline implemented, one needs to program the transformation for the arc-length parametrization. The transformation functions eq. (2.4) and eq. (2.5) have to be computed numerically. First, we focus on implementing eq. (2.4), which consist of defining the function $\sqrt{x'^2 + y'^2}$ and numerically solving the integral. Thanks to `boost::math::cubic_b_spline` definition of the function is straightforward. The class offers the function `boost::math::cubic_b_spline::prime`, which returns the first derivate of the spline at a given point. The argument of the integration is defined with the use of C++14 lambdas (anonymous functor objects). From the number of available numerical integration methods, Adaptive Gaussian Quadrature (AGQ) is the most commonly used to solve this specific problem [GP90; Fre14; CLS12]. Adaptive methods are critical components for the speed and correctness (e.g., the error should be within acceptable range) of the computation. This work uses Adaptive Trapezoidal Quadrature (ATQ) [TW14] to solve the integral numerically. The method has comparable performance to AGQ and is provided by `boost::math::quadrature::trapezoidial`.

Listing 3.3 shows the implementation of the transformation function A using ATQ and C++14 anonymous lambda functors.

```
#include <boost/math/quadrature/trapezoidal.hpp>

using boost::math::quadrature::trapezoidal;

double Baseframe::A(const double t) const {
  // ... sanity checking and value lookup in lookup tables
  auto f = [this](double t) {
    return sqrt(pow(_x.prime(t), 2) + pow(_y.prime(t), 2));
  };
  result = trapezoidal(f, 0.0, t);
  // ... save results to lookup tables
  return result;
}
```

Listing 3.3: The implementation of `Baseframe::A` uses a numerical integration method implemented in `boost::math::quadrature::trapezoidal`. The function to integrate is implemented as an anonymous C++14-style lambda which is passed to the integrator. To reduce the computational complexity of the function the result of the integration is stored in a lookup table and retrieved when the function is called for the same parameter value. In favor of simplicity and shortness of the listing the implementation of the lookup table is omitted. For a discrete implementation of the full function the interested reader is referred to the documentation (see C).

Computing the inverse function of A also calls for a numerical solution. Finding the inverse $A^{-1}(s)$ of $A(t)$ for s is equal to finding the root of $f(t) = A(t) - s$.

Again, Boost implements a variety numerical root finding methods (with or without deviates) that can be utilized to solve this problem. The Bisection-method[15] is picked from the available options because it is commonly used in the literature to solve this problem. Bisection is known to be reliable and numerically stable, without getting stuck in local minima [WKA03]. However, experiments during implementation showed that other available methods (i.e., TOMS-748[16] and Bracket and Solve[17]) perform comparably to the bisection-method in runtime and performance. Listing 3.4 shows the implementation of the inverse transformation function using C++14 lambdas and `boost::math::root::bisection`.

[15] http://www.boost.org/doc/libs/1_65_1/libs/math/doc/html/math_toolkit/roots/roots_noderiv/bisect.html
[16] http://www.boost.org/doc/libs/1_65_1/libs/math/doc/html/math_toolkit/roots/roots_noderiv/TOMS748.html
[17] http://www.boost.org/doc/libs/1_65_1/libs/math/doc/html/math_toolkit/roots/roots_noderiv/bracket_solve.html

```
#include <boost/math/tools/roots.hpp>

using boost::math::roots::bisect;

double Baseframe::inv_A(const double s) const {
  // ... sainity checking and table lookup ...

  std::uintmax_t iterations = 1000;
  auto low = std::lower_bound(_segment_lut.begin(), _segment_lut.end(), s);
  const double segment = low - _segment_lut.begin();
  auto f = [this, s](double t) { return A(t) - s; };
  double result = bisect(f, segment - 1, segment,
                         eps_tolerance<float>(), iterations).first;

  // ... add result to lookup table

  return result;
}
```

Listing 3.4: The implementation of the inverse transformation A^{-1} uses `boost::math::roots::bisect` and C++ lambdas to find the root of $A(t) - s$. The computational complexity of the bisection-method is dependent on the boundaries of the search. A lookup table `_section_lut`, which stores the values of the function A at the knots of the cubic B-spline is used to determine the boundaries of the search. To reduce the computational complexity of the function a sRT-LUT is used to retrieve already calculated values. The discrete implementation of the sRT-LUT is omitted in favor of simplicity and size of this listing. The interested reader is refered to the documentation (see appendix C).

Now, with `Baseframe::inv_A` implemented, the heavy lifting is done. Implementing the arc-length parametrized cubic spline is not more complicated than putting `Baseframe::Q` and `Baseframe::inv_A` together. This solution appears natural, since it is equivalent to substituting eq. (2.5) in eq. (2.2) to yield eq. (2.6).

```
point Baseframe::P(const double s) const { return Q(inv_A(s)); }
```

Listing 3.5: The arc-length parametrization of the cubic b-spline is implemented by combining `Baseframe::Q` and `Baseframe::inv_A` in a simple getter function.

However, this described implementation contains a significant pitfall - assuming that the provided numerical integration has a computational complexity of $\mathcal{O}(n^2)$,

the computational complexity of the transformation function also has $\mathcal{O}(n^2)$. Since this function is repeatedly called from the root-finding algorithm to find A^{-1}, a call to `Baseframe::P` as a computational complexity of at least $\mathcal{O}(n^3)$. This is a severe performance bottle-neck and would result in unacceptable computation times. This work uses a programming technique to solve this problem: sRT-LUTs. With the use of sRT-LUTs it was possible to reduce the computational complexity of `Baseframe::Q` to at least $\mathcal{O}(n^3 \log(n))$.

3.5. Coordinate Transformation

Section 2.4 described the transformation between the local Cartesian and the Curvilinear coordinate system. This transformation is found by numerical minimization of the squared distance between the point in Cartesian Coordinates and the base fame. One can find the minimum of a continuous non-linear function by finding the root of its first derivative with respect to the parameter that one wishes to minimize. Equation (3.1) gives the first derivative of the squared distance between the point $\boldsymbol{x}$ and the arc-length parametrized spline curve $\boldsymbol{P}(s)$ with respect to s.

$$\frac{\mathrm{d}}{\mathrm{d}s} d(s)^2 = 2(x - x_b(s))x_b'(s) + 2(y - y_b(s))y_b'(s) \tag{3.1}$$

Once again the root finding algorithms of `boost::math::roots` come to use. [WKA02] suggests using the iterative Newton-Raphson method, since it rapidly converges to a solution. Newton-Raphson finds the root of a continuous differentiable function $f(x)$ by iteratively executing eq. (3.2). The idea is to linearize the function at an origin (i.e., finding the tangent) and taking the root of this tangent as an improved approximation of the root of $f(x)$. Given an initial guess x_0, this step is iteratively repeated until a tolerance is reached.

$$x_{N+1} = x_N - \frac{f(x)}{f'(x)} \tag{3.2}$$

As one can see from eq. (3.2) and is known from calculus, the tangent of $f(x)$ is defined as the first derivative with respect to x, $f'(x)$. Therefore to use the Newton-Raphson-Method to minimize the squared distance eq. (2.13) the first and second derivatives of $d^2(s)$. Equation (3.3) gives the second derivative of the function.

$$\frac{\mathrm{d}^2}{\mathrm{d}s^2} d(s)^2 = 2((x - x_b(s))x_b''(s) + x_b'^2(s) + (y - y_b(s))y_b''(s) + y_b'^2(s)) \tag{3.3}$$

However, the performance of the method is dependent on the quality of the initial guess. For a poor initial guess, the method can get stuck in local minima or diverge. [Jin+08] suggests a numerically stable method for finding the closest point on a spline curve. However, the implementation of this is out of the scope of this work. In this work the initial guess for the transformation is the extrapolated position of the vehicle on

the spline curve, given the current velocity v and passed time interval Δt and the last known position on the spline curve in meters s_{last}.

$$s_{\text{init}} = s_{\text{last}} + v\Delta t \tag{3.4}$$

The implementation (see listing 3.6) uses `boost::math::roots::newton_raphson_iterate` to find the transformation between Cartesian and curvilinear coordinates. This function takes the following arguments: A callable functor object that, given a parameter x returns the values of $f(x)$ and $f'(x)$, the lower and upper bound of the domain in which the algorithm should search for the solution, the initial guess x_0 for the function, the desired error tolerance, and the maximum allowed iterations.

```
std::pair<double, double> Baseframe::localize(const point position) const {
  auto D_prime = [ this, x_0 = position.x(), y_0 = position.y() ](double s)
  {
    // ... getting x, y, x_prime, y_prime, x_pprime, y_pprime ...
    double dist_prime = 2 * (x - x_0) * x_prime + 2 * (y - y_0) * y_prime;
    double dist_pprime = 2 * ((x - x_0) * x_pprime + pow(x_prime, 2) +
                              (y - y_0) * y_pprime + pow(y_prime, 2));
    return std::pair<double, double>(dist_prime, dist_pprime);
  };
  // ... stetting the barriers and initial guess
  auto result = newton_raphson_iterate(
      D_prime, lower_barrier, initial_guess, upper_barrier,
      std::numeric_limits<double>::digits / 2, iterations);
  return std::make_pair(result, bg::distance(P(result), position));
}
```

Listing 3.6: The transformation between cartesian and curvilinear coordinates uses `boost::math::roots::bisect::newton_raphson_iterate` and anonymous lambdas to minimize the distance function.

3.6. Path Generation

A Maneuver is represented as a cubic polynomial in the curvilinear coordinate system (see section 3.6). The shape of the polynomial is determined by its coefficients. The coefficients are found by solving the linear system that results from the boundary conditions eqs. (2.18) to (2.21) in eq. (3.5).

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ \Delta s_f^3 & \Delta s_f^2 & \Delta s_f & 1 \\ 0 & 0 & 1 & 0 \\ 3\Delta s_f^2 & 2\Delta s_f^2 & 1 & 0 \end{bmatrix} \begin{bmatrix} q_i \\ q_f \\ \tan(\theta) \\ 0 \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \tag{3.5}$$

This system is easily solved by inverting the Matrix A.

$$\boldsymbol{A}\boldsymbol{b} = \boldsymbol{c} \rightarrow \boldsymbol{b} = \boldsymbol{c}\boldsymbol{A}^{-1} \tag{3.6}$$

One can directly see in eq. (3.5) that the coefficient a is always going to be 0 and the coefficient c is always going to be $\tan(\theta)$. This finding may be used to simplify the system. However, in the implementation presented by this work, the full system is solved, and the values of a and c are asserted to be 0 and $\tan(\theta)$ as a sanity check. The implementation in listing 3.7 declares the linear system using `Eigen::Matrix4d` and `Eigen::Vector4d` and solves the system using `Eigen::Matrix4d::inverse()`. The coefficients `_a, _b, _c, _d` are private members of the class `Maneuver` and are set during construction of an instance of the class.

```
#include <eigen3/Eigen/Dense>
Maneuver::Maneuver(double velocity, double q_i, double q_f)
  const double delta_s_f = v_gain * velocity + min_length;
  Eigen::Matrix4d A;
  Eigen::Vector4d b, c;
  A << 0, 0, 0, 1, // first row
      pow(delta_s_f, 3), pow(delta_s_f, 2), delta_s_f, 1, // second row
      0, 0, 1, 0, // third row
      3 * pow(delta_s_f, 2), 2 * delta_s_f, 1, 0; // fourth row
  c << q_i, q_f, tan(_theta), 0.0;
  b = A.inverse() * c;
  _a = b(0), _b = b(1), _c = b(2), _d = b(3);
```

Listing 3.7: The coefficients of the cubic polynomial of a maneuver are calculated in its constructor. `Eigen::Dense` is used to solve the linear system, and the results are stored in member variables `_a,_b,_c` and `_d`.

With the coefficients found the polynomial, its derivate, and curvature can be computed (see eqs. (2.15) to (2.17)). The implementation of those equations is trivial and won't be discussed here. For an in-depth discussion of the implementation of `Maneuver::q`, `Maneuver::q_prime`, `Maneuver::q_pprime` and `Maneuver::curvature` the interested reader is referred to the documentation (see appendix C.4).

The discretized path in the configuration space is found by numerically solving the ODEs of the kinematic model of vehicle motion in eq. (2.26). The introduced ATQ from `boost::math::quadrature::trapezoidal` is unsuited for this problem, since it does not allow stepwise solving the integral while storing the intermediate results. However `boost::numeric::odeint`[18] provides some sophisticated numerical solvers for ODEs. One can choose between adaptive and constant integration methods and a variety of numerical steppers. While `boost::numeric::odeint` implements some higher order

[18] http://www.boost.org/doc/libs/1_65_1/libs/numeric/odeint/doc/html/index.html

integrators (like the Runge-Kutta method), this work found that the classical first-order Euler integration provides excellent performance with a tolerable error. The system of ODE is implemented as a C++14 lambda and passed to the numerical integrator along with the stepper method, initial value and integration boundaries. The values are stored in private member `Maneuver::_path` which is an instance of `std::vector<state>`. Listing 3.8 shows the implementation of the forward integration of the kinematic model.

```
#include <boost/math/numeric/odeint.hpp>
std::vector<pose> Maneuver::path() {
    std::vector<double> x_0 = {_position.x(), _position.y(), _heading};
    auto system = [this](const std::vector<double> &x,
            std::vector<double> &dxds, const double s) {
      double K_b{_baseframe->curvature(s)}, K{curvature(s)};
      double Q_s{Q(s)}, theta{x[2]};

      dxds[0] = Q_s * cos(theta);
      dxds[1] = Q_s * sin(theta);
      dxds[2] = Q_s * K;
    };
    boost::numeric::odeint::euler<state_type> stepper;
    boost::numeric::odeint::integrate_const(
        stepper, system, x_0, s_i(), s_f(), FLAGS_path_granularity,
        push_back_state_and_arc_length(_path));
    return _path;
}
```

Listing 3.8: `boost::numeric::odeint` is used to forward integrate the vehicle model, which is defined as an anonymous lambda. The inital state of the model is given as an `std::vector<double>`. Along with the integration boundaries and the integration stepper, it is passed to the integration method. An observer pushes the steps of the integration in a member `Maneuver::_path`, which is returned by the function.

3.7. Collision Checking

The discrete path in the Cartesian Coordinate system has to be checked for its feasibility. The violation of holonomic-constraints of vehicle motion is a trivial task and can be checked by iterating over the generated path performing the necessary calculations. However, checking feasibility regarding collisions in the workspace is a complex task and calls for a sophisticated solution. The collision checking algorithm proposed by this work uses a spatial index to efficiently find obstacles that are at risk of colliding with the vehicle. To do so, one must describe the geometrical shape of

the vehicle and obstacles and their position in the Cartesian coordinate system. The library `boost::geometry` provides a number of types to represent geometrical primitives and complex shapes, i.e., `boost::geometry::box` or `boost::geometry::polygon`. Also, algorithms that solve geometrical problems are provided, i.e. translations, rotators, distance calculations and intersections of shapes. Further, the library implements a spatial index `boost::geometry::index::rtree`. The spatial index is managed and owned by the planner (see section 3.9) which updates its contents given new sensor information. Each instance of a maneuver can query the R-Tree using an observer that is implemented using `std::shared_ptr`. Listing 3.9 provides the implementation of algorithm 1.

```
#include<boost/geometry/geometry.hpp>
#include<boost/geometry/index/rtree.hpp>
namespace bg = boost::geometry, bgi = boost::geometry::index;
double Maneuver::collision() {
  _collision_length = 0.0; _collision_checked = true; Car car;

  for (const auto &p : path()) {
    std::vector<value> risks;
    _rtree->query(bgi::intersects(car.aabb(p)), std::back_inserter(risks));
    for (const auto &risk : collision_risk) {
      polygon obstacle{*risk.second};
      if (bg::intersects(car.obb(p), obstacle)) {
        _collision_length = p.s;
        return _collision_length;
      }
    }
  }
  return _collision_length;
}
```

Listing 3.9: The collision checking function uses `boost::geometry::index::rtree` to find geometries that intersect with the bounding box of the car. The geometry of the car is implemented in the class `planner::Car`, which defines getters for the AABB and OBB at a vehicle pose.

3.8. Objective Function

The objective function is a weighted sum of three cost functions (see eq. (2.33)). The implementation of the objective function itself is trivial and does not require explanation (see listing 3.10),

```
double Maneuver::J(const Maneuver &previous,
                   const std::shared_ptr<std::vector<Maneuver>> maneuvers,
                   double w_s, double w_k, double w_c) const {
  return w_s * C_s(maneuvers) + w_k * C_k() + w_c * C_c(previous);
}
```

Listing 3.10: The objective function is implemented as a weighted sum of the cost functions. The weights w_s,w_κ,w_c can be set using command line flags (see section 3.10).

The contrary is the case for the cost functions, which require costly numerical computation. The implementation of the smoothness and consistency cost functions is relatively straightforward, however. Using ATQ (see section 3.4) the integral eq. (2.35) can be numerically solved. Listing 3.11 shows the source code of the implementation of the smoothness cost functional. The implementation of the consistency cost is shown in Listing 3.12.

```
double Maneuver::C_k() const {
  auto l = [this](const double s) { return pow(curvature(s), 2) * Q(s); };
  return trapezoidal(l, s_i(), s_f());
}
```

Listing 3.11: `boost::math::quadrature::trapezoidal` is used to numerically integrate the curvature over the length of a maneuver to calculate the smoothness cost.

```
double Maneuver::C_c(const Maneuver &previous) const {
  double s_1 = s_i(), s_2 = s_f();
  auto l = [this, &previous](double s) {
    point a(s, q(s));
    point b(s, previous.q(s));
    return bg::distance(a, b);
  };
  double result = 1 / (s_2 - s_1) * trapezoidal(l, s_1, s_2);
  return result;
}
```

Listing 3.12: The consistency cost functional is implemented using lambdas and `boost::math::quadrature::trapezoidal` to integrate the difference in offset for the overlapping section of the maneuvers.

The safety cost functional requires the implementation of a normal probabilistic density function (PDF) to solve the discrete Gaussian convolution in eq. (2.34). An implementation of a PDF for a Gaussian normal distribution is provided by `boost::math::normal`. The discrete convolution can be performed using the implemented collision checking function (see section 3.7) and the PDF from boost. Listing 3.13 shows the implementation.

```
#include<boost/math/distribution/normal.hpp>.
using boost::math::normal;
double Maneuver::C_s(std::shared_ptr<std::vector<Maneuver>> maneuvers,
                     double sigma) const {
  double result = 0.0;
  normal g(q_f(), sigma);

  for (auto &m : *maneuvers) {
    result += static_cast<bool>(m.collision()) * pdf(g, m.q_f());
  }

  return result;
}
```

Listing 3.13: The discrete Gaussian convolution which calculates the safety cost functional is implemented using `boost::math::normal`. The standard deviation of the normal distribution can be set through a design parameter (see section 3.10).

3.9. Plan Update

With the implementation of the baseframe, transformation functions, maneuvers, collision detection and objective function in place, one can implement algorithm 2. The implementation in listing 3.14 utilizes the STL container `std::map`. Map is an ordered associative container, which means that the order in which the values are internally stored depends on the values of the key of the key-value pair. This property can be utilized to find the path from the set of feasible maneuvers that minimizes the objective function. Since the container is ordered (from lowest to highest value), the path with the minimal cost is the one at the front of the container. Equivalently, if no feasible path is available (all paths are collision paths), the path that maximizes the length in the free configuration space can be found at the back of the data-structure. Listing 3.14 shows the source code of the implementation.

```
const Maneuver &Planner::update(point position, double heading,
       double velocity, std::vector<polygon> obstacles) {
```

```
  _maneuvers.clear(); add_to_spatial_index(obstacles);
  std::pair<double, double> p =
      _baseframe->localize(position, velocity, heading, dt, _last_known_s);
  _last_known_s = p.first; const double q_i = p.second;

  for (double q_f = -_q_max; q_f <= _q_max; q_f += (2 * _q_max / _m_max)) {
    Maneuver m = Maneuver(position, heading, velocity, q_i, q_f,
    _last_known_s, _baseframe, _rtree);
    if (m.driveable()) _maneuvers.push_back(m);
  }

  std::map<double, Maneuver> candidates, collisions_paths;

  for (auto &m : _maneuvers) {
    if (m.collision()) {
      collisions_paths.insert(std::make_pair(m.collision(), m));
    } else {
      double cost =
          m.J(_maneuvers,
                  std::make_shared<std::vector<Maneuver>>(_maneuvers));
      candidates.insert(std::make_pair(cost, m));
    }
  }

  _maneuver = candidates.empty() ? collisions_paths.rbegin()->second
                                 : candidates.begin()->second;

  return _maneuver;
}
```

Listing 3.14: The function `Planner::update` uses the implemented functions of `Baseframe` and `Maneuver`. Obstacles, which are passed to the function are added to the R-tree. The vehicle is localized in the curvilinear coordinate system, and the candidate set of maneuvers is initialized. Maneuvers that violate non-holonomic constraints get discarded. From the set of maneuvers the one that minimizes the objective function is picked.

3.10. Design Parameters

Some parameters of the program can be used to influence quantitative or qualitative aspects like runtime or optimality of the solution of the planner. In this work, command line flags are used to set such design parameters of the program. Command line

flags allow pass options to the program on calling it, without the need of recompiling the software or editing configuration files. The software library `gflags` is used to implement the options. The user of the software can edit the value of an option by adding the parameter name and desired value to the program call as consultive text, i.e.: `--name_of_flag=new_value`. Section 3.10 gives an overview of the design parameters of the program and the options used to set them from the command line.

symbol	default	type	equation / listing	flag
$q_{\max}$	4.0	`double`	listing 3.14	`--max_manouver_offset`
Δs	1.0	`double`	listing 3.8	`--granularity`
N	30	`int`	listing 3.14	`--number_manouvers`
$\Delta s_{\min}$	20	`double`	eq. (2.24),listing 3.7	`--min_manouver_length`
k_v	1.0	`double`	eq. (2.24),listing 3.7	`--manouver_speed_gain`
$\kappa_{\max}$	0.5	`double`	eq. (2.31)	`--max_curvature`
σ_s	1.0	`double`	eq. (2.34),listing 3.13	`--collision_standart_deviation`
w_s	1.0	`double`	eq. (2.33),listing 3.10	`--weight_safety`
w_κ	1.0	`double`	eq. (2.33),listing 3.10	`--weight_smoothness`
w_c	1.0	`double`	eq. (2.33),listing 3.10	`--weight_consistency`

Table 3.1.: List of design parameters. The design parameters of the program are implemented using the software library `gflags`. The user can set parameter values on program call, by passing the corresponding option and desired value as successive text.

4. Results

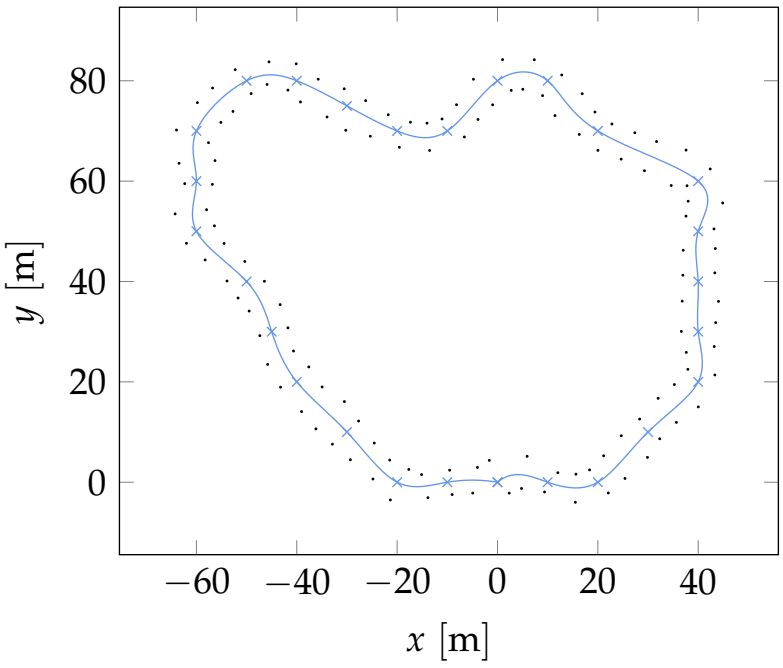
This chapter discusses the achieved results of this work. It introduces the methods that were used to measure the results and illustrates the effects of the cost functions and design parameters on the solution. Further, it provides performance benchmarks for the implemented algorithm and discusses scenarios that lead to a failure of the planner. Finally, the results are discussed and compared to different solutions from the literature.

4.1. Simulation

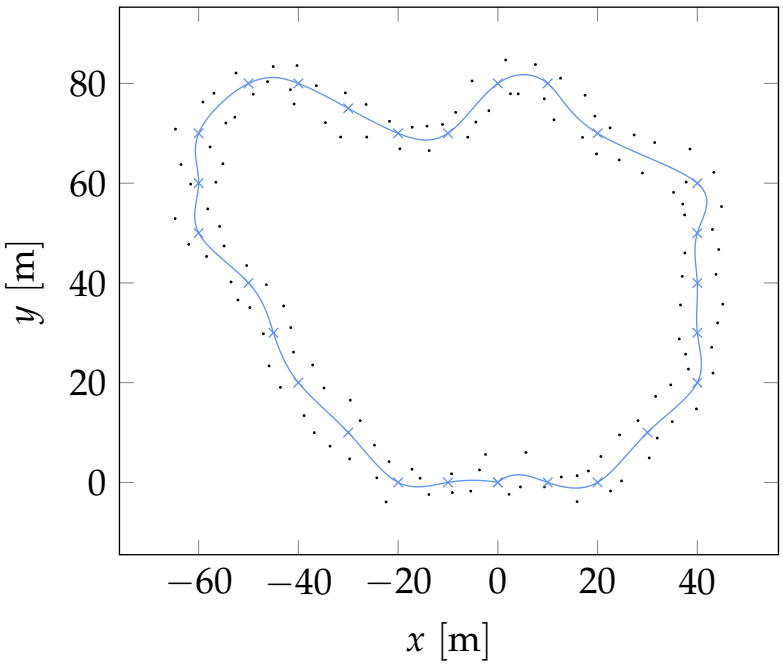
The software for the path planning algorithm was developed parallel to extensions on the general software framework and improvements to the hardware of the racecar. Despite all efforts, it was not possible to move the car to a functional state and perform a full integration test of the proposed software on the target hardware. Therefore another measure had to be found to ensure the correct working of the implementation. In this work, a simple SIL-simulation is used to test the behavior of the path planner for a scenario as described in section 1.2. In addition to the software-libraries described in section 3.2 the simulation uses the typesetting software LaTeX to render images and plots for every step of the simulation. The simulation itself is a simple C++ program which constructs a planner for a given set of way-points. The program will place polygons that represent line delimiter cones at equidistant points along the base-frame. The placement of the road-delimiters is then blurred using a normal-distributed random variable with a user-defined standard deviation. This represents the human-error during cone placement, since in a realistic scenario one can not expect perfect conditions [Lau17a]. Figure 4.1 illustrates the effect of the standard deviation σ on the cone placement.

The simulation then iteratively performs planning steps. The vehicle moves along the path and a normal-distributed random noise with a user-defined standard deviation is added to the configuration of the vehicle to represent imperfections in the controller and sensing. The simulation ends successfully when the vehicle reaches the end of the baseframe. Figure 4.2 illustrates the result of a simulation for an example scenario with an objective function J that equally weights the cost functions.

An animation of the full simulation can be viewed online (https://goo.gl/f6VoNN).

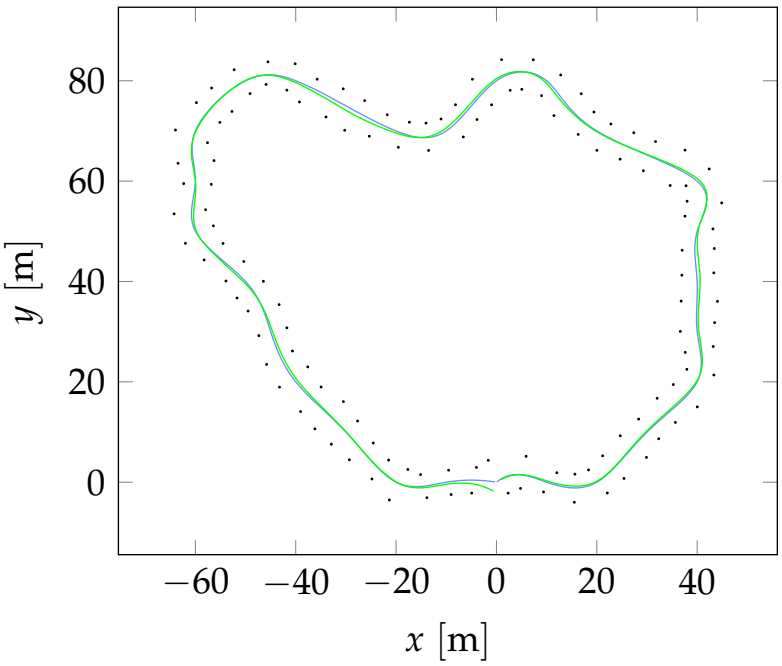


(a) $\sigma = 0.5$

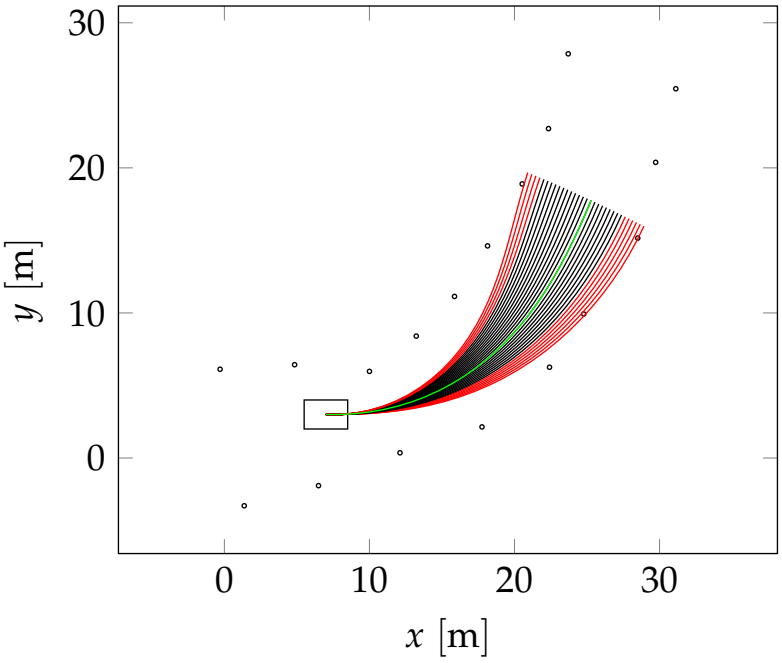


(b) $\sigma = 1$

Figure 4.1.: Effects of the standard deviation on cone placement. Road delimiter cones are marked as black polygons, the way-points and generated baseframe of the planner are drawn in blue. The road delimiter cones are placed with additional normal distributed noise to model human introduced imperfections to the simulated race track.



(a) Simulated near-optimal path with $w_s = w_\kappa = w_c$



(b) Simulated maneuver selection with $w_s = w_\kappa = w_c$

Figure 4.2.: Simulated results for a planner with equally weighted cost functions $w_s = w_\kappa = w_c$. The road-delimiter cones are drawn as black polygons, the baseframe of the planner is drawn in blue. In figure fig. 4.2a the near-optimal path is drawn with green ink. An animation of the simulation can be viewed online (https://goo.gl/f6VoNN).

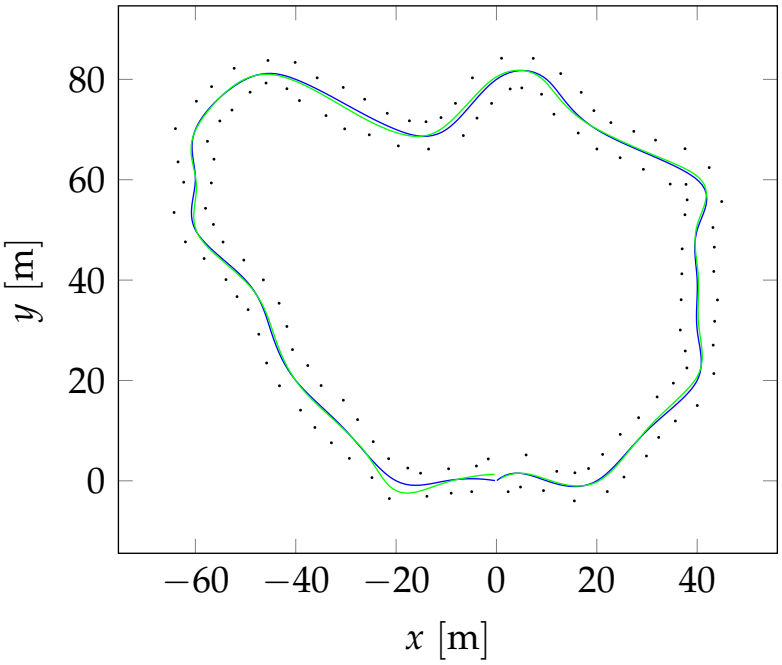
4.2. Effects of Cost Functions

The shape of the near-optimal path is dependent on the choice of the cost function described in section 2.8. In a real-world racing scenario, the path may be optimized by carefully tuning the weights of the objective function to maximize performance. However, first one needs to understand the addition to the path and workings of the cost functions. SIL-simulation is used to simulate paths through an example track with distinct objective functions that rely entirely on a specific cost. The results of this simulation are shown in figs. 4.3 to 4.5. Animations of the simulations can be viewed online:

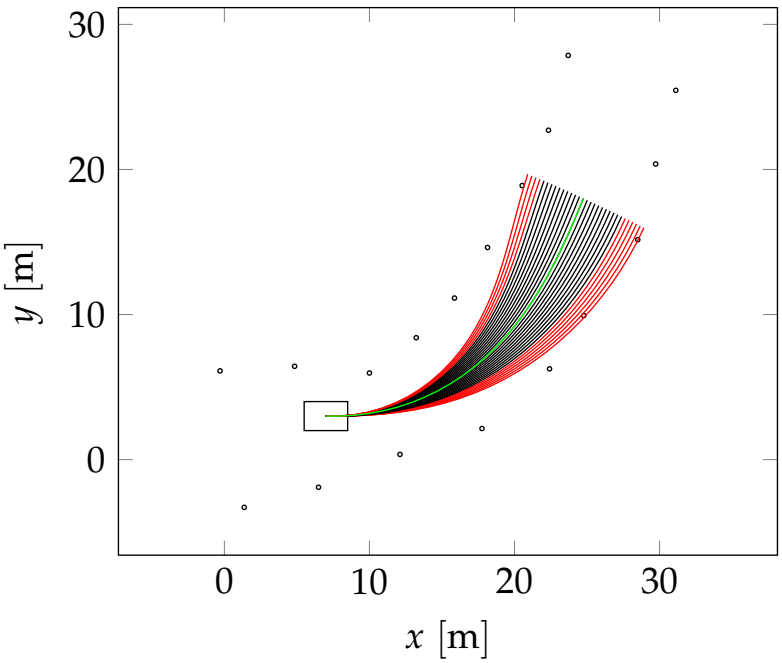
- For an objective function that only considers safety (https://goo.gl/pNUqP8)
- For an objective function that only considers smoothness (https://goo.gl/cUzKmv)
- For an objective function that only considers consistency (https://goo.gl/2WH8gM)

One can see in fig. 4.3a and fig. 4.3b that an objective function that only considers the safety cost C_s attempts to maximize the distance to obstacles. For a near-perfect racetrack with road delimiters that are relatively close to each other (in driving direction), such a planer will favor a path that is close to the middle of the road. However, an objective function that only considers safety may cause the planning to fail for imperfect scenarios (see section 4.5). Figure 4.4a and fig. 4.4b show the effects of an objective function that only considers the smoothness of a path. As one may expect, the resulting path is close to a *racing*-line through the track. While this allows higher velocities during a track drive, again there is a possibility that such an objective function may leave the racetrack. In contrast to an objective that only considers safety, this does not necessarily lead to total failure of the planner. An objective function that only favors consistency is shown in Figure 4.4a and fig. 4.4b. In this case, the planner favors paths that are closer to those produced during previous planning iterations. The effect of this is that the final path is closer to the baseframe and tends not to leave the racetrack. Deviations from the base-frame occur in areas of high curvature or to avoid obstacles.

The consistency cost serves as an equalizer to the safety and smoothness. Therefore the cost functions have to be considered in unison. For optimal performance, the weights of the objective function have to be chosen carefully and with the scenario at hand in mind.

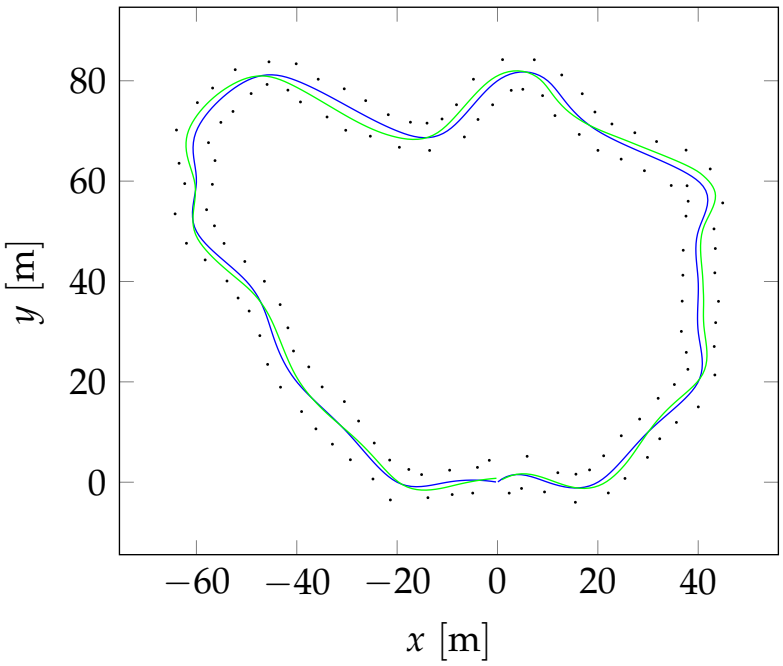


(a) Near-Optimal path for $w_s = 1,\ w_\kappa = 0,\ w_c = 0$

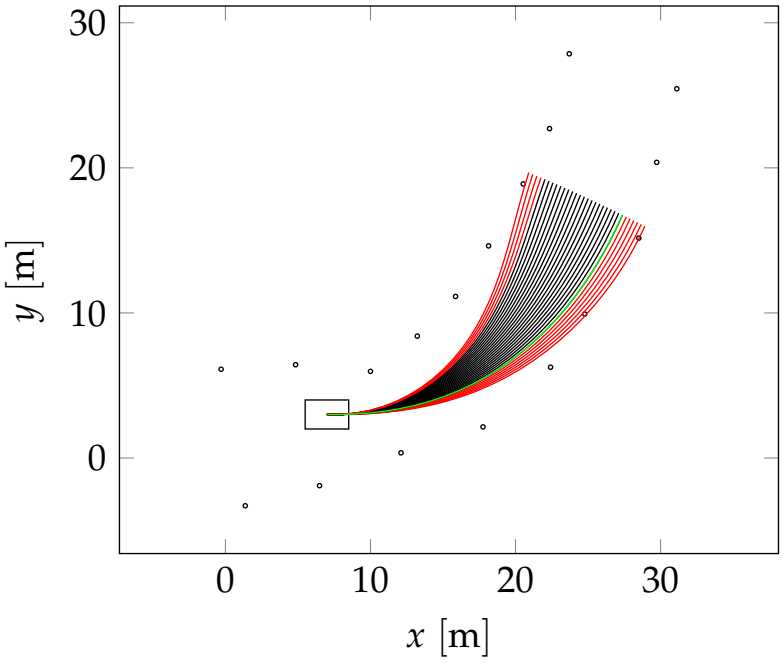


(b) Maneuver selection for $w_s = 1,\ w_\kappa = 0,\ w_c = 0$

Figure 4.3.: Effects of the weight of the safety cost function w_s. An objective function that only considers safety will try to maximize the distance to obstacles. An animation of the simulation can be viewed online (https://goo.gl/pNUqP8).

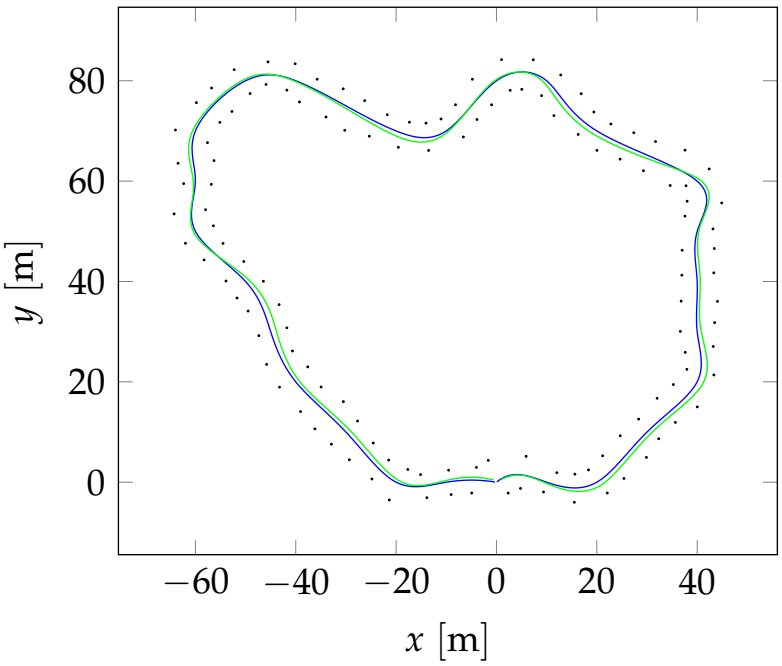


(a) Near-Optimal path for $w_s = 0,\ w_\kappa = 1,\ w_c = 0$

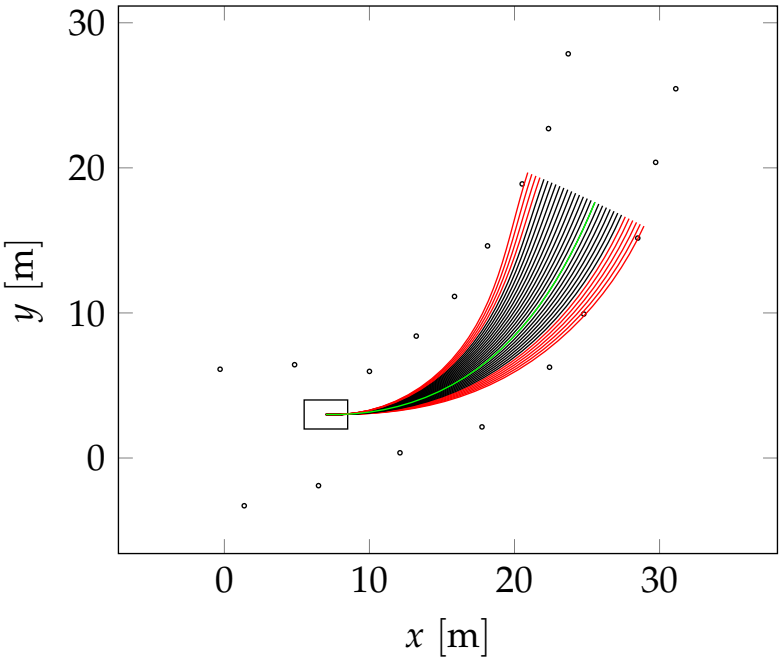


(b) Maneuver selection for $w_s = 0,\ w_\kappa = 1,\ w_c = 0$

Figure 4.4.: Effects of the weight of the smoothness cost function w_κ. If the objective function only considers smoothness, the resulting path will stay closer to a racing line. An animation of the simulation can be viewed online (https://goo.gl/cUzKmv).



(a) Near-Optimal path for $w_s = 0,\ w_\kappa = 0,\ w_c = 1$



(b) Maneuver selection for $w_s = 0,\ w_\kappa = 0,\ w_c = 1$

Figure 4.5.: Effects of the weight of the consistency cost function w_c. An objective that only considers the consistency cost tends to stay closer to the baseframe. An animation of the simulation can be viewed online (https://goo.gl/2WH8gM).

4.3. Effects of Design Parameters

Alongside the weights of the cost functions, one can use a set of design parameters (section 3.10) to adjust the behavior of the planner for the scenario at hand and tune the performance of the planner (see section 4.4).

This section discusses the influence of the design parameters of the behavior of the planner.

If the width of the road or racetrack is known beforehand, the maximum offset from the baseframe $q_{\max}$ can be used to achieve a uniform coverage of the relevant section of the configuration space while sampling. Figure 4.9 illustrates the effect of $q_{\max}$ on the coverage of the configuration space.

The granularity of the path generation Δs is closely related to the velocity of the vehicle and frequency of planning iterations. For example, let a vehicle travels with $1\,\mathrm{m\,s^{-1}}$ along a track and updates its planner with 10 Hz. In this case, a path that provides an exact solution would have a minimum resolution of 0.1 m, since that is the distance that the vehicle travels during a planning intervall. The granularity is used to adjust the resolution of the discrete path. A resolution that is too low might lead to a biased or false result of collision checking, since the path is only checked at the discrete points defined by the granularity. A low-resolution path that happens to be collision-free and feasible at its discrete points may lead to a collision in-between those discrete poses. It is essential to choose a granularity that is lower than the minimum length (or width) of the bounding box of the vehicle. A low-resolution also introduces a bigger error in the forward integration of the vehicle model. While this error could be solved by using a higher-order numerical integration methods, this would result in lower-runtime performance. While a high resolution is desired, the granularity has a severe impact on the runtime of a planning iteration (section 4.4) and therefore has to be picked wisely. Figure 4.10 illustrates the effect of the granularity on the resolution of the generated path.

The number of samples N again influences the coverage of the configuration space and is linked to the offset from the baseframe. A wider section (i.e., a larger $q_{\max}$) in general also requires more samples to achieve the same level of coverage. Figure 4.11 shows the effect of the number of samples on the coverage of the configuration space.

The distance that the planner plans ahead is influenced using the minimum maneuver length $\Delta s_{\min}$ and maneuver speed gain k_v. Both parameters are closely related and work in union to determine the length of the generated path in a planning iteration. How the parameters collaborate to determine the look-ahead distance of the planner is illustrated in figs. 4.12 and 4.13.

The standard deviation of the safety cost function σ_s is used to influence the acceptable width of gates for the vehicle to pass through. It can be used to minimize the risk of the vehicle to leave the track. Larger standard deviation will favor wider gates over smaller ones. However, this only works if the distance between the road delimiters in traveling distance is smaller than the width of the road. Otherwise a big σ_s will have the opposite effect and lead to the vehicle leaving the track.

The maximum curvature κ_{max} is used to set the maximum turning radius for the used vehicle. For example, the maximum turning radius of the racecar build by the REV-Team is 2 m, therefore the maximum possible curvature of this vehicle is $0.5\,\mathrm{m}^{-1}$.

4.4. Benchmarking

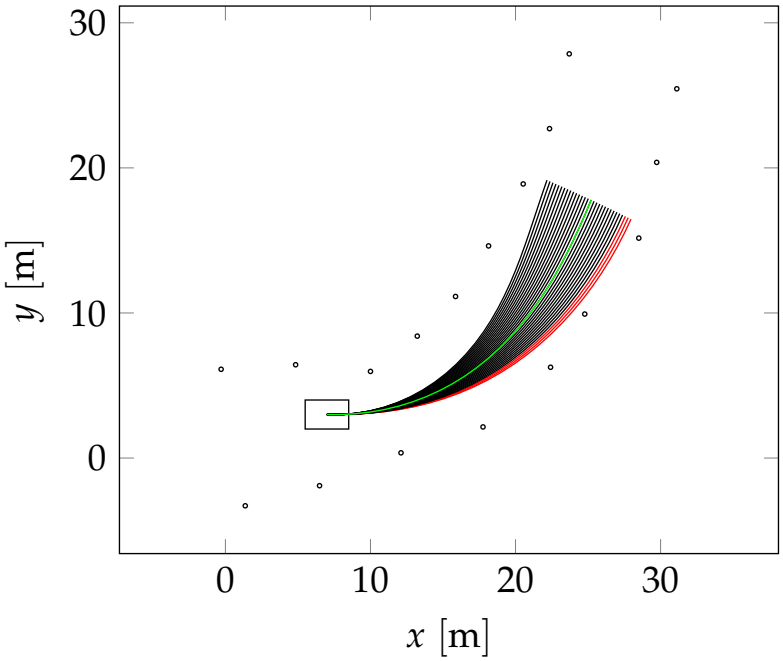
The presented implementation is developed to be applied for a highly dynamic scenario and on embedded hardware. Runtime performance, therefore, is a crucial feature concerning the usability. During an autonomous race, one may expect velocities up to $20\,\mathrm{m\,s}^{-1}$. An iterative planner that runs with a frequency of $\geq$20 Hz still allows updating the planning with a granularity of 1 m. Frequent updates with respect to distance traveled minimize the risk of collisions and increase the resolution of the planned path.

The environment and design parameters influence the runtime-performance of the planning algorithm. The parameters can be tuned to equalize the impact of the environment and achieve the desired update-frequency for the scenario at hand. However, one needs insight on the impact of environment and choice of parameters on the runtime-benchmark of a planning-iteration.

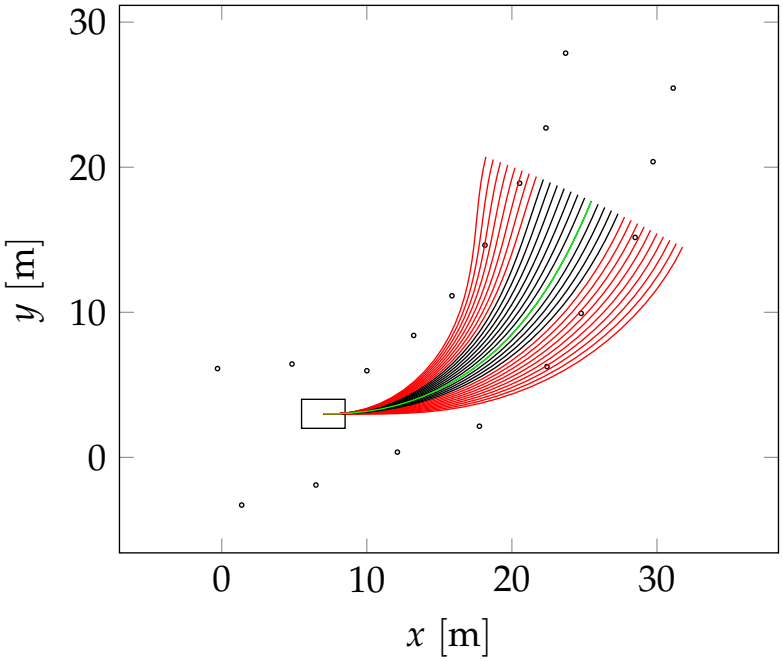
In this work, the statistical mean of the runtime of 100 planning iterations is used to produce a benchmark for a given scenario. To produce an unbiased and representative result all design-parameter-benchmarks where produced given the same scenario: A 1 km long straight track with delimiters on both sides. All design parameters, except the one considered in the benchmark, set to their default values. Benchmarks were produced for variations of all design parameters (see section 3.10), but it was found that only the granularity of the path Δs, the number of maneuvers sampled N, the minimum length of a maneuver Δs_{min} and the maneuver speed gain k_v have a direct impact on the benchmark.

As was stated before in section 2.7, collision checking takes up a significant part of the runtime of a program. Therefore, one may expect that the number of obstacles in the workspace has a considerable impact on the runtime of a planing-iteration. This is due to the increased complexity of collision checking. To produce a representative benchmark of the impact of the environment on the runtime, the following scenario was utilized: Again a 1 km long track and a planner with a default set of parameters are used. Now, some obstacles are sampled from an uniform distribution along the track, with an offset of $\pm$3 m to $\pm$ 5 m. Again the statistical mean of 100 planning iterations is used to produce the benchmark.

The benchmarks were produced using a 2.6 GHz Intel Core i5 Processor. Other processes on the hardware were suppressed during the recording of the benchmark so that the recording used 99 % of the available resources. The calculated benchmarks are shown in fig. 4.14.

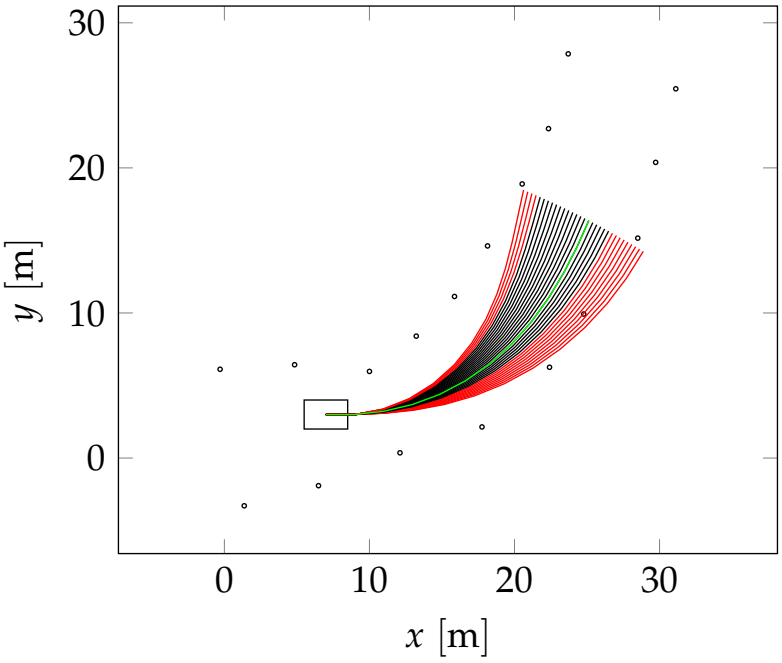


(a) $q_{\max} = 3\,\mathrm{m}$

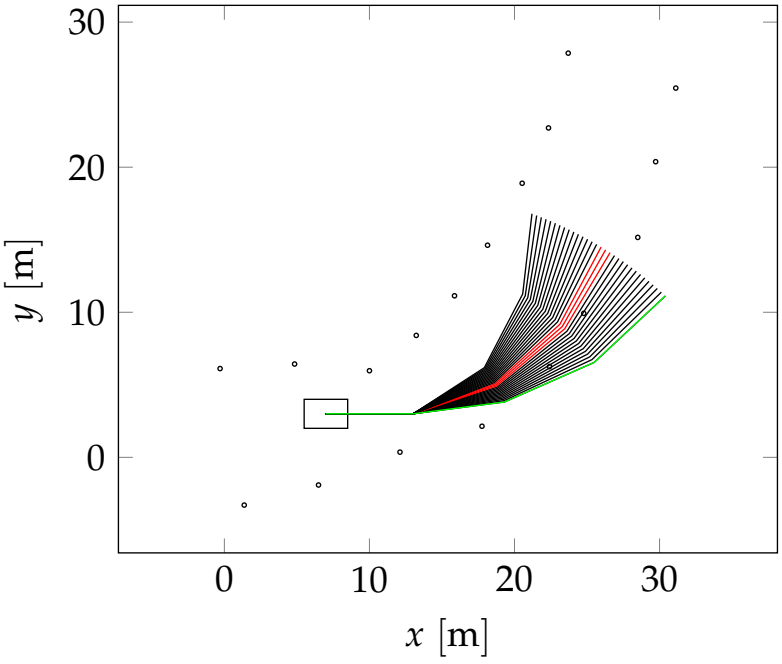


(b) $q_{\max} = 7\,\mathrm{m}$

Figure 4.6.: Effects of $q_{\max}$. The maximal offset from the baseframe is used to achieve good sampling coverage of the relevant section of $\mathcal{C}$. The offset influences the section of the road in which paths are sampled. Figure 4.9a illustrates sampling for a maximal offset of 3 m, Figure 4.9b for a maximal offset of 7 m.

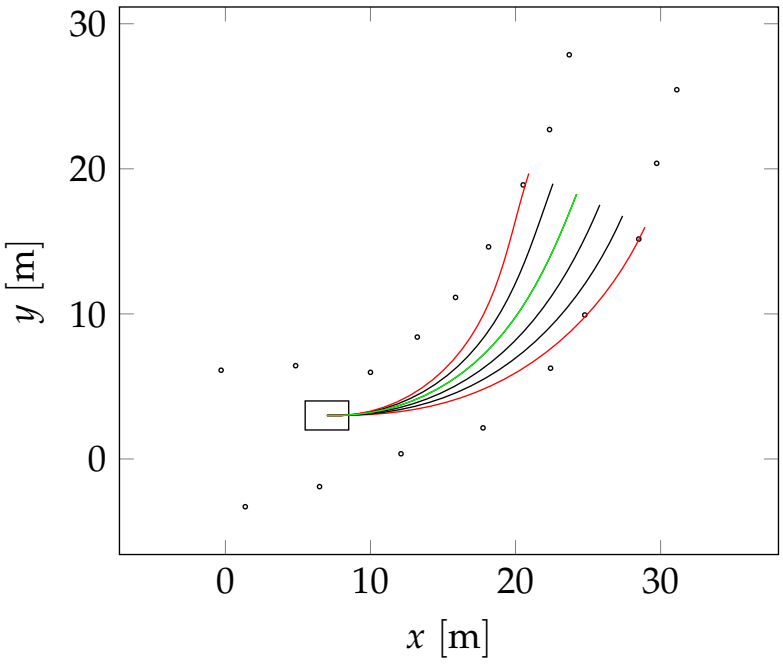


(a) $\Delta s = 2\,\mathrm{m}$

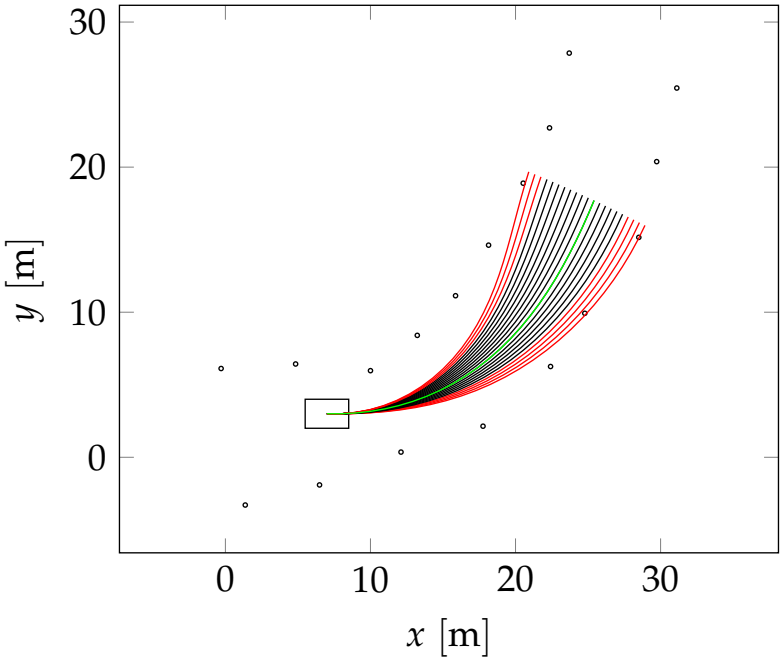


(b) $\Delta s = 6\,\mathrm{m}$

Figure 4.7.: Effects of Δs. The granularity has direct impact on the resolution of the generated path. A high resolution (i.e., $\Delta s = 2\,\mathrm{m}$, fig. 4.10a) produces geometrically reasonable paths with correct collision checking results, while resolution that is too low (i.e., $\Delta s = 6\,\mathrm{m}$, fig. 4.10b) may produce incorrect results during collision checking and introduce greater errors during generation of the discrete path.

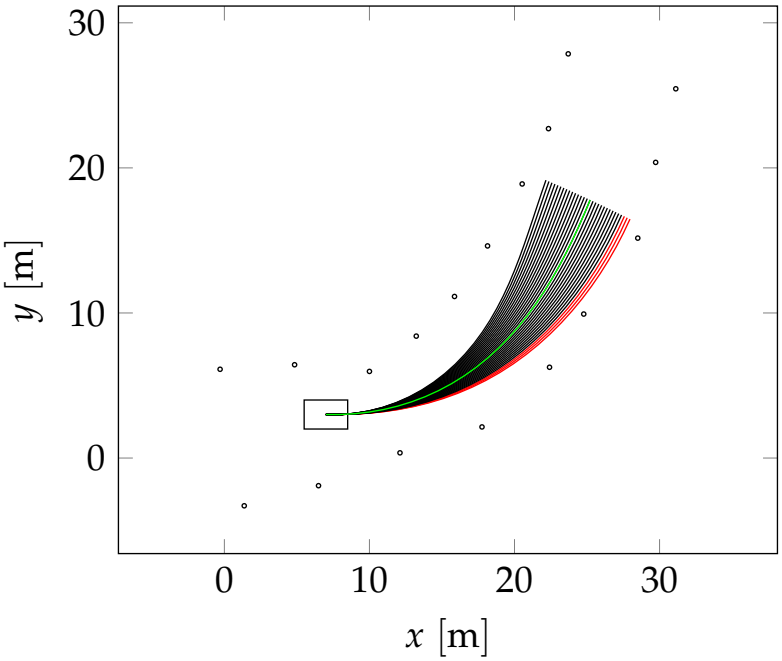


(a) $N = 6$

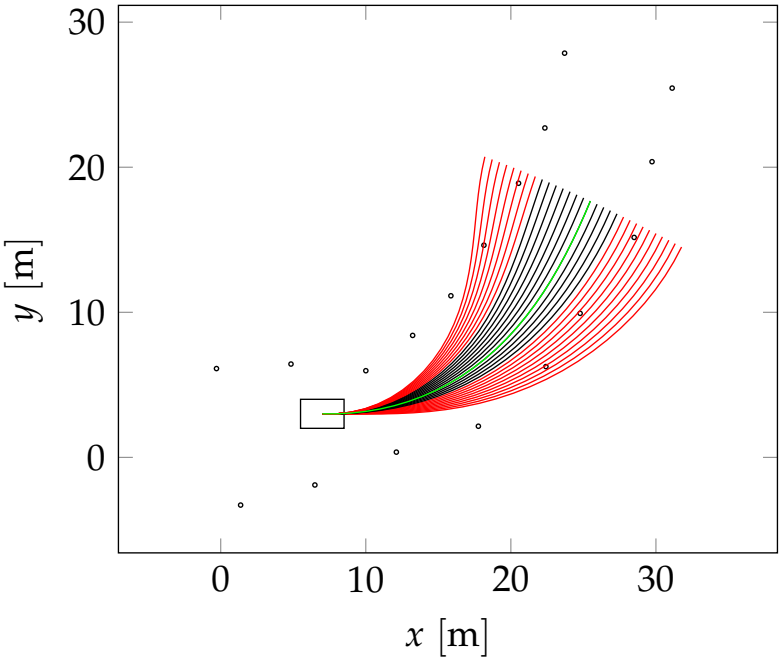


(b) $N = 20$

Figure 4.8.: Effect of N. The number of samples is linked to the resolution of the sampling coverage of the relevant section of $\mathcal{C}$ and the likelihood of finding a feasible path.

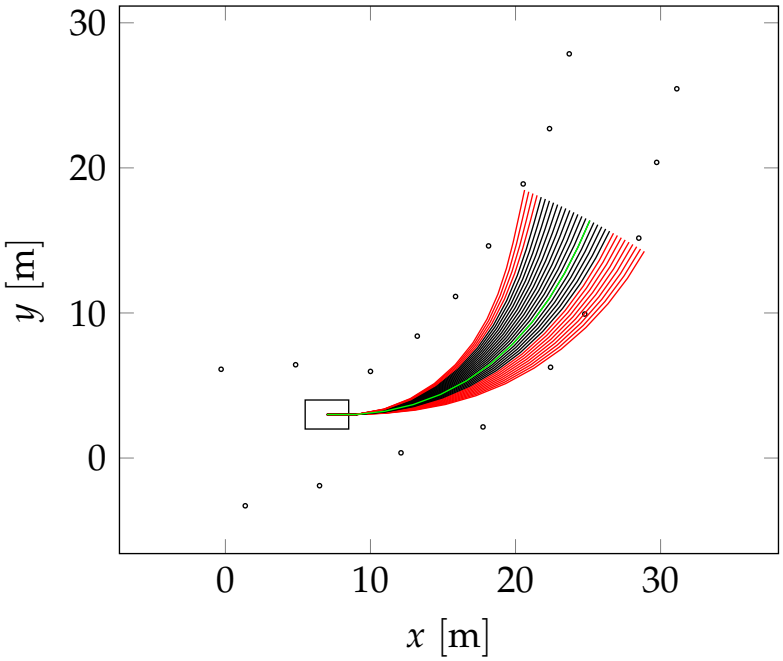


(a) $q_{\max} = 3\,\mathrm{m}$

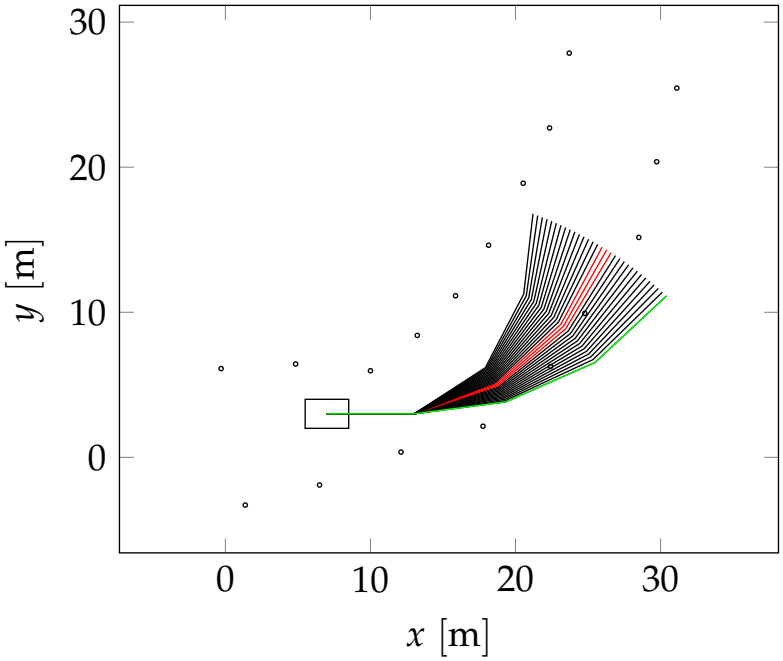


(b) $q_{\max} = 7\,\mathrm{m}$

Figure 4.9.: Effects of $q_{\max}$. The maximal offset from the baseframe is used to achieve good sampling coverage of the relevant section of $\mathcal{C}$. The offset influences the section of the road in which paths are sampled. Figure 4.9a illustrates sampling for a maximal offset of 3 m, Figure 4.9b for a maximal offset of 7 m.

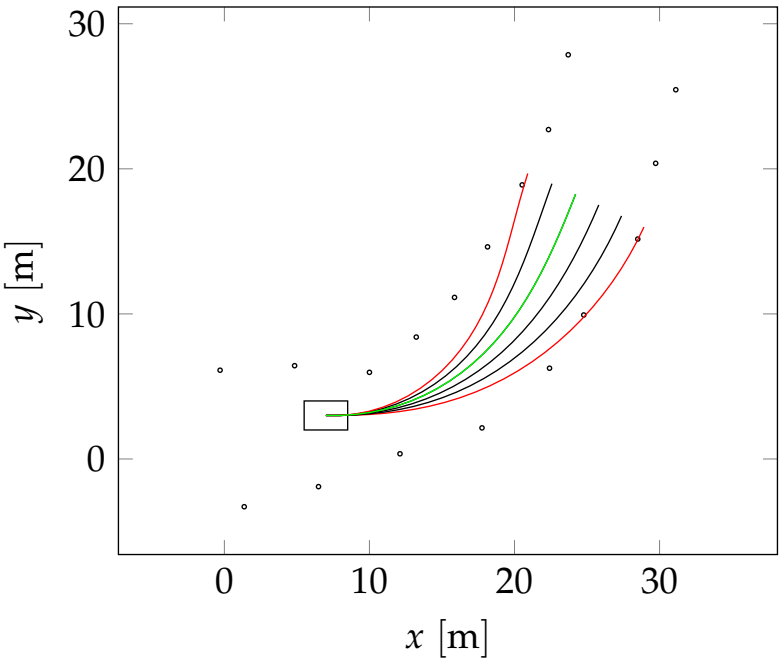


(a) $\Delta s = 2\,\mathrm{m}$

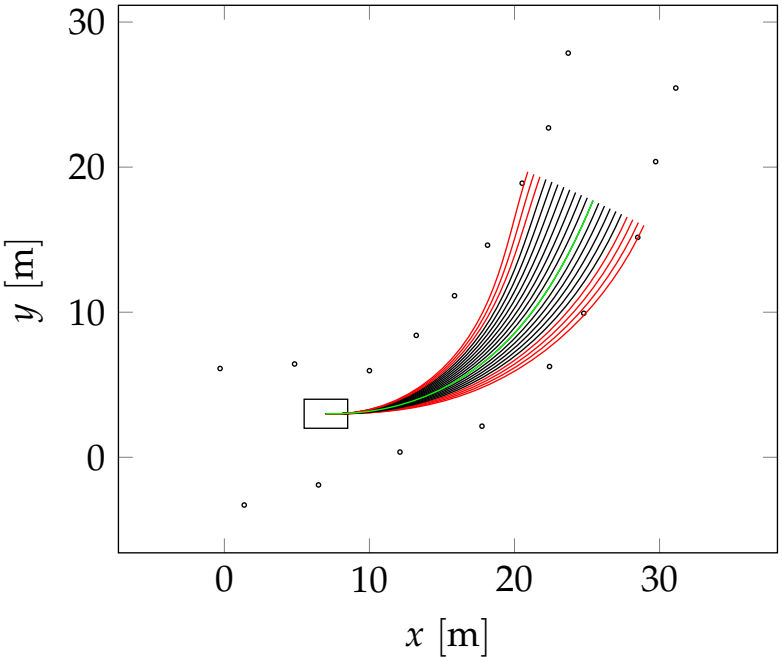


(b) $\Delta s = 6\,\mathrm{m}$

Figure 4.10.: Effects of Δs. The granularity has direct impact on the resolution of the generated path. A high resolution (i.e., $\Delta s = 2\,\mathrm{m}$, fig. 4.10a) produces geometrically reasonable paths with correct collision checking results, while resolution that is too low (i.e., $\Delta s = 6\,\mathrm{m}$, fig. 4.10b) may produce incorrect results during collision checking and introduce greater errors during generation of the discrete path.

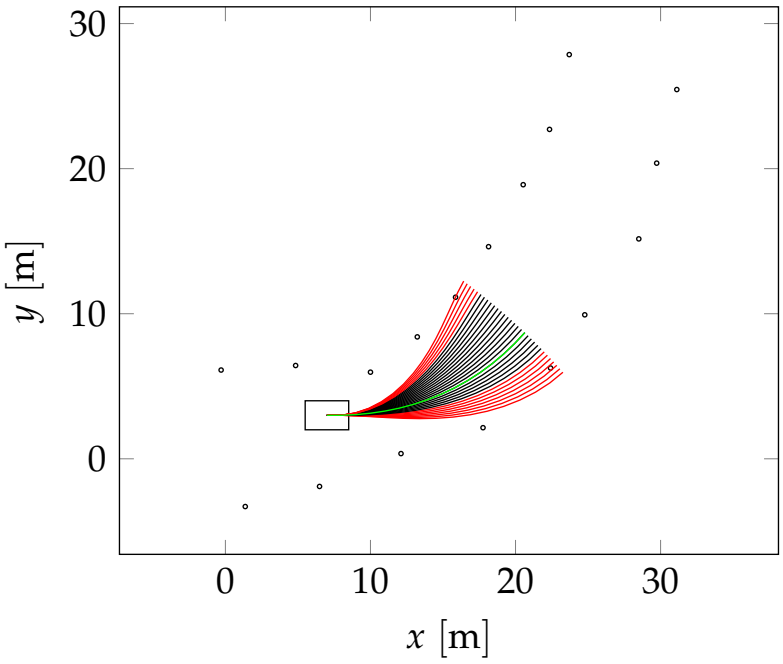


(a) $N = 6$

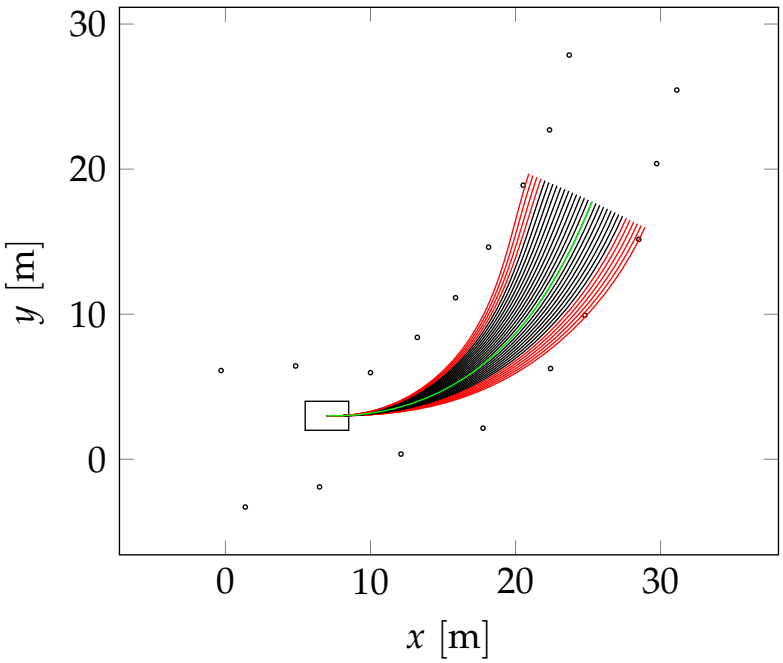


(b) $N = 20$

Figure 4.11.: Effect of N. The number of samples is linked to the resolution of the sampling coverage of the relevant section of $\mathcal{C}$ and the likelihood of finding a feasible path.

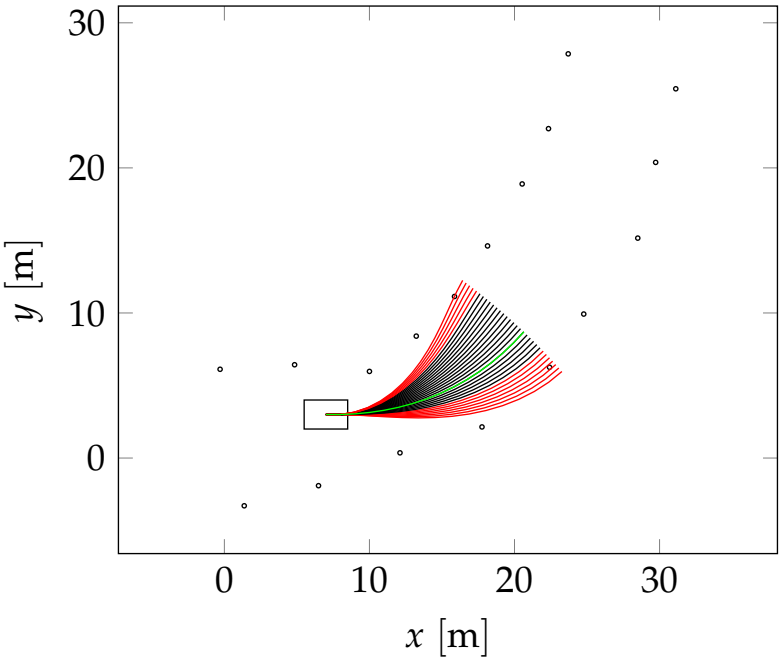


(a) $s_{\text{min}} = 5\,\text{m},\ k_v = 1,\ v = 5\,\text{m}\,\text{s}^{-1}$

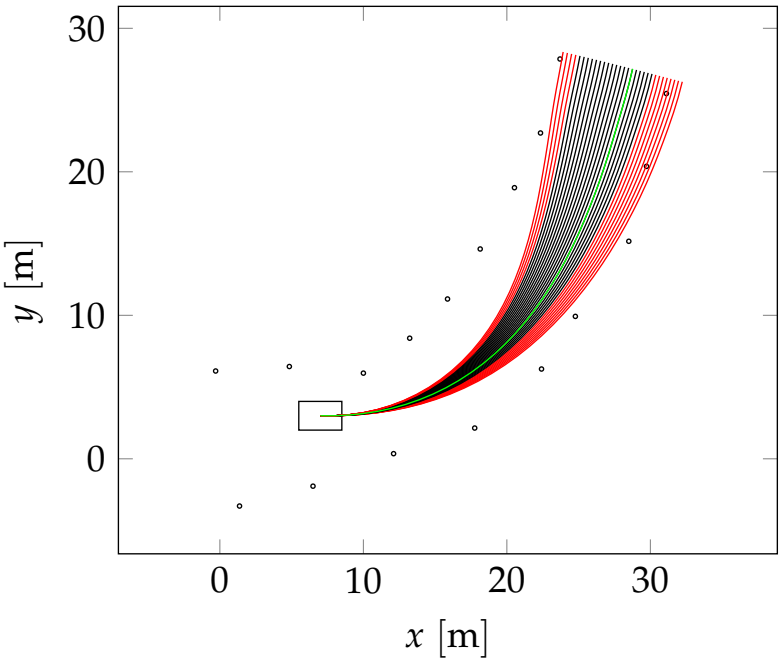


(b) $s_{\text{min}} = 20\,\text{m},\ k_v = 1,\ v = 5\,\text{m}\,\text{s}^{-1}$

Figure 4.12.: Effects of Δs_{min}. The minimum maneuver length Δs_{min} and speed gain k_v work in unison to determine the overall length of a maneuver. The impact of Δs_{min} on the length of a maneuver is fixed.



(a) $s_{\min} = 10\,\mathrm{m},\ k_v = 1,\ v = 5\,\mathrm{m\,s^{-1}}$



(b) $s_{\min} = 10\,\mathrm{m},\ k_v = 5,\ v = 5\,\mathrm{m\,s^{-1}}$

Figure 4.13.: Effects of the speed gain k_v. k_v determines the additional length of a maneuver about the current velocity v of the vehicle. This takes the dynamic of the vehicle into account, since lower velocities allow to drive less smoother path than one can drive with higher velocities.

4.5. Failures

During testing several scenarios occurred in which the proposed planner did diverge or was not able to produce a path to the goal section at all. In some scenarios, the planner produced a false positive result. The simulation provided served as a tool to analyze and identify those scenarios and the cause for the failure of the planning algorithm.

Divergence may occur for a poorly chosen objective function, i.e., one that only considers safety. After a tight curve, the planner may choose a path that passes through a gate at the side of the road and leaves the track. This is reasonable for the given objective function, since this path may maximize the distance to obstacles. Once the vehicle is outside of the track, the planner diverges and eventually fails.

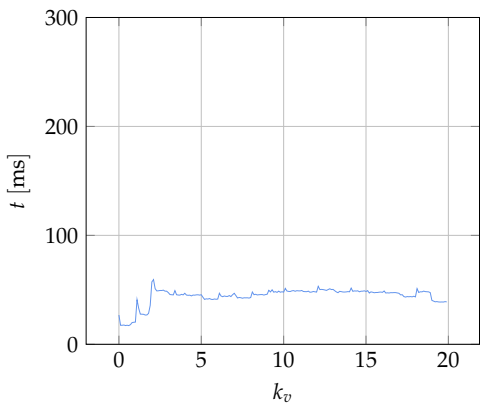
An objective function that only considers safety will diverge, since the path that maximizes the distance to obstacles is the one on the far outer side. After a few iterations, the planner will produce no more feasible paths, since the final offset q_f of all paths is bigger or equal to the radius of curvature, and the paths that lead back to the track exceed the maximum turning radius of the vehicle. The planner cannot recover from this error and fails with an exception. A false positive result may be produced, if the planner leaves the track and reaches the end of the baseframe before total failure occurs. It this case the planner will return a result, but the end of the path will be out of the boundaries of the track which is undesired behavior. Figure 4.15 illustrates such a scenario for a poorly chosen objective function and racetrack that allows the car to leave the track.

This behavior can be avoided by designing an objective function that considers the consistency of the path. The consistency in path choice disfavors paths that rapidly change direction and therefore minimizes the risk of leaving the track.

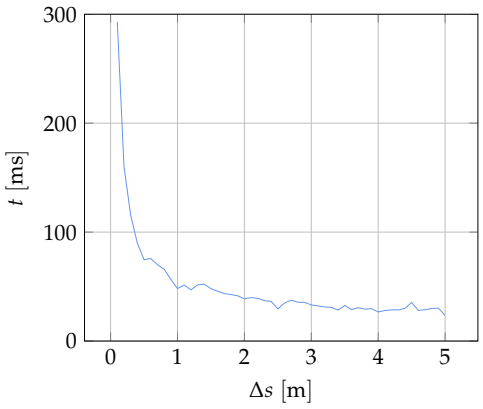
4.6. Discussion

The presented results indicate that the implemented planner could perform reliably in a racing-scenario like the FSD. However, all presented results were achieved in simulation and a controlled environment, without considerations of errors introduced by imperfect sensors or control algorithms. Therefore the results have to be interpreted with a caution until a full integration test of the planner on the target hardware can be achieved by future work. In past work [Fre14; CLS12] it was possible to perform integration tests and successfully experimented with an application of a comparable planner in an urban-race scenario. An application on the hardware of the REV-race is expected to be successful.

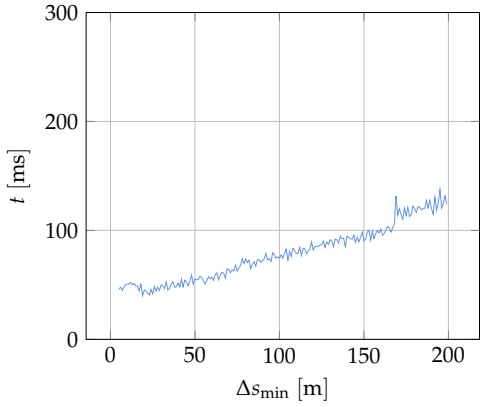
This work assumes that the rules of the FSG can be adapted or extended so that a map of the racetrack is provided to the planner. In a real racing-scenario the rules are fixed, and therefore it is unlikely that a map is known beforehand. Such a map may be recorded during an exploration drive in which SLAM is performed. This exploration phase is not implemented and is not subject of this work. Therefore no reliable statement can be made, if the planner would work in a real-world FSD-race.



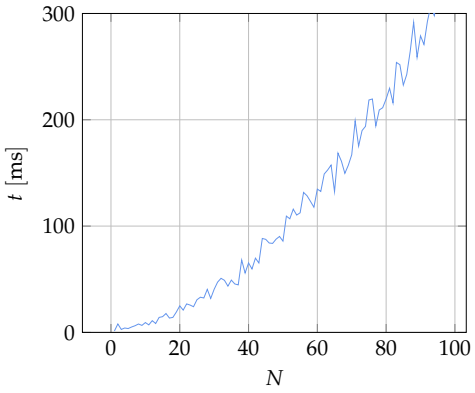
(a) Benchmark for impact of k_v



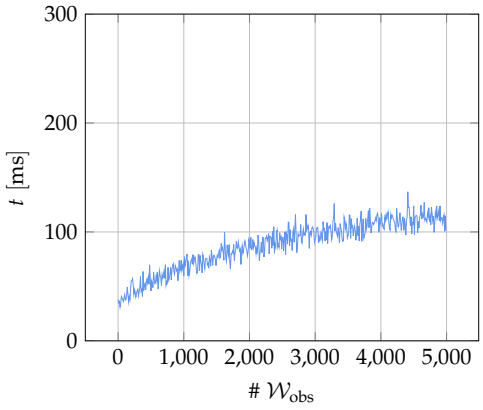
(b) Benchmark for impact of Δs



(c) Benchmark for impact of Δs_{min}

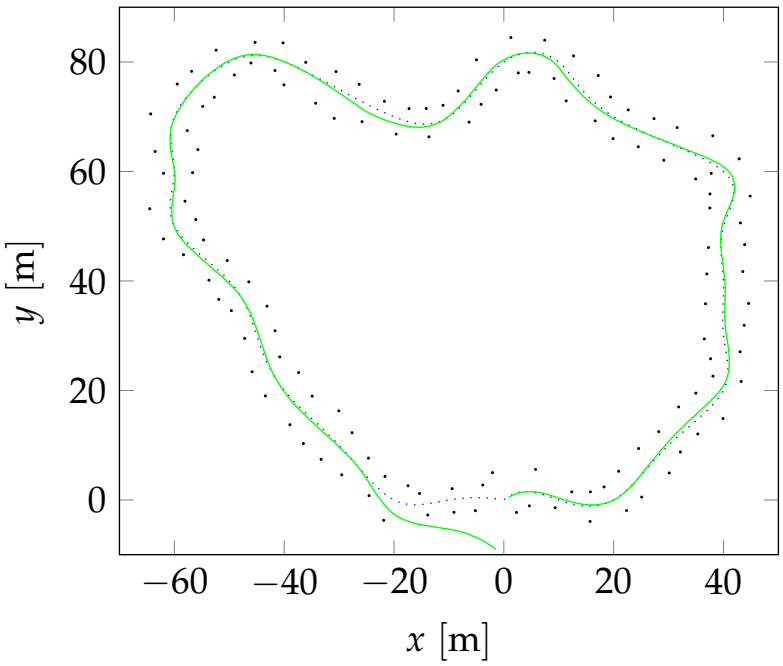


(d) Benchmark for impact of N

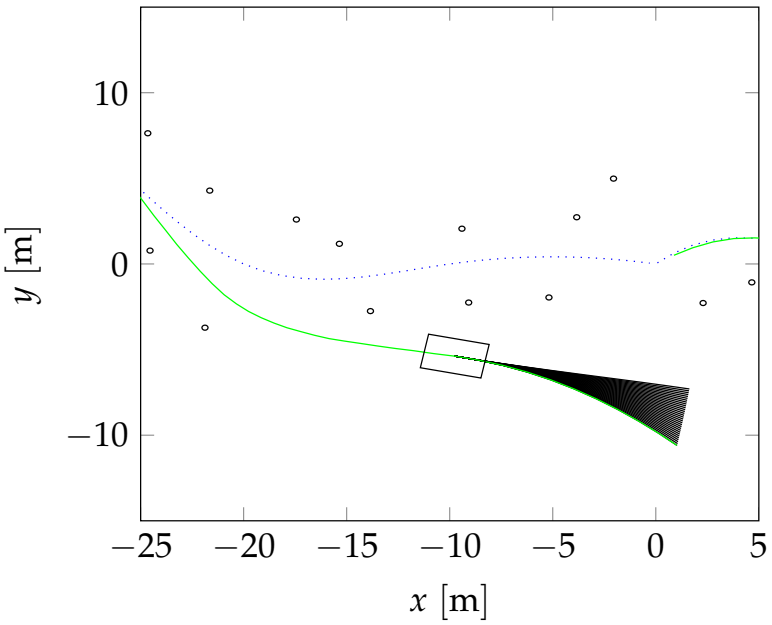


(e) Benchmark for impact of $\#\mathcal{W}_{\text{obs}}$

Figure 4.14.: Benchmarks for variations of design parameters. The benchmarks were calculated as the statistical mean of 100 planning iterations. Each benchmark was calculated under similar conditions, with only the parameter of interest alternated. The calculations where performed on a 2.6 GHz Intel Core i5 Processor.



(a) False positive solution path



(b) Divergence of vehilce after leaving the track

Figure 4.15.: For a poorly chosen objective function a planning failure or false positives might occur. This illustration shows an example of a planer with $w_s = 1, w_c = 0, w_\kappa = 0$. Since this planner will try to maximize the distance to obstacles, it will always choose the far outer path after leaving the track. If the planer reaches the end of the track before leaving the radius of curvature, the planner marks the solution as valid and produces a false positive result.

A recently published article [Lau17b] reports of an approach to the problem of finding a planning algorithm suitable for the FSD. This article suggests that an approach different to the one used in this work might be more suitable for the task at hand: A randomized and sampling based planner in combination with a machine learning technique to pick the best path. While this article contains no detailed technical background, the subject of the article is the algorithm used by the team that won the first FSD. This suggests that their solution might be reliable.

In the past years, RRTs have been highly successful for path planning in an unknown environment [Kuw+09; Kuw+08; Thr+06]. RRTs can perform randomized sampling-based planning in an uncertain environment without the need for a global map. Compared to the planner used in this work, RRTs are computational more intensive and may not be operated at such high frequencies as the proposed planner. This may require relatively slow velocities in a race-like scenario. Modern machine-learning techniques in combination with specialized hardware may provide a high-performance solution in unknown environments. While such planners have been tested in urban environments, no application in race-like scenarios is known. A comparison between the planer used in this work and end-to-end machine learning based controllers as used in [Boj+17; Boj+16] is therefore only partially possible until future work examines the use of machine-learning based planners in race environments.

While the provided implementation has been developed with the FSG and autonomous racing in mind, the used planning algorithm may also be suitable for other domains. In the past, comparable planners have been applied in urban and highway driving [Li+16; Wer+10]

5. Conclusion

This chapter concludes this thesis. Possible future work is discussed and the central parts and accomplishments of this work are summarized and recapitulated.

5.1. Future Work

This work was undertaken with the aim to provide a solid research platform for future students. Therefore, the algorithm and implementation used in this work leave room for improvements: The provided planner does not consider velocity profiles for the planned path. An additional velocity profile for the generated path would allow planning a dynamic racing-trajectory. The kinematic model described in section 2.6 can be extended and improved to allow a more dynamic driving style, i.e., by taking wheel slip [VT05] and chassis inertia into account [RNH12]. The Newton-Raphson minimization method described in section 3.5 is highly dependent on the initial guess and search-boundaries to be successful. The algorithm tends to diverge or get stuck in local minima for poor initial guesses. This problem could be solved by replacing Newton-Raphson with a numerically stable and fast converging algorithm [Jin+08].

Additional optimization steps during the generation of the baseframe may add additional racing performance: Conjugate gradient optimization could be used to smooth the curvature of the baseframe, while still allowing collision-free driving [Li+16]. Machine learning techniques may be used to tune the design parameters of the planner to allow optimal performance.

This work raised reasonable doubt that the provided algorithm would perform in an autonomous race without a map provided beforehand. Future work may extend the planner with an additional *exploration*-phase. In this phase, another sampling based algorithm, like RRTs would find a path through the racetrack. During the (expectedly slow) first exploration drive SLAM would be performed. After the drive, the provided planner is used for high-velocity driving.

The implementation of the provided planner was tested in simulation. While the results of the simulation appear promising, a full integration test has to be performed to obtain information of its performance on the target hardware and in a real-world scenario. This requires to implement a reliable and extensible software framework for the vehicle with modules for environment sensing, dead reckoning and feedback control. Former work from members of the REV-Team implemented a software which provides such functionality [Dra13], however it was found that integration in this software framework is a non trivial task, due to the lack of documentation, extensible interfaces

and unified communication protocols between modules. It is therefore recommended to rework parts of the software of the REV-vehicle to enable the integration of the provided planner. Porting to an unified framework like Robot Operating System (ROS)[Qui+09] could be a valuable approach to this problem. ROS provides rationales for software design, manages communication between the modules and provides implementations for common algorithms and data types used in robotics. The author strongly recommends to consider the use of ROS for the software of the REV-vehicle, since it not only enables an easy integration of the provided path planner, but solutions to problems in software design and communication.

5.2. Summary

This work aimed to solve the path planning problem in the context of autonomous racing. The Formula Student Driverless served as an example scenario for a possible rule set of a race. An iterative sampling-based path planning algorithm was used to attack the problem. An implementation of the proposed algorithm in the object-oriented programming language with the ISO standard of 2014 was introduced and discussed. The implemented algorithm was tested and benchmarked in the simulation.

It was found that the implementation finds a near-optimal solution to the path-planning problem for the simulated scenario. Performance measures suggest that the planner can be operated with 20 Hz. This enables high-dynamic driving with up-to $20\,\mathrm{m\,s^{-1}}$. This work uses a novel collision-checking algorithm that drastically reduces the impact of collision checking on the run-time. It was shown that the implemented algorithm allows efficient collision checking in workspaces that are densely populated with obstacles while retaining close to real-time performance.

This work provides a fully-documented open-source implementation of the discussed planning algorithm (see appendix C) that provides a base for further research on the topic. A full integration test on a race-car that conforms with the rules of the Formula-SAE in a scenario similar to the one described in the rules of the FSD is likely to happen in the near future.

A. Acronyms

AABB axis aligned bounding box

AGQ Adaptive Gaussian Quadrature

AMZ Akademischer Motorsportverein Zürich (Academic Motorsport Society Zurich)

ATQ Adaptive Trapezoidal Quadrature

BLAS basic linear algebra subroutine

FSD Formula Student Driverless

FSG Formula Student Germany

GPS global positioning system

GPU graphics processing unit

HMI human machine interface

HIL hardware in the loop

IMU inertial measurement unit

IVP initial value problem

Lidar light detection and ranging

OBB oriented bounding box

ODE ordinary differential equation

PDF probabilistic density function

PEAS Performance, Environment, Actuators, Sensors

PRM Probabilistic Roadmap

PMP Pontryagin's maximum (or minimum) principle

ROS Robot Operating System

RRT Rapidly Expanding Random Tree

REV Renewable Energy Vehicle

SAE Society of Automotive Engineers

SI International System of Units

SIL software in the loop

SLAM simultaneous localisation and mapping

STL standard template library

sRT-LUT static Runtime Lookup Table

UWA University of Western Australia

UML Unified Modeling Language

B. Nomenclature and Notation

This section introduces the mathematical notation and nomenclature that are used within this work. Key concepts of robotics and motion planning are very briefly introduced.

Vectors Vectors are denoted as bold lowercase letters of the Latin or Greek alphabet, i.e. $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ or $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$. Letters from the Latin or Greek alphabet or numerical values can also serve as indices for vector elements.

$$\boldsymbol{a} = \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix}$$

Unless explicitly noted otherwise, all vectors in this work are column vectors. Row vectors are denoted as transpose vectors i.e. $\boldsymbol{a}^T, \boldsymbol{b}^T, \boldsymbol{c}^T$ or $\boldsymbol{\alpha}^T, \boldsymbol{\beta}^T, \boldsymbol{\gamma}^T$.

$$\boldsymbol{a}^T = \begin{bmatrix} a_1 & \cdots & a_n \end{bmatrix}$$

Matrices Vectors are denoted as uppercase letters of the Latin or Greek alphabet, i.e. $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ or $\boldsymbol{\Gamma}, \boldsymbol{\Delta}, \boldsymbol{\Sigma}$. Letters from the Latin or Greek alphabet or numerical values can also serve as indices for matrix elements.

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}$$

Transposed matrices are denoted similar to vectors, i.e. $\boldsymbol{A}^T, \boldsymbol{B}^T, \boldsymbol{C}^T$ or $\boldsymbol{\Gamma}^T, \boldsymbol{\Delta}^T, \boldsymbol{\Sigma}^T$.

$$\boldsymbol{A}^T = \begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1n} & \cdots & a_{nn} \end{bmatrix}$$

The inverse of a Matrix is denoted as $\boldsymbol{A}^{-1}, \boldsymbol{B}^{-1}, \boldsymbol{C}^{-1}$ or $\boldsymbol{\Gamma}^{-1}, \boldsymbol{\Delta}^{-1}, \boldsymbol{\Sigma}^{-1}$

$$\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}$$

where $\boldsymbol{I}$ is the identity matrix.

Functions A function is denoted by any letter of the Greek or Latin alphabet. If not otherwise stated the following notation is used: Lowercase letters indicate functions that map onto a domain of scalars. Uppercase letters indicate functions that map onto a domain of sets. Bold lowercase letters indicate functions that map on a domain of vectors. Bold uppercase letters indicate functions that map on a domain of matrices. A function f is declared by stating its domain X and co-domain Y using the expression:

$$f(x) : X \to Y$$

Sets Sets are denoted as uppercase, calligraphic letters i.e. $\mathcal{A}, \mathcal{B}, \mathcal{C}$.

Units All units in this work are given according the International System of Units (SI). For example, this means that the time t is given in seconds [s] and distances are given in meters [m].

Robotics In this work the following notation is used for key-concepts of robotics. For a in-depth description of those concepts the interested reader is referred to [Lav06] and [Ham57].

$\mathcal{W}$ Work Space $\mathcal{W} \subset \mathbb{R}^2$ and $\mathcal{W} = \mathcal{W}_{\text{obs}}\mathcal{W}_{\text{free}}$.

$\mathcal{W}_{\text{obs}}$ Obstacle in workspace, set of points that can not intersect with the boundaries of the vehicle.

$\mathcal{W}_{\text{free}}$ Free-Work Space. $\mathcal{W}_{\text{free}} = \mathcal{W} \setminus \bigcup_i \mathcal{W}_{\text{obs}i}$ for $i = 1, \ldots, N$ where $\mathcal{W}_{\text{obs},i}$ is the i-th obstacle in the workspace and N is the absolute numer of obstacles in the workspace.

$\boldsymbol{x}$ Configuration of the vehicle.

$\mathcal{A}(x)$ set of points in $\mathcal{W}$ occupied by the vehicle at configuration $\boldsymbol{x} \in \mathbb{R}^n$. $\mathcal{A}(x) \subset \mathcal{W}$.

$\mathcal{C}$ Configuration Space. Set of all possible configurations of the robot $\mathcal{C} \subset \mathbb{R}^n$ and $\mathcal{C} = \mathcal{C}_{\text{obs}}\mathcal{C}_{\text{free}}$.

$\mathcal{C}_{\text{obs}}$ Obstacle in configuration space $\mathcal{C}_{\text{obs},i} = \{\boldsymbol{x} | \mathcal{A}(\boldsymbol{x}) \bigcap_i \mathcal{W}_{\text{obs},i} \neq \emptyset\}$.

$\mathcal{C}_{\text{free}}$ Free-Configuration Space $\mathcal{C} \setminus \cup_i \mathcal{C}_{\text{obs},i}$ where $\mathcal{CO}_i$ is the i-th obstacle in the configuration space.

Source Code This work contains source code listing and online source code. Inline source code or names of programs or software libraries are given `in this fond`. The presented source code listings in this work are often simplified or adapted to illustrate specific implementation features. Such adjustments are indicated by source code comments (i.e. `//... here is something missing`), which describe what kind of adjustments have been undertaken.

C. Source Code

This section describes how to access the source-codes and documentation of the implementation, compile and run the example programs. The instructions are provided for Debian based operation system with APT installed. Instructions which are intended to be executed in a terminal start with a `$`, i.e:

```
$ echo I am an instruction which the user should run in a terminal
```

C.1. Accessing the Sources

The sources can be accessed using the source code management system `git` or from the medium attached to this document. To access the sources using the source code management system, first one needs to install git on their system, ether by downloading and running the installer from https://git-scm.com/ or by executing

```
$ sudo apt-get install git
```

Then, to download the sources, in a terminal run:

```
$ git clone https://github.com/RomanCPodolski/samling_based_planning.git
```

Accessing the sources through git is recommended.

C.2. Installing Dependencies

The provided software depends on a handful of third-party software libraries which need to be installed in order to build the sources.

Boost Instructions how to install boost are provided online[1]. Please note that at least version `1.65.0` is needed. While boost can also be installed via APT, this is discouraged since the packages are generally outdated and will not contain the required version.

Gflags Download the sources using

```
$ git clone https://github.com/gflags/gflags.git
```

Build and install the software using the steps described in the `INSTALL` file. Full install instructions for gflags are provided in its documentation[2].

[1] http://www.boost.org/doc/libs/1_65_1/more/getting_started/unix-variants.html
[2] https://gflags.github.io/gflags/download

Glog The installation process is similar to the steps described in appendix C.2. Download the sources using

```
$ git clone https://github.com/google/glog.git
```

Then follow the install instructions in the `INSTALL` file.

Python Almost any modern operating system has an executable version of python installed. To check if python is already installed run

```
$ python -v
```

If this command does not return a version number, download an installer from https://www.python.org/downloads/ for your operating system and follow the instructions.

C.3. Compiling the Sources

The project uses `waf`[3] to build the sources. This is a python based tool and doesn't need additional installation steps.

First make sure you have all additional software installed (see appendix C.2). Then, in a terminal, change the directory to the project root.

```
$ cd sampling_based_planning
```

The project root should (next to the sources) contain a file called `waf`. This is the executable of the build system.

To build the project run

```
$ alias waf=python $PWD/waf
$ waf configure build
```

This will check if all necessary software is installed and build the project. The executables can be found in a folder called `build/src`.

C.4. Accessing the Documentation

The documentation can be accessed online[4] or locally. To access the documentation locally follow the steps described in appendix C.3 and additionally run:

```
$ waf build -d
```

This will build the documentation in the folder `build/doc` as html and latex files.

[3] https://waf.io
[4] https://romancpodolski.github.io/samling_based_planning/

List of Figures

List of Tables

Bibliography

[Bar+02] T. Barfoot, C. Clark, S. Rock, and G. D'Eleuterio. "Kinematic Path-planning for Formations of Mobile Robots with a Nonholonomic Constraint." In: *IEEE/RSJ International Conference on Intelligent Robots and System*. Vol. 3. Lausanne, Switzerland, 2002, pp. 2819–2824. ISBN: 0-7803-7398-7. DOI: 10.1109/IRDS.2002.1041697. URL: http://ieeexplore.ieee.org/document/1041697/.

[Boj+16] M. Bojarski, D. Del Testa, D. Dworakowski, et al. *End to End Learning for Self-Driving Cars*. Tech. rep. 2016, pp. 1–9. arXiv: 1604.07316. URL: http://arxiv.org/abs/1604.07316.

[Boj+17] M. Bojarski, P. Yeres, A. Choromanska, et al. *Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car*. Tech. rep. 2017, pp. 1–8. DOI: 10.1109/ICRA.2016.7487486. arXiv: 1704.07911. URL: http://arxiv.org/abs/1704.07911.

[Chr+08] Chris Urmson, J. Anhalt, D. Bagnell, et al. "Autonomous Driving in Urban Environments: Boss and the Urban Challenge." In: *J. Field Robotics* 25.June (2008), pp. 245–267. ISSN: 14746670. DOI: 10.1002/rob.20255. arXiv: 10.1.1.91.5767.

[Chu15] T. Churack. *Improved Road-Edge Detection And Path-Planning for an Autonomous SAE Electric Race Car*. Tech. rep. October. Perth: Universtity of Western Australia, 2015.

[CLS12] K. Chu, M. Lee, and M. Sunwoo. "Local path planning for off-road autonomous driving with avoidance of static obstacles." In: *IEEE Transactions on Intelligent Transportation Systems* 13.4 (2012), pp. 1599–1616. ISSN: 15249050. DOI: 10.1109/TITS.2012.2198214.

[Cou92] R. C. Coulter. *Implementation of the Pure Pursuit Path Tracking Algorithm*. Tech. rep. January. 1992. DOI: CMU-RI-TR-92-01.

[CS14] S. Chacon and B. Straub. *Pro Git*. Apress, 2014. URL: https://git-scm.com/book/en/v2.

[DMC90] E. D. Dickmanns, B. Mysliwetz, and T. Christians. "An Integrated Spatio-Temporal Approach to Automatic Visual Guidance of Autonomous Vehicles." In: *IEEE Transactions on Systems, Man and Cybernetics* 20.6 (1990), pp. 1273–1284. ISSN: 21682909. DOI: 10.1109/21.61200.

[Dra13] T. H. Drage. *Development of a Navigation Control System for an Autonomous Formula SAE-Electric Race Car*. Tech. rep. November. University of Western Australie, 2013, p. 86. URL: http://robotics.ee.uwa.edu.au/theses/2013-REV-Navigation-Drage.pdf.

[Fis+16] D. L. Fisher, M. Lohrenz, D. Moore, et al. "Humans and Intelligent Vehicles: The Hope, the Help and the Harm." In: *IEEE Transactions on Intelligent Vehicles* PP.99 (2016), pp. 1–1. ISSN: 2379-8904. DOI: 10.1109/TIV.2016.2555626. arXiv: 2379-8858. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=%7B%5C&%7Darnumber=7467508%7B%5C&%7Durl=http://ieeexplore.ieee.org/xpls/abs%7B%5C_%7Dall.jsp?arnumber=7467508%7B%5C%%7D5Cnhttp://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7467508.

[For17] Formula Student Germany. *DT FSG Driverless Vehicle (DV) Technical Specifications 2017*. 2017.

[Fre14] M. French. *Advanced Path Planning for an Autonomous SAE Electric Race Car*. Tech. rep. October. 2014.

[FS 17] FS Germany. *FSG Rules 2017*. 2017. URL: https://www.formulastudent.de/uploads/media/Rules2017%7B%5C_%7DV1.0.pdf.

[GJa10] G. Guennebaud, B. Jacorb, and E. al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org.

[God17] M. Godbolt. "What Has My Compiler Done for Me Lately? Unbolting the Compiler's Lid." In: *CppCon*. 2017. URL: https://github.com/CppCon/CppCon2017/blob/master/Keynotes/What%20Has%20My%20Compiler%20Done%20for%20Me%20Lately%20-%20Unbolting%20the%20Compiler's%20Lid/What%20Has%20My%20Compiler%20Done%20for%20Me%20Lately%20-%20Unbolting%20the%20Compiler's%20Lid%20-%20Matt%20Godbolt%20-%20CppCon%202017.pdf.

[GP90] B. Guenter and R. Parent. "Computing the Arc Length of Parametric Curves." In: *IEEE Computer Graphics and Applications* 10.3 (1990), pp. 72–78. ISSN: 02721716. DOI: 10.1109/38.55155.

[Gut84] A. Guttman. "R-trees: A Dynamic Index Structure for Spatial Searching." In: *ACM SIGMOD Record* 14.2 (1984), p. 47. ISSN: 01635808. DOI: 10.1145/971697.602266. arXiv: ISBN0-89791-128-8. URL: http://portal.acm.org/citation.cfm?doid=971697.602266.

[Ham57] K. a. Hamilton. *Principles of Robot Motion*. Vol. 53. 1957, pp. 1079–1083. ISBN: 9780262033275. DOI: 10.1017/S0263574706212803. URL: http://books.google.de/books/about/Principles%7B%5C_%7Dof%7B%5C_%7DRobot%7B%5C_%7DMotion.html?id=S3biKR21i-QC%7B%5C&%7Dredir%7B%5C_%7Desc=y.

[HS09] O. Holtz and N. Shomron. "Computational Complexity and numerical stability of linear problems." In: *European Congress of Mathematics Amsterdam, 14–18 July, 2008*. 2009, pp. 381–400. DOI: 10.4171/077-1/16. arXiv: arXiv:0906.0687v2. URL: http://www.ems-ph.org/doi/10.4171/077-1/16.

[ISO14] ISO/IEC. *ISO International Standard ISO/IEC 14882:2014(E) – Programming Language C++. [Working draft]*. Geneva, Switzerland, 2014.

[Jin+08] X. Jinting, L. Weijun, B. Hongyou, and L. Lun. "Accurate and Efficient Algorithm for the Closest Point on a Parametric Curve." In: *Proceedings - International Conference on Computer Science and Software Engineering, CSSE 2008* 2 (2008), pp. 1000–1002. DOI: 10.1109/CSSE.2008.618.

[Kuw+08] Y. Kuwata, G. A. Fiore, J. Teo, et al. "Motion Planning for Urban Driving using RRT." In: *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS* (2008), pp. 1681–1686. DOI: 10.1109/IROS.2008.4651075.

[Kuw+09] Y. Kuwata, J. Teo, G. Fiore, et al. "Real-Time Motion Planning with Applications to Autonomous Urban Driving." In: *IEEE Transactions on Control Systems Technology* 17.5 (2009), pp. 1105–1118. ISSN: 10636536. DOI: 10.1109/TCST.2008.2012116.

[Lau15] L. Laursen. *Robot Cars To Join Formula E Racing*. 2015. URL: https://spectrum.ieee.org/cars-that-think/transportation/advanced-cars/robot-cars-to-join-formula-e-series.

[Lau17a] L. Laursen. *Students Race Driverless Cars in Germany in Formula Student Competition*. 2017. URL: https://spectrum.ieee.org/cars-that-think/transportation/self-driving/students-race-driverless-cars-in-germany-in-formula-student-competition (visited on 11/02/2017).

[Lau17b] L. Laursen. *The Tech That Won the First Formula Student Driverless Race*. 2017. URL: https://spectrum.ieee.org/cars-that-think/transportation/self-driving/the-tech-that-won-the-first-formula-student-driverless-race (visited on 11/02/2017).

[Lav06] S. M. Lavalle. *Planning Algorithms*. 2006, p. 842. ISBN: 9780511546877. DOI: 10.1017/CBO9780511546877. arXiv: arXiv:1011.1669v3. URL: http://planning.cs.uiuc.edu/booka4.pdf.

[Li+16] X. Li, Z. Sun, D. Cao, et al. "Real-Time Trajectory Planning for Autonomous Urban Driving: Framework, Algorithms, and Verifications." In: *IEEE/ASME Transactions on Mechatronics* 21.2 (2016), pp. 740–753. ISSN: 10834435. DOI: 10.1109/TMECH.2015.2493980.

[MS93] R. M. Murray and S. S. Sastry. "Nonholonomic Motion Planning: Steering Using Sinusoids." In: *IEEE Transactions on Automatic Control* 38.5 (1993), pp. 700–716. ISSN: 00189286. DOI: 10.1109/9.277235.

[Pad+16] B. Paden, M. Cap, S. Z. Yong, et al. "A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles." In: *IEEE Transactions on Intelligent Vehicles* 1.1 (2016), pp. 33–55. ISSN: 2379-8904. DOI: 10.1109/TIV.2016.2578706. arXiv: 1604.07446. URL: http://arxiv.org/abs/1604.07446.

[Pom89] D. a. Pomerleau. "Alvinn: An Autonomous Land Vehicle in a Neural Network." In: *Advances in Neural Information Processing Systems 1* (1989), pp. 305–313.

[Qui+09] M. Quigley, K. Conley, B. Gerkey, et al. "ROS: an open-source Robot Operating System." In: *Icra* 3.Figure 1 (2009), p. 5. ISSN: 0165-022X. DOI: http://www.willowgarage.com/papers/ros-open-source-robot-operating-system. arXiv: 1106.4561. URL: http://pub1.willowgarage.com/%7B~%7Dkonolige/cs225B/docs/quigley-icra2009-ros.pdf.

[RN09] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 3rd edition*. 2009, p. 1152. ISBN: 9780136042594. DOI: 10.1017/S0269888900007724. arXiv: 9809069v1 [arXiv:gr-qc].

[RNH12] A. Rucco, G. Notarstefano, and J. Hauser. "Computing minimum lap-time trajectories for a single-track car with load transfer." In: *Proceedings of the IEEE Conference on Decision and Control* (2012), pp. 6321–6326. ISSN: 01912216. DOI: 10.1109/CDC.2012.6426265. arXiv: 1204.3968.

[SA04] H. Sutter and A. Alexandrescu. *C++ coding standards : 101 rules, guidelines, and best practices*. 2004. DOI: 10.1017/CBO9781139083683.

[SAE17] SAE International. *2017-18 Formula SAE Rules Table of Contents*. 2017. URL: http://www.fsaeonline.com/content/2017-18%20FSAE%20Rules%20PRELIMINARY.pdf.

[Sch17] B. Schäling. *The Boost C++ Libraries*. 2017. URL: https://theboostcpplibraries.com/ (visited on 08/01/2017).

[SFP17] A. Shvets, G. Frey, and M. Pavlova. *Software Development Anti-Patterns - The Blob*. 2017. URL: https://sourcemaking.com/antipatterns/the-blob (visited on 09/18/2017).

[Str13] B. Stroustrup. *the C++ Programming Language 4Th Edition*. 2013, pp. 3–36. ISBN: 9780321563842. DOI: DOI-Test.

[Thr+06] S. Thrun, V. Pratt, P. Stang, et al. "Stanley: The Robot that Won the DARPA Grand Challenge." In: *J. Field Robotics* 23.June (2006), pp. 245–267. ISSN: 14746670. DOI: 10.1002/rob. arXiv: 10.1.1.91.5767.

[Thr02] S. Thrun. "Probabilistic robotics." In: *Communications of the ACM* 45.3 (2002), pp. 1999–2000. ISSN: 00010782. DOI: 10.1145/504729.504754. arXiv: arXiv:1011.1669v3. URL: http://portal.acm.org/citation.cfm?doid=504729.504754.

[TW14] L. N. Trefethen and J. A. C. Weideman. "Trapezoidal Rule." In: 56.3 (2014), pp. 1–2. DOI: 10.1002/0471667196.ess2752.

[VT05] E. Velenis and P. Tsiotras. "Minimum time vs maximum exit velocity path optimization during cornering." In: *IEEE International Symposium on Industrial Electronics* I (2005), pp. 355–360. DOI: 10.1109/ISIE.2005.1528936.

[Wal+11] J. Walter, M. Koch, G. Winkler, and D. Bellot. *Boost uBLAS*. 2011. URL: http://www.boost.org/doc/libs/1%7B%5C_%7D65%7B%5C_%7D1/libs/numeric/ublas/doc/ (visited on 11/04/2017).

[Wer+10] M. Werling, J. Ziegler, S. Kammel, and S. Thrun. "Optimal trajectory generation for dynamic street scenarios in a frenét frame." In: *Proceedings - IEEE International Conference on Robotics and Automation* (2010), pp. 987–993. ISSN: 10504729. DOI: 10.1109/ROBOT.2010.5509799.

[WKA02] H. Wang, J. Kearney, and K. Atkinson. "Robust and efficient computation of the closest point on a spline curve." In: *Proceedings of the 5th ...* (2002), pp. 397–405. ISSN: 0972848207. URL: http://homepage.cs.uiowa.edu/%7B~%7Dkearney/pubs/CurvesAndSufacesClosestPoint.pdf.

[WKA03] H. Wang, J. Kearney, and K. Atkinson. "Arc-Length Parameterized Spline Curves for Real-Time Simulation." In: *Curve and surface design: Saint-Malo 2002.* (2003), pp. 387–396.