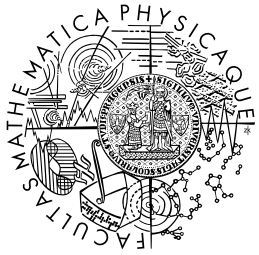


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Šimon Rozsíval

Trajectory planning for fast moving cars

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	Prof. RNDr. Roman Barták, Ph.D.
Study programme:	Computer Science
Study branch:	Artificial Intelligence

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Plánování dráhy pro rychle se pohybující automobily

Autor: Šimon Rozsíval

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: prof. RNDr. Roman Barták, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Cílem této práce je vytvořit umělého agenta pro autonomní závodní vozidlo. Tento projekt je inspirován soutěží F1/10. Agent používá plánovací algoritmus pro nalezení trajektorie minimalizující dobu cesty do cíle. Pro dosažení plánování v reálném čase provádí agent nejprve analýzu trati a plánuje svůj pohyb pouze pro dvě nejbližší zatáčky. Agent opětovně plánuje trajektorii několikrát za vteřinu zatímco se pohybuje podél závodního okruhu, aby kompenzoval nedokonalé sledování naplánované trajektorie. Úspěšně jsme agenta otestovali pomocí simulátoru Gazebo a dosáhli jsme uspokojivých výsledků. Algoritmus jsme testovali také na vlastním robotovi vybaveným palubním počítačem a senzory, ovšem jen částečně uspokojivými výsledky.

Klíčová slova: plánování, trasa, mobilní roboti

Title: Trajectory planning for fast moving cars

Author: Šimon Rozsíval

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Prof. RNDr. Roman Barták, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: The goal of this thesis is to create an artificial agent for an autonomous racing vehicle. This project is inspired by the F1/10 racing competition. The agent uses a planning algorithm to find a time-optimal trajectory. To achieve real-time performance, the agent analyzes the map of the track and it plans only for the next two corners immediately ahead of the vehicle. The agent re-plans several times per second as it drives along the circuit to account for imprecise trajectory following. We successfully tested the agent in the Gazebo simulator with good results. We also tested the algorithm on a custom car-like robot equipped with an on-board computer and sensors, but with limited success.

Keywords: planning, trajectory, mobile robots

Contents

Introduction

Autonomous vehicles are slowly but steadily making their way from research facilities to the public roads around the world and it seems inevitable that they will soon become part of our daily lives. Autonomous vehicles could have a significant impact on transportation of people and goods. For example, thousands of people die in car accidents every year because of human error [1]. Some believe that replacing humans with accurate electronic sensors and computer algorithms, which never make errors, could reduce the number of accidents on roads significantly. It seems that companies around the world see the potential in this technology and invest a lot of effort and money in research of self-driving cars. Even though nobody can guarantee that autonomous vehicles will never make errors and that they will solve other problems of road transportation we face today, the trend is clear. The key to the success of this technology will be reliable and robust algorithms which will work in all driving scenarios and weather conditions and which the public will trust.

The idea of self-driving cars is far from new. Prototypes of autonomous vehicles were demonstrated on public roads as early as the 1980s. Among some of the well-known projects from this era were for example the NavLab vehicle of the Carnegie Mellon University [2] or the project of Ernst Dickmanns in cooperation with the Daimler company [3]. The DARPA challenges, organized between 2004 and 2007, gained a lot of media attention. We can see that self-driving cars are no longer just science fiction. Since then, many driving assistants such as adaptive cruise control, lane keeping assistants, parallel parking assistants, collision warning, and emergency braking assistants started appearing in commercially available vehicles. Some systems, like the Tesla Autopilot, go even further and allow complex autonomous maneuvers such as overtaking other vehicles.

These commercially available assistants provide the user only with partial autonomy. The driver is required to pay attention to the road all the time and he or she is required to be prepared to take control of the vehicle if the system fails or if it cannot handle the traffic situation. Fully autonomous vehicles will not require any human interaction. They might even lack any means of manual driving. The computer will be responsible for the vehicle from the start to the destination irrespective of the weather or traffic. There is an ongoing research into full autonomy by some companies like Waymo and Uber and prototypes of vehicles equipped with this technology are already being tested on public roads.

One of the key problems in autonomous driving is the ability to find a good trajectory for the car in various different scenarios. This trajectory must be safe for the passengers of the vehicle and for other road users. It must also conform to road limitations such as speed limits and other traffic rules. Automated planning is an area of artificial intelligence which gives us means to formulate and solve this problem. For a well-defined planning problem which describes the physics of the vehicle and our goals, we can find a sequence of control inputs which will drive the vehicle to the goal as we need.

In this thesis, our focus is trajectory planning for fast moving cars. We will design and develop an autonomous vehicle for the task of autonomous racing. The racing scenario is simplification of real-world driving. It requires us to avoid

collisions and to find efficient trajectories while moving at high speeds and to adapt to unforeseen obstacles on the road. At the same time, this simplified problem allows us to ignore any traffic rules. We do not have to consider any speed limits, crossroads, stop signs, or other road users.

We will create an autonomous racing agent which utilizes a trajectory planning algorithm to find fast routes along the racing circuit. We will test different algorithms for searching solutions to the planning problem and compare them in a series of experiments on a real-world car model with sensors and an embedded on-board computer.

First, we introduce the autonomous driving problem and split it into several sub-problems. We give a brief overview of related works in the field of autonomous driving. We describe an architecture of an artificial agent for the task of autonomous racing.

Next, we examine the individual sub-problems we must solve in order to create the racing agent. We analyze and formulate the trajectory planning problem for a fast moving car. We describe adaptations of several planning algorithms for our specific problem. We study how the vehicle reacts to steering commands and we describe a model of the actuators and of the vehicle movement. We then describe algorithms for following the planned trajectory.

Finally, we implement the racing agent and we conduct several experiments to verify the performance and success rate of our algorithms and we test them on a custom-built real-world experimental vehicle.

1. Autonomous Driving

Autonomous driving is a complex task. The vehicle collects information about the surrounding environment from its sensors and then it decides its next control input to the actuators of the vehicle. There are two general approaches to this problem: end-to-end driving and decomposition into several subproblems.

The end-to-end driving approach takes the sensor data as an input and maps them to the control inputs. The end-to-end driving algorithm can be implemented for example using a stream of images from a front-facing camera and a neural network trained using supervised or reinforcement learning. It was successfully demonstrated for example in ALVINN, a simple 3-layer neutral network used in the NavLab autonomous vehicle of Carnegie Mellon University in 1989 [4], or more recently with a deep convolutional neural network by Nvidia [5]. End-to-end driving avoids explicit modeling of the world and the vehicle and it relies on the knowledge extracted from the training data or by learning by trial and error during reinforcement learning.

The other approach is to split the complex problem into several smaller problems, which are solved independently. We can split the problem into three main subproblems: Perception, Planning, and Control.

Perception is the process of collecting data from the sensors and processing them to obtain the current state of the world and the internal properties of the vehicle. The task of determining the position of the vehicle on a map is called localization. The sensors which could be used for localization are cameras, radars, ultrasonic sensors, and LIDARs. The data from these sensors can be used to determine distances to nearby obstacles in different directions. Based on the previous known position of the vehicle, the estimate of its movement, and the distances to obstacles in different directions, the position on a known map can be determined using an algorithm such as Adaptive Monte Carlo Localization (AMCL) [6] [7]. In certain scenarios, the relative distances to the obstacles might not be enough to determine correct position. This can happen for example when the vehicle is driving through a straight tunnel and the distances to the walls are constant even though the vehicle is moving. It is useful to combine readings from multiple sensors which provide odometry such as wheel encoders which measure how many times the wheels turn and an Inertial Measurement Unit (IMU) with gyroscopes and accelerometers which measures the linear and angular acceleration of the vehicle. The combination of data from multiple different types of sensors is called sensor fusion. Kalman filter is an example of an algorithm which is frequently used to fuse data from different sources [8].

The vehicle must also be able to read road signs and road surface markings for driving along public roads. The data from the sensors can also be used to detect obstacles and for obstacle tracking. The type of obstacle is then identified and in the case of dynamic obstacles such as other cars, bicyclists, pedestrians, animals, or moving inanimate objects their movement in time must be predicted to prevent collisions.

The geometry of a car-like vehicle limits its controllability. The vehicle cannot turn while it is stopped and it can only start turning once it is moving forwards or backwards. Usually the only way to control the turning radius is by turning

the front wheels and the rear wheels are fixed. In order to be able to make any reasoning about the future states of the vehicle, we must be able to predict the effect of a sequence of control inputs on the motion of the vehicle. Without an accurate model of the vehicle we could select a trajectory which cannot be safely followed, or which would be ineffective. We will discuss vehicle modelling in detail in chapter Vehicle Model.

With the knowledge of the vehicle model, we can search for a plan consisting of control inputs over some time period which will result in a collision-free trajectory through the environment which minimizes some cost function (e.g., the time it will take to reach some goal location). Planning several steps into the future can give us an advantage over a simple end-to-end driving approach. For example, in the case of autonomous racing, the agent can take advantage of the knowledge of the racing track map, and account for the shape of the track hidden around the next corner. We should be able to decide, when to slow down and when to accelerate, as well as at which point to start turning into a corner, in order to reach the best lap times. We call this subproblem trajectory planning.

Executing the sequence of control inputs one by one might cause the vehicle to drive off the track. The trajectory planning algorithm relies on a vehicle model which might not be accurate. The reference trajectory is therefore idealized, and it might not be possible to achieve it exactly by the actual vehicle. It might also lead to a collision with an obstacle on the track which was not known at the time of planning. A trajectory following algorithm chooses the next action based on the current state of the vehicle and the distance to the position, vehicle orientation, and speed at a corresponding point on the reference trajectory. The algorithm should also avoid any unexpected obstacles detected by the sensors.

The actual effects of the control inputs are measured by the sensors and the control algorithm tries to minimize the error between the planned trajectory and the actual trajectory of the vehicle in its next step. This process is referred to in control theory as closed feedback loop.

1.1 Related Work

DARPA Urban Challenge In 2007, the DARPA Urban Challenge took place in the USA. The challenge was successfully completed by several vehicles with different approaches to trajectory planning. The entry from the Carnegie Mellon University, Boss, won the competition. It uses the Anytime D* algorithm for navigation in unstructured environment [9]. The Stanford team used an algorithm called Hybrid A* in their vehicle called Junior [10]. This algorithm is an extension of the A* algorithm which discretizes the continuous search space into a discrete grid to avoid examining similar configurations multiple times, but it keeps the information about the trajectories in the continuous configuration space to produce smooth feasible trajectories. The team from MIT used modified RRT algorithm in their vehicle [11].

MIT RACECAR RACECAR (Rapid Autonomous Complex-Environment Competing Ackermann-steering Robot) is a project at MIT which uses an RC car to test various perception, planning, and control algorithms. The cars are used

by students in various courses at MIT[1].

AutoRally The Georgia Tech uses an RC-car based platform AutoRally to test algorithms for self-driving cars. The publications of algorithms tested on AutoRally include both classical planning-based approaches, such as MPC, and also deep learning end-to-end driving policies[2].

The F1/10 Competition This problem is inspired by the F1/10 competition organized by the University of Pennsylvania and the University of Virginia [12]. We will use the resources from this competition to build a similar autonomous vehicle based on an RC car. To evaluate the performance of the vehicle, we will use the criteria of Time Trial Race of F1/10. In the Time Trial Race, the vehicle drives for 5 minutes around a circuit trying to achieve as many laps as possible without a collision with the boundary of the track or with static obstacles on the track. The time of the fastest lap is also recorded.

CTU F1/10 Research A team at the Czech Technical University in Prague called "Řeřicha" successfully entered the F1/10 competition several times. In the April of 2018, the team won the competition in Porto, Portugal. One of the students, Martin Vajnar, described the construction of the vehicle in his Master Thesis [13] and the source code of the algorithm is publicly available on GitHub[3].

The team succeeded with a purely reactive algorithm called "Follow The Gap" which processes the raw data from a LIDAR and for each reading, it selects a steering angle in the direction where there is no obstacle (in the direction of the biggest "gap") in the laser scan data and it adjusts the speed based on the steering angle (when the car drives mostly straight, the speed is high, when the car turns, the speed is lower). Later in the same year, the CTU team finished third in Torino, Italy.

Jan Dusil describes an updated hardware of the vehicle of the CTU team and he describes algorithms for detecting the slip angle of the vehicle in his Bachelor Thesis [14]. Jaroslav Klapálek describes an algorithm for dynamic obstacle avoidance for F1/10 car where he solves the problem of distributed intersection control in his Master Thesis [15].

Roborace is a competition of full-scale autonomous cars. All contestants use the same hardware, a specially designed racing vehicle called *Robocar*. The first race which was held in 2019 was won by a team from Technical University of Munich. This team published several papers about the algorithms and methods they used, including the trajectory planning algorithm [16], an extension of the AMCL ROS package [17], and state estimation method [18].

[1] `https://racecar.mit.edu/education`

[2] `https://autorally.github.io/`

[3] `https://github.com/f1tenth/F110CPSWeek2018/tree/master/CTU`

2. Artificial Racing Agent

In this chapter we will first analyze how human racing drivers approach the task of car racing. We will then use this information to design the behavior of an autonomous agent and break the problem down into several smaller tasks. We will discuss how we can analyze the racing track and prepare it for trajectory planning.

2.1 Racing Line

When a human driver drives on a racing circuit, his or her task is to complete several laps around the circuit in the shortest possible time. In order to minimize the lap times, the driver trades off the total distance the car travels for the average speed at which it moves. The trajectory of the vehicle through the racing track is sometimes referred to as the racing line and it depends on the shape of the track and on the vehicle. The driver will follow different racing lines for different vehicles on the same circuit based on many of their different properties such as maximum speed, acceleration, braking power, or the grip of tires.

For a bend with a constant curvature there is a maximum speed at which a specific car can go and stay on the track. Exceeding this speed will cause a loss of friction between the tires and the road surface and the tires will not be able to exert enough lateral force to keep the car on the curve and the car will start to travel along a curve with a greater radius than the one intended by the driver. This situation is called understeer or oversteer, depending on whether it is the front or rear wheels which lose the friction and which wheels are powered by the engine.

On the other hand, for a constant vehicle speed, the vehicle can safely travel along any curve with a radius greater than the minimum safe radius. The driver will therefore try to follow curves with higher radii when going through corners but only to the point where the maximum safe speed reaches the maximum speed of the vehicle. Increasing of the curvature when it is not possible to increase the speed adds to the distance the car travels but it does not increase the average speed and therefore leads to a longer lap time.

The trajectory of the vehicle going through a corner can be split into several stages. First, the vehicle aligns itself with the outer edge of the track. Second, the vehicle must adjust its speed so it can safely stay on the curve. It is common to use the brakes mostly while the vehicle is moving straight to avoid locking the wheels in mid-turn. Third, the car starts turning at a turn-in point to go through the apex of the curve at the desired moment. Fourth, the vehicle goes through the apex, which is the point, where it is the closest to the inner edge of the track. Next, the vehicle starts exiting the corner and enters another part of the track.

There are two common ways of going through a corner, a classical one and one called late apex. The classical way of leaving the corner is to keep the constant curvature of the turn and to align the vehicle with the outer edge of the track. The late apex is achieved by starting to turn into the corner later and exiting the corner much closer to the inner edge of the track than the classical way. During the late apex turn, the vehicle slows down more before entering the corner and it

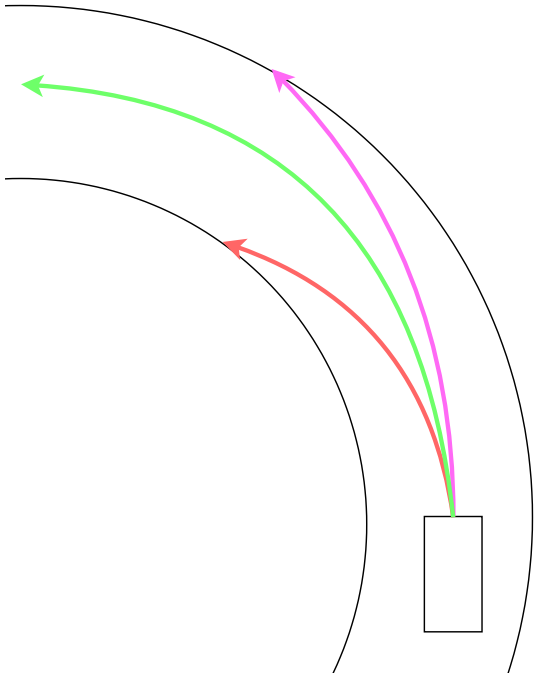


Figure 2.1: This figure shows the difference between oversteer and understeer. The red (leftmost) arrow shows oversteer, when the vehicle turns more tightly than it should and it collides with the inner boundary of the track. The purple (rightmost) arrow shows understeer as the vehicle does not turn enough and it collides with the outer boundary of the track of the corner.

straightens the line before hitting the apex allowing for greater exit speed. The comparison between these two lines can be seen in Figure 2.2.

Another factor which affects the speed at which the driver can go through a corner is the shape of the track around the corner. When there is a straight stretch following the corner, the driver can maximize the speed at which the vehicle leaves the corner to increase the average speed. If there is another corner immediately after the first one, the driver must plan ahead and adjust the speed before entering the first corner to a level at which the exit speed of the first corner will be appropriate to go through the following corner.

From the description of the racing behavior of human expert drivers, it is apparent that the key components of choosing the optimal racing line are the knowledge of the behavior of the vehicle and the shape of the track. The knowledge of the track gives the driver an opportunity to plan how he or she is going to approach driving on the track. The driver can plan how to approach each individual corner and what speeds and turning radii are optimal and still safe.

2.2 Artificial Agent

Definition 1. *A rational agent is an entity which gathers information from the environment through sensors and changes the state of the environment in order to maximize a performance measure.*

A racing driver can be thought of as a rational agent as defined in Definition 1.

The driver observes the position of the vehicle on the racing track and the state of the vehicle as it moves along the track. He also observes the distance to the boundaries of the track and to any obstacles which can be present on the track. The driver reacts to these perceptions by giving control inputs to the vehicle through the steering wheel, brake and accelerator pedals, and shifting into different gears. The performance measure is the lack of collisions with the boundaries of the track or any obstacles on the track and minimizing of the lap time.

An artificial racing agent would use electronic sensors such a LIDAR, wheel encoders, an IMU, or a camera to observe the state of the environment. Based on this observation, the agent can estimate its state in the world and the state of other entities in the world, such as obstacles or opponents. Based on this information, the agent has to select a control input for the actuators of the vehicle.

The agent can use the information about its initial state and calculate a time-optimal racing line from this initial state through the whole circuit up to the finish line using a trajectory planning algorithm. Depending on the size of the circuit, this could be a computationally expensive operation. If later on there is a need to re-plan the trajectory during the race, because the vehicle could not follow the trajectory accurately and it is necessary to come up with a contingency plan, the vehicle might have to repeat the expensive calculation. Instead, the agent can analyze the shape of the circuit and identify the corners and focus its effort only on the next two or three corners ahead of him.

2.2.1 Decision Making Process

An agent is sometimes described in the form of an agent function. This function takes the data from the sensors and outputs a command to the actuators. The command is then executed and it has an effect on the environment. In the next step we measure the changed state of the environment and the agent reacts to this changed state. The agent can select the next command in an order to correct the outcome of the previous command when it is different from the predicted outcome. This process is then repeated over and over in a so-called closed-feedback loop.

In order to avoid describing the logic of the racing agent in a single complicated function, we can divide the decision making process of the agent into several smaller independent sub-problems: localization, track analysis and waypoint selection, trajectory planning, and trajectory following. Some of these sub-problems output information which is necessary as an input for another of these sub-problems they form nodes of a dependency graph of the decision making process which is visualized in Figure 2.3.

There is one more reason to break the task down into several sub-problems. The rate at which the individual sub-problems produce outputs is different and they work asynchronously. While the trajectory following algorithm should react to any location update with an action to correct the movement of the vehicle and it should do this many times per second, the planning process of a trajectory will take some non-trivial time and so the trajectory planning process will have a much lower output frequency.

Current state As the vehicle moves on the track, the agent needs to know its current state: its position on the map, orientation, and speed. A localization algorithm collects the data from different sensors and estimates the current state of the vehicle. The accuracy and the frequency of state updates depend greatly on the capabilities of the sensors and the processing power of the on-board computer. The agent takes the raw data from a localization algorithm and produces a convenient representation of the current state for track monitoring, trajectory planning, and trajectory following.

Track analysis & monitoring Before the race begins, the agent analyses the track and finds the corners and bends on the track and marks a coordinate of a point near the apex of the corner and forms a list of waypoints along the track. During the race, the agent will keep track of which of the waypoints discovered during track analysis it drove past the last. The waypoint selection process will publish the following n waypoints as the next goal for trajectory planning. The parameter $n \in \mathbb{N}$ is the lookahead of the agent. The selection of this parameter is a trade off between the quality of the trajectory and the size of the configuration space that will be searched. This directly affects the update rate of the trajectory planning process.

Trajectory planning The trajectory planning algorithm finds a feasible trajectory from the last state of the vehicle estimated by the localization algorithm through the selected waypoints. The trajectory planning process will start planning a new trajectory as soon as it finishes planning the previous one. As the agent drives along the circuit and drives past a waypoint, the trajectory planning algorithm receives a new sequence of waypoints and the next trajectory it plans will account also for the next corner of the circuit within the lookahead instance. Planning can be a slow process even when we try to reduce the size of the search space as much as possible. For the practical application, the planning algorithm must update the trajectories faster than the vehicle moves through the circuit. If the agent comes to the end of an old reference trajectory and it has no update, it has to stop and wait for an update. This would obviously affect the lap time which is undesirable.

Trajectory following The latest reference trajectory and the current estimate of the state of the vehicle is used to calculate the next command for the actuators of the vehicle. In the ideal case, this trajectory following algorithm will send the commands to the hardware at the maximum possible rate of which the hardware is capable. In practice we are limited by the rate at which we receive the state estimates from the localization algorithm. The trajectory following algorithm is also responsible for collision avoidance either by actively guiding the vehicle around the obstacles it might hit, or by slowing down the vehicle while waiting for a new reference trajectory.

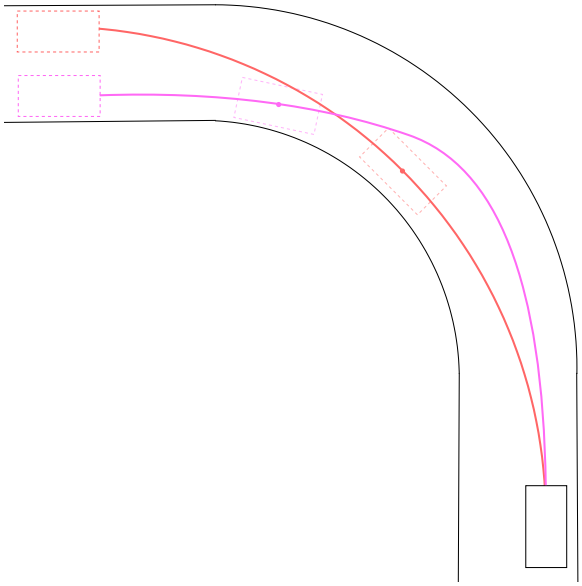


Figure 2.2: Apex of the racing line is the point where the car is the closest to the inner boundary of the track. This figure shows two racing lines in a left turn. The red line is a classical racing where the car starts and ends at the track and ends at the outer boundary of the track. The late apex line is purple where the car starts braking and turning into the corner later and exits the corner at the inner boundary of the track.

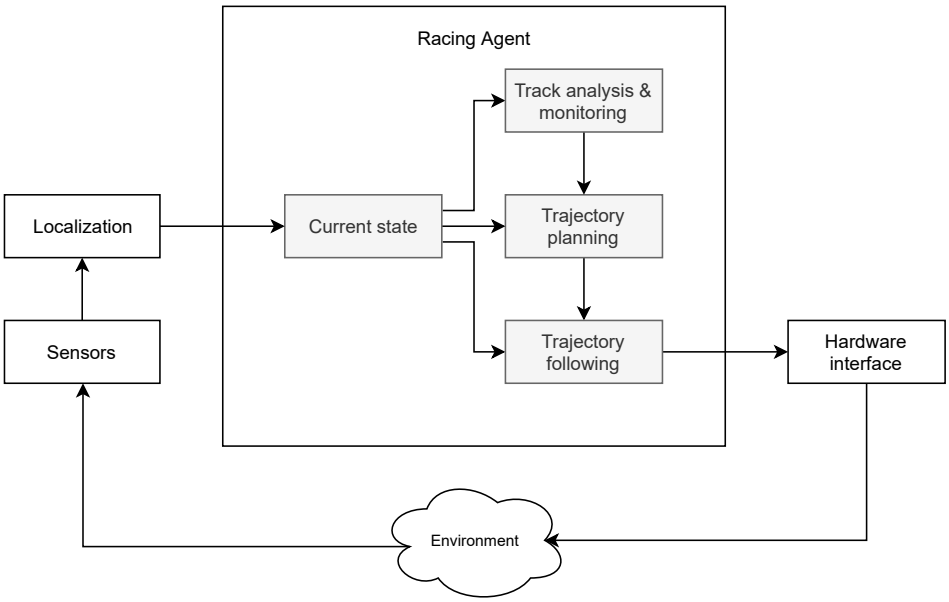


Figure 2.3: A diagram of the decision making process of the racing agent.

3. Trajectory Planning

The main focus of this thesis is the problem of trajectory planning for a fast moving car. In this chapter, we will analyze this problem in depth. We will first look into the planning problem in general and then we will discuss what the term "fast moving cars" means and how trajectory planning for a fast car is different from the general planing problem.

With the knowledge of the theory, we will formulate our trajectory planning problem. We will consider several well-known search algorithms used to solve planning problems and adapt them to our problem.

The solution to the planning problem will be a trajectory which is both the path and the speed profile the vehicle should follow. If a robot is able to follow this trajectory, it will have an advantage over reactive steering algorithms, such as end-to-end driving, as it will be able to go through a series of corners efficiently by slowing down and accelerating at convenient times and by following an appropriate racing line. We will discuss a way of following the trajectory later in Chapter 4.

3.1 Introduction to Planning

In this section, we will give a brief overview of the basic concepts of automatic planning and planning under differential constraints. The main source of the information for this chapter is a book by Steven M. LaValle called *Planning Algorithms* [19]. This book describes all the topics in this section in much more detail and it is a good source of further information on this topic.

3.1.1 Automatic Planning

Planning is the task of solving a problem by finding a sequence of actions, which transforms the world to some desired state. The world of a planning problem is defined in terms of states and actions.

A single state is a full description of all the important aspects of the world. A state can be encoded in many different ways, for example it can be a set of logical propositions which hold true in the given state or an n-dimensional vector of real numbers. All possible states of the world form a state space.

An action is a way of changing a state of the world into a different state. Not all actions can be applied in all states and so the set of possible actions can differ from state to state. All actions together then form an action space.

The planning task is to find a sequence of actions, which changes the state of the world from a given initial state to some desired state. Such sequence of actions is referred to as a feasible plan.

With the intuition of what a planning problem is, we can formulate it formally:

Definition 2 (Planning Problem)**.** *A planning problem is a tuple* (X, U, f, x_0, g) *consisting of:*

1. *A nonempty set* X *of world states, called the state space.*

2. *For each state $x \in X$, a set of actions $U(x)$. A set of all actions $U = \bigcup_{x \in X} U(x)$ is called an action space.*

3. *A state transition function f defined for every $x \in X$ and $u \in U(x)$, which produces the state of the world after applying the action u to the state x.*

4. *An initial state of the world $x_0 \in X$.*

5. *A goal region $X_g \subset X$.*

Definition 3 (Feasible Plan)**.** *A solution to a planning problem (X, U, f, x_0, g) is a feasible plan, which a finite sequence of actions $\langle u_0, u_1, \ldots, u_k \rangle \in U^*$ such that:*

$$\forall i \in \{0, 1, \ldots, k\} : u_i \in U(x_i) \wedge x_{i+1} = f(x_i, u_i)$$
$$x_{k+1} \in X_g.$$

Definition 4 (Optimal Plan)**.** *Let (X, U, f, x_0, g) be a planning problem and $\Pi = \{\pi \in U^* \mid \pi \text{ is a feasible plan}\}$ a set of all feasible plans. We say that a plan $\pi^* \in \Pi$ is an optimal plan with respect to a cost function $\gamma : \Pi \to \mathbb{R}$ if*

$$\pi^* = \arg\min_{\pi \in \Pi} \gamma(\pi).$$

Reachability Graph

We can imagine the states as vertices of a graph $G = (V, E)$ and the applications of actions through the state transition function define directed edges between the vertices:

$$V = X$$
$$E = \{(x_1, x_2) \mid \exists u \in U(x_1) : x_2 = f(x_1, u)\}.$$

This graph can in general have several subgraphs. We are only interested in the component, which contains the initial state x_0 and all the vertices which are reachable from x_0 via a directed path. A feasible plan is then a directed path in the graph starting in the initial state vertex x_0 and ending in any of the goal state vertices. Finding a path in a graph is a well-studied problem and there are several efficient algorithms to solve it, such as the Dijkstra algorithm or its extension called A*.

The size of the state space and the action space has great impact on the way how we approach the planning problem and how we find a solution. If the graph is finite or if the number of vertices is countably infinite and the branching factor is finite, we can find the solution (if one exists) with a systematic search algorithm in a finite amount of time. If no solution exists and the graph is infinite, the algorithm will keep trying different plans infinitely. In practice this can be avoided with a limiting criterion, such as maximum length of a plan.

When the number of vertices is uncountably infinite or the branching factor is infinite, the problem becomes much harder and we cannot rely on a simple graph search anymore. We will soon show that the state space of all vehicle configurations in our problem is uncountably infinite. These problems can be solved using *sampling algorithms*.

We must keep in mind that the number of states of the system can be very high even if it is finite because the state space represents all combinations of the possible values across all state space dimensions.

3.1.2 Planning Under Differential Constraints

When describing the motion of a car-like robot on a two-dimensional plane, we assume that we can treat it as a rigid body, usually represented as a bounding rectangle. The configuration of the rigid body is then described by the pose of the vehicle: the (x, y) Cartesian coordinates of a fixed reference point of the body in the world reference frame, and the heading angle θ between the longitudinal axis of the vehicle and the x axis of the world reference frame. The (x, y) location is in some bounded area $P \subset \mathbb{R}^2$ and the heading angle is an arbitrary angle $\theta \in [0, 2\pi)$. This simple configuration space already has three continuous dimensions and it is uncountably infinite.

The kinematics and dynamics of robots are typically described by differential equations. These equations give us the velocities at which the state of the robot changes when an action is applied. As an example of these constraints, we can look at a model of a *simple car* [19, Section 13.1.2.1].

Example. The state space of a simple car will consist of the poses in an infinite 2D plane (x, y, θ) as described earlier. The control inputs are two dimensional vectors (u_s, u_φ), where u_s is the commanded speed of the vehicle in the direction perpendicular to the rear axis, and u_φ is the steering angle of the front wheels. For a better understanding of this example, see the Figure 3.1.

For a car with a wheelbase length $L \in \mathbb{R}$, the velocity can be approximated by this set of equations:

$$\begin{aligned} \dot{x} &= u_s \cos\theta \\ \dot{y} &= u_s \sin\theta \\ \dot{\theta} &= \frac{u_s}{L} \tan u_\varphi. \end{aligned} \tag{3.1}$$

The configuration space of all possible transformations of the robot and possibly additional variables required to keep the state of the kinematics and dynamics of the robot together form a continuous *state space* X. In the example of the simple car, the state space would be just the configuration space itself, therefore $X = \mathbb{R}^2 \times [0, 2\pi)$, but different models could have more dimensions as we will see in Section 3.5.

Obstacles Additionally, we must be able to split the state space X into two complementary subsets: the obstacle region $X_{obs} \subseteq X$ and the free region $X_{free} = X \setminus X_{obs}$. Only the states in X_{free} can be entered safely while the states in X_{obs} must be avoided to prevent collisions of the robot with obstacles.

State Transition Function In order to be able formulate the planning problem for a system with differential constraints, we must change the meaning of the *state transition function* f from the previous Definition 2. This function used to produce the next state x' after an action u is applied to a state x, i.e. $x' = f(x, u)$. When the state space is continuous, the outcome of an execution

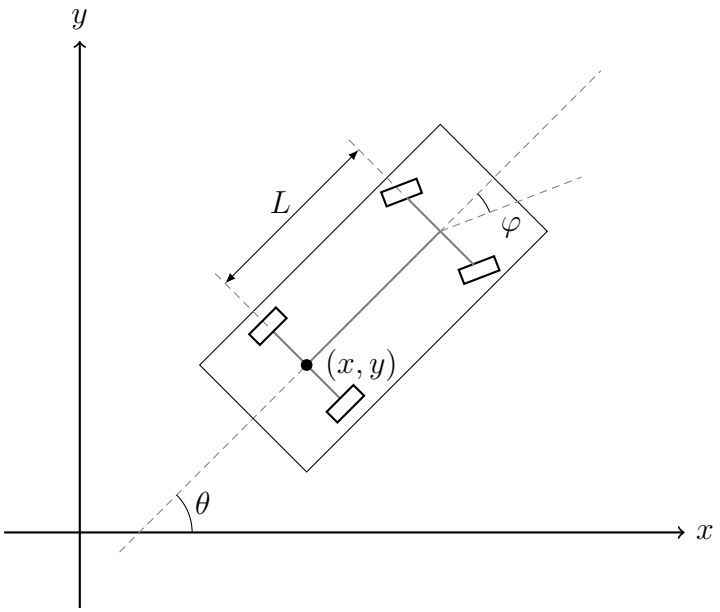


Figure 3.1: The simple car from the example has three state variables. The speed and the steering angle are action variables and they can change at any moment, so the vehicle can stop on a spot and change the steering angle instantaneously, which is of course not possible in a real world car.

of an action depends on for how long it is being applied. Instead of a direct transition to the next state, the function f will now express the velocity in the state space as defined by the differential constraints:

$$\dot{x} = f(x, u).$$

The function f must be defined for every $x \in X$ and $u \in U(x)$. The task of a planning algorithm is then to find an *action trajectory* which produces a collision-free *state trajectory* which reaches a goal state. An *action trajectory* is a continuous function $\tilde{u} : T \to U \cup \{u_t\}$ which maps an infinite interval $T = [0, \infty)$ to an action, which should be executed at the given point in time. From a point in time $t_{end} \in [0, \infty)$, a termination action u_t can be applied to mark the end of the action trajectory ($\forall t > t_{end} : \tilde{u}(t) = u_t$). To *state trajectory* corresponding to the action trajectory $\tilde{u}$ is a function $\tilde{x} : T \to X$, such that $\tilde{x}(0)$ is a given initial state x_0, and $\forall t \in (0, t_{end}]$:

$$\tilde{x}(t) = \tilde{x}(0) + \int_0^t f(\tilde{x}(\tau), \tilde{u}(\tau))d\tau, \tag{3.2}$$

such that $\forall \tau \in [0, t_{end}] : \tilde{u}(\tau) \in U(\tilde{x}(\tau))$. A collision-free trajectory adds a requirement to go only through the free region ($\tilde{x}(\tau) \in X_{free}$).

Definition 5 (Planning Problem Under Differential Constraints)**.** *A planning problem under differential constraints (X, U, f, x_0, g) is an extension planning problem given in Definition 2, such that:*

1. *The state space X is continuous.*

2. *A state transition function f defined for every $x \in X$ and $u \in U(x)$, which produces the velocity in the state space after applying the action u to the state x: $\dot{x} = f(x, u)$.*

Definition 6 (Feasible Plan)**.** *A solution to a planning problem under differential constraints (X, U, f, x_0, g) is a feasible action trajectory $\tilde{u}$ when $\exists t_{end} \in (0, \infty)$ such that:*

$$\forall t \leq t_{end} : \tilde{u}(t) \in U(\tilde{x}(t))$$
$$\tilde{x}(t) = \tilde{x}(0) + \int_0^t f(\tilde{x}(\tau), \tilde{u}(\tau))d\tau$$
$$\tilde{x}(t_{end}) \in X_g.$$

Sampling-Based Planning Methods

In order to be able to plan in a world with a continuous state space and continuous time, sampling-based methods can be used. The main idea is to discretize the action trajectory by using the same action for a non-trivial time period. This effectively discretizes time into time intervals, which usually have a fixed length Δt or the duration depends on the action which is used ("a motion primitive" [19, Section 14.2.3]). The action trajectory can then be expressed as a simple sequence of actions and for every action in the sequence it is possible to determine the time at which it is supposed to be applied and for how long it is supposed to be applied. We can still obtain a continuous trajectory through the state space by integrating the action sequence.

Ideally, every segment of a state trajectory obtained by integrating an action from a state over a time period should be checked for collision. In practice, this could be very resource intensive, and so only a few samples (sometimes only the end state of the segment) are tested using a "black box" collision detection function to see if the segment is in X_{obs} or X_{free}.

If the action space is infinite, it should be sampled as well and only a finite number of actions should be considered. This makes it possible to construct a graph where samples of X are the vertices and the edges between them are actions. Starting from x_0, we compute all possible state trajectories starting in this state with all the actions in the finite set of actions over a time period (either a fixed Δt or a time associated with the action) and add the final states as vertices to the graph and connect them with directed edges to the origin state. We repeat this for all added states over and over again.

The size of the graph is affected also by the discretization of time. The shorter the time interval is, the larger the graph will be. Nevertheless, we can perform graph search over and find solutions to planning problems with differential constraints. We will describe concrete search algorithms for our problem in Section 3.3.

The discretization of time might make some states of the state space unreachable and this could make finding a feasible plan impossible. In order to achieve resolution-completeness of the sampling based algorithm, every time a plan is not found, the discretization should be made finer (e.g., by decreasing the length of the time intervals by half) [19, Chapter 14.2].

3.2 Trajectory Planning For Fast Moving Cars

In this section we will discuss what it means to plan a trajectory for a fast moving car. What is the difference between trajectory planning for a slow moving car and for a fast moving car? What do we even consider to be a "fast moving car" and what is just a slow moving car?

3.2.1 Terminology Clarification

A fast moving car is not a widely used term and it does not have a formal definition. Intuitively, we would consider a "fast moving car" to be a Formula 1, a rally car, or a different racing car. The racing drivers push their vehicles to their limits in order to be the first one behind the finish line. We might also consider ordinary driving on a motorway to be "fast", especially during a lane change maneuver or when we accelerate to overtake a vehicle in front of us. We might even consider driving at lower speeds to be "fast" when it involves taking sharp turns.

One thing these scenarios have in common is that the car reaches a speed at which it can become hard to steer the car in a specific direction because the tires might lose grip and the car might start moving sideways or in a more extreme scenario the car could roll over sideways. The speed which could be considered "fast" varies based on the radius of the turn.

For the purposes of this thesis, we will formulate the term in the following definitions:

Definition 7 (Handling limits of a vehicle)**.** *We say, that a car is moving at its handling limits during a turn of a radius of* $r \in (r_{min}, \infty]$ *meters, if it is driving close to a speed* $v_{max,r}$ *at which the total force vector acting on the body of the vehicle would exceed the forces of the tires which keep the vehicle traveling at the constant radius* r*.*

The minimum turning radius $r_{min} \in \mathbb{R}$ is a constant specific for a given vehicle. It refers to the minimum turning radius which the vehicle can achieve at a very low speed and with the maximum steering angle possible. We can consider driving straight as driving along a circle with the radius of ∞ meters.

Definition 8 (Fast moving car)**.** *A car is moving fast when it is approaching its handling limits during a turn.*

Definition 9 (Trajectory planning problem for a fast moving car)**.** *We say that a trajectory planning problem for a fast moving car is a planning problem under differential constraints. The solution to this problem is a collision-free time-optimal state trajectory and the vehicle does not at any point in time exceed its handling limits. The state trajectory describes both a path and a speed profile for the given vehicle and track.*

To achieve a time-optimal trajectory, we will search for a trade off between the length of the path and the speed at any given moment. To keep within the handling limits of the vehicle, our differential constraints must closely model the behavior of the vehicle. Even if our model of the vehicle describes its behavior

well, it is more than likely that the robot will deviate from the planned trajectory at some point. The noise in sensor readings, imperfections in the actuators, or imprecise map of the track, all of these factors will contribute to errors in trajectory tracking. At some point, the difference between the planned path and velocity profile of the vehicle will be so large, that it will be advantageous to create a new plan from the current pose and speed of the vehicle. To make this possible, we need to be able to re-plan the trajectory in a short period of time.

Given our assumption, that we will most likely have to re-plan the trajectory at some point in the future anyway, it does not make too much sense to plan the trajectory for the whole circuit at once. The longer the trajectory we are planning, the longer it will take to find a suitable plan. Instead, we could split the track into multiple shorter segments and plan a trajectory only for a few of the segments directly in front of the current pose of the vehicle.

When the vehicle is moving fast, a belated or too early decision to change the direction or adjust the speed of the vehicle might result in a crash or a sub-optimal trajectory. Fast reaction times are therefore key. If we plan a trajectory for only a limited stretch in front of the vehicle, we must be able to find the plan for the following stretch before we reach the end of the current reference trajectory. Failing to do this, the vehicle will not have a trajectory to follow and it would probably have to preform some kind of an emergency braking in order to prevent collisions, or it would be forced to resort to some simpler reactive steering strategy, which does not involve planning ahead and which would most likely be sub-optimal.

We can now summarize the discussion in this section into a set of requirements for the trajectory planning algorithm which will make it more suitable for fast moving cars:

- The whole track should be split into smaller segments, so that we do not have to waste resources to plan a trajectory for segments which will most likely not be reached by the time we will need to re-plan the trajectory.
- The differential constraints of the vehicle must accurately describe the movement of the vehicle at its handling limits and we must be able to eliminate dangerous maneuvers, which could lead to a crash or a trajectory or which could not be followed closely by the vehicle.
- The search algorithm we will use must find good solutions which are time-optimal or close to time-optimal in a very short period of time.

3.2.2 Problem Analysis

The Configuration Space

We only consider the motion of our vehicle on a two-dimensional flat surface. For simplicity, we will consider the shape of the vehicle to be a rectangle of a constant width and height, which fully encloses all parts of the body of the vehicle. This gives us a simple way of checking collisions with obstacles and by adjusting the size of the rectangle we can also add a small safety margin around the vehicle to stay on the "safe side" when closely passing obstacles.

Any position of the boundary rectangle in the 2D plane can be expressed as a rigid body transformation. We will define a *configuration space* C consisting of all these possible transformations of the vehicle. Each transformation can be expressed as a vector $(x, y, \theta) \in \mathbb{R}^2 \times [0, 2\pi)$ where (x, y) are the coordinates of the center of gravity of the vehicle (corresponding to the center of the rectangle) and θ is the angle between the x axis of the coordinate system and the longitudinal axis of the vehicle, i.e. the heading angle of the vehicle. This configuration space is continuous.

Collision Detection

A collision detection function $c_R : C \to \{0, 1\}$ divides the configuration space into two parts C_{free} and C_{obs}:

$$C_{obs} = \{x \in C \mid c_R(x) = 1\}$$
$$C_{free} = C \setminus C_{obs}.$$

The value of the function is 1 when a collision between the vehicle and the obstacles occurs and 0 otherwise. The subscript R represents a specific instance of the collision detection function for a rectangular shape of a specific width and height. The task of the collision detection function is to determine if the boundary rectangle of the vehicle in the given configuration $x \in C$ collides with some obstacle in the world or not. This function is the only way how the planning algorithm can determine the shape of the track and any additional obstacles which appear on the track.

Representation of Obstacles We represent the obstacles in the world as an occupancy grid. An *occupancy grid* is a matrix $O_r^{m \times n} \in \{0, 1\}^{m \times n}$ where $m, n \in \mathbb{N}$ is the size of the grid and $r \in \mathbb{R}$ is the resolution. Every element of the matrix represents a square tile of $r \times r$ meters placed in a grid covering an area of $rm \times rn$ square meters. The tiles can be in two states: free (0) or blocked (1).

To check if a single point of the world $(x, y) \in \mathbb{R}^2$ collides with an obstacle or not, we must first determine the coordinates of the tile in which it belongs:

$$x_r = \left\lceil \frac{x}{r} \right\rceil$$
$$y_r = \left\lceil \frac{y}{r} \right\rceil$$

If the point lies outside of the area covered by the occupancy grid, we treat it as an obstacle. Otherwise we look into the appropriate cell of the matrix. The collision detection function c_1 for a single point is defined as follows:

$$c_1(O_r^{m \times n}, x, y) = \begin{cases} 1 & x_r < 1 \vee y_r < 1 \vee y_r > m \vee x_r > n \\ (O_r^{m \times n})_{y_r, x_r} & \text{otherwise.} \end{cases}$$

We assume that the occupancy grid is aligned with the origin of the world reference frame and that the column indexes grow parallel to the x axis and the indexes of rows parallel to the y axis. If there is a transformation between the origin of the world reference frame and the occupancy grid, the point (x, y) has

to be rotated and translated appropriately before the coordinates of the corresponding tile can be determined. Please note that matrix rows and columns are indexed from 1, unlike for example arrays in the C and many other programming languages, which start indexing at 0.

We chose the occupancy grid because it is a standard way of representing 2D maps in Robot Operating System (ROS). We used these technologies to build our custom experimental vehicle and therefore we also used this map representation.

Moving Obstacles In this thesis, we will not consider moving obstacles. If it was necessary, we would have to extend the definition of the collision detection function to include a time parameter and it would have to extrapolate and predict a state of the world at the given time and check the collision against this prediction. The rest of the planning algorithm would not be affected by this extension.

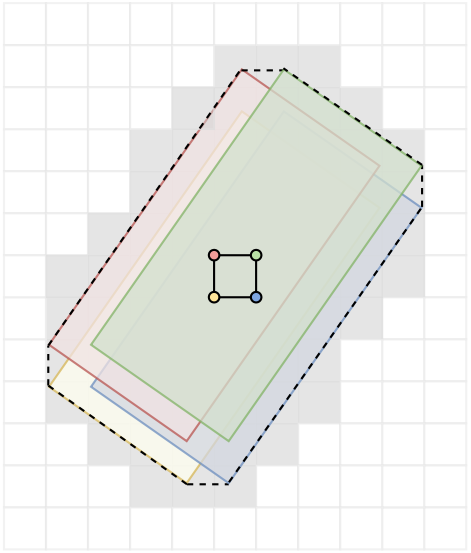
Collision Detection Implementation The collision detection function must be evaluated every time the planning algorithm considers a use of an action from some state. If the action would lead to a state which would collide with an obstacle, this action should not even be allowed. The complexity of the collision detection function will therefore have a great impact on the performance of the algorithm on real hardware.

The footprint of the body of the vehicle is a rotated rectangle and we have to check if any of the parts of this rectangle does not hit an obstacle cell. We considered two versions of the collision detection algorithm:

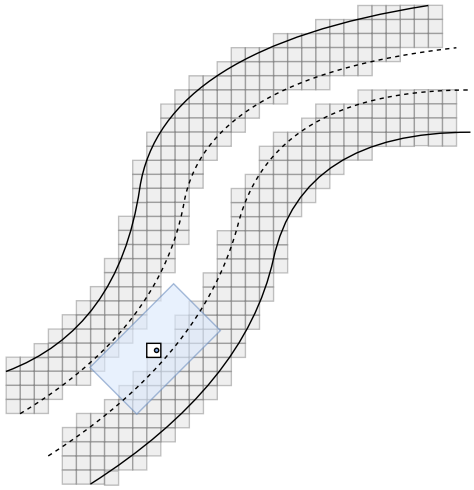
1. Pre-calculating small occupancy grids for several configurations of the vehicle and overlaying them with the occupancy grid representing the world.

2. Inflating the obstacles in the occupancy grid and performing a single point check.

The idea of *the first method* is to calculate a list of tiles relative to the tile in which the center of the rectangle lies which the vehicle could collide with. We would split the heading angle dimension of the configuration into several regular intervals. For each of these intervals we would take the value in the middle and use it as a heading angle θ. We would then rotate the bounding rectangle by the angle θ and mark which cells of a grid with the resolution r it would overlay. We must take into consideration that the center of the rectangle can be at any point of the tile and different cells would be overlaid for example when the center coincides with the top left corner of the cell or the bottom right corner. Figure 3.2a depicts this idea. To check collisions, we would first determine the cell corresponding to the center of the bounding rectangle of the vehicle and determine the discretization of the heading angle θ. For the discretized heading angle, we would then remember a list of integer coordinates of the cells which would be occupied relative to the center of the rectangle. We would then check the state of the occupancy grid for the tiles occupied by the footprint of the vehicle and if any of them was marked as X or if the coordinate would be outside of the bounds of the $m \times n$ grid, we would report a collision.



(a) The figure shows which tiles can be occupied by the vehicle when the heading angle is close to θ for a center anywhere in the cell which contains its center. The dashed line is a boundary of all possible configurations of the vehicle. The dark gray squares are the grid cells which should be tested for collision.



(b) The obstacles are "inflated" by the radius of the vehicle and every cell, which is closer to any obstacle than the radius of the vehicle is marked as dangerous. To detect collision, it is necessary to check if the cell which contains the center of the vehicle body is dangerous or not.

Figure 3.2: These figures compare two approaches to collision checking. The second approach requires fewer operations whenever we need to test a collision and therefore we chose the "obstacle inflation" method.

The second idea is simpler. First, we calculate the radius r of the bounding circle of the rectangular footprint of the vehicle. We then create a new occupancy grid in such a way, that for each cell we check its distance to the nearest cell occupied by an obstacle. If this distance is greater than r, then the cell is considered safe, otherwise the cell is considered dangerous. To check for collisions of a vehicle at any given position, we only check whether the occupancy grid cell which contains the center of gravity of the vehicle is safe or dangerous. This method is depicted in Figure 3.2b.

The obvious advantage of the first method is its higher accuracy. The second method could be insufficient for tasks such as perpendicular parking between two cars, where the whole space between the two cars could be filled with the inflated obstacles. The occupancy grid can in principle change between two executions of the planning algorithm (e.g., a new obstacle is discovered) and therefore we might have to calculate the inflation before any invocation of the planning algorithm. In practice, inflating the obstacles takes very little time when compared to the planning algorithm itself.

In the end, we implemented a combined approach. We first use the *second* method to efficiently rule out collision whenever the vehicle is far from any obstacle. If the center of the gravity of the vehicle is in a dangerous cell though, we use the more expensive *first* method to get a more precise collision detection.

Waypoints

When we analyzed the problem earlier in this chapter, we decided to plan a trajectory for the next $l > 0$ track segments ahead (each segment should ideally correspond to a corner of the track or to a straight stretch between two corners). These segments will be given in the form of a list of points at the end of each segment in the coordinate frame of the track $\hat{w} = \langle \boldsymbol{w_0}, \boldsymbol{w_1}, \ldots, \boldsymbol{w_l} \rangle$.

To ensure that our vehicle goes around the circuit in the correct direction and it follows the track correctly even in cases, when the track intersects itself and forms loops, the trajectory of the vehicle must pass these waypoints in the given order. The concept of "passing a sequence of waypoints" is formulated by the following definitions:

Definition 10 (Directly Visible Points)**.** *We say that two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^2$ are directly visible points, if there is no obstacle in an occupancy grid $O_r^{m \times n}$ on the line between these two points. The set $D(O_r^{m \times n}) \subseteq \mathbb{R}^2 \times \mathbb{R}^2$ for the given occupancy grid captures this relation:*

$$(\boldsymbol{x}, \boldsymbol{y}) \in D(O_r^{m \times n}) \iff \forall t \in [0, 1] : c_1(O_r^{m \times n}, \boldsymbol{x} + t(\boldsymbol{y} - \boldsymbol{x})) = 0.$$

.

Definition 11 (Waypoint)**.** *Waypoint goal region in an occupancy grid $O_r^{m \times n}$ is a set of configurations*

$$G_r(\boldsymbol{w}) = \left\{ c \in C \mid \|(c_x, c_y) - \boldsymbol{w}\| < r \wedge ((c_x, c_y), \boldsymbol{w}) \in D(O_r^{m \times n}) \right\},$$

where $r \in \mathbb{R}$ is the radius of the goal region and $\boldsymbol{w} \in \mathbb{R}^2$ is the center of the waypoint. $G_r(\boldsymbol{w})$ is a set of all configurations which are considered to pass the waypoint.

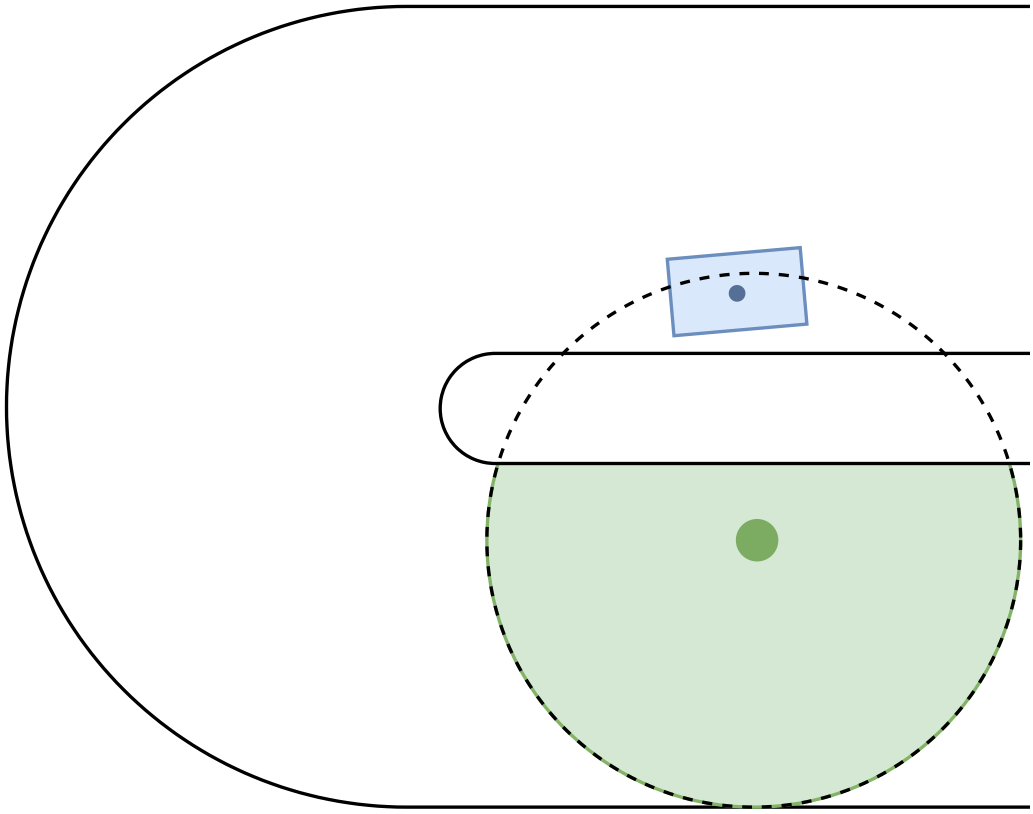


Figure 3.3: If the waypoint did not require direct visibility, the blue car would pass the waypoint whose radius is marked with the dashed line. The green area represents the actual waypoint area.

The notation $\|\cdot\|$ represents the Euclidean norm in $\mathbb{R}^2$. The choice of the radius of the *waypoint goal region* is not too important. The radius should be large enough to cover the whole width of the track, so that it does not force the vehicle to change the trajectory just to pass the artificially added waypoint. On the other hand, the waypoints should not overlap very much. The requirement of *direct visibility* is there to prevent scenarios when the vehicle could pass a waypoint "behind a wall" if the radius is too large, as is shown in Figure 3.3.

The State Space

The complete *state space* X of the vehicle will be a combination of the configuration space C, the number of passed waypoints, and several additional variables representing the kinematics and dynamics required by the differential constraints.

We will define a function $\kappa : X \to C$ which returns the configuration of the vehicle in the given state, and also $\kappa_{x,y} : X \to \mathbb{R}^2$ to obtain just the position of the given state.

The state space also has to contain an addition finite dimension for the number of waypoints which have already been reached. The size of this dimension will always depend on the number of waypoints l. To denote the number of passed waypoints of a state, we will use a function $\mu : X \to \{0, 1, 2, \ldots, l\}$.

We will discuss the choice of appropriate vehicle models in Section 3.5 and we will define a concrete state space for each of the vehicle models.

We can also extend of the partitioning of the configuration space into C_{free} and C_{obs} for the state space. Collision depends only on the configuration of the vehicle body, not on the kinematics and dynamics of the vehicle. We can therefore define X_{free} and X_{obs} by the following expression:

$$\forall x \in X : (x \in X_{obs} \iff \kappa(x) \in C_{obs}) \wedge (x \in X_{free} \iff \kappa(x) \in C_{free}) .$$

Initial State At the start of the race, the vehicle is expected to be situated at a predefined starting location of the racing circuit with its heading angle identical to the direction of the race. The number of passed waypoints is zero. The vehicle starts from a standstill with the wheels not spinning and the front wheels pointing the heading direction of the car.

During the race, the speed and the steering angle have to be measured using sensors. We must take into account that this state is most likely slightly inaccurate due to the errors in the measurements and due to a delay between the measurement and the start of planning. By the time the planning algorithm finds a trajectory, the state vehicle will have already changed and it will be different from the initial conditions of the planning algorithm.

Action Space

Our vehicle has two actuators which can be controlled by:

- Steering servo which we can move to a desired position achieving a specific steering angle between the maximum left position, center position, and maximum rotation to the right.
- Motor which can be set to a specific Revolutions Per Minute (RPM) and a direction of rotation when there is no load. The RPM will differ based on the load of the vehicle and the motor will require more voltage to achieve some RPM when the load is increased.

The electric signal we send to the actuators is interpreted as a target value and the actuator adjusts its output to match the target value over a time period at a rate which we cannot control. We define the *action space* for these actuators like this:

$$U = \{(\delta_t, \tau_t) \mid \delta_t \in [\delta_{min}, \delta_{max}] \, \tau_t \in [-1, 1]\}$$

U is a set of tuples of a steering angle δ_t, and a throttle position τ_t. These actions are an abstraction of the signals which would be sent to the hardware. Negative throttle position τ_t means that the motor should spin in reverse, while positive values should result in the motor spinning in the normal direction of travel. When the vehicle is moving in a direction and the action specifies a throttle position in the opposite direction, the vehicle might engage brakes, if it is equipped with some. Since these actions represent merely the target values of the state of the actuators, our actions do not have any preconditions for the state in which they can be executed.

The action space U as we defined it is infinite. In practice, our actuators can be set only to a fixed number of values. The Pulse Width Modulation (PWM) signal which controls the servo motors can only encode a finite number of levels. For a specific vehicle hardware, we can use only a finite subset $U_f \subset U$, $|U_f| \in \mathbb{N}$ of valid actions.

Discrete-Time Model

Earlier in Section 3.1.2, we described an action trajectory as a continuous function of time which specifies the action at any given moment. This concept is unrealistic for our use case because we want to apply it to real hardware. The speed of an electric motor controlled by a PWM signal can be adjusted only once per a duty cycle period. This information alone gives us a good reason to treat time not as a continuous function, but as a sequence of evenly spaced samples of time points, in which the planning algorithm can choose the next action. The choice of the sampling period will be a trade off between maximum possible control over the hardware and the number of decisions the planning algorithm will make.

We will integrate state transition function over a constant time period Δt with a constant action $u \in U_f(x(t))$ based on (3.2) and we will call the resulting function $f_{\Delta t}$ the *system simulator*:

$$x(t + \Delta t) = f_{\Delta t}(x(t), u) = x(t) + \int_0^{\Delta t} f\left(x(t + \tau), u\right) d\tau, \tag{3.3}$$

where $\forall \tau \in [t, t+\Delta t] : x(\tau) \in X \wedge u \in U_f(x(\tau))$. The state trajectory function $x(t)$ can be reduced into a sequence $\langle x_0, x_1, x_2, \ldots, x_k \rangle$ and the action trajectory to a sequence of $\langle u_0, u_1, u_2, \ldots, u_{k-1} \rangle$, where the i-th step of the sequence x_i corresponds to $x(i\Delta t)$, therefore we can write

$$x_{i+1} = f_{\Delta t}(x_i, u_i).$$

Executing some actions for a time period Δt might result in a collision with an obstacle. To avoid this problem, we will limit the action set of a vehicle state by incorporating the collision detection function and the system simulator $f_{\Delta t}$ to allow only safe actions:

$$U_f(x) = \left\{ u \in U_f \mid c(f_{\Delta t}(x, u)) = 0 \right\}.$$

It is worth noting that if the vehicle is in state when its action set is empty, the vehicle is in a state when a collision within Δt is inevitable.

The *system simulator* function describes the motion of the vehicle, but it does not model how the discrete state dimension of the number of passed waypoints μ behaves. We must extend the function $f_{\Delta t}$ to increase the number of passed waypoints when the next waypoint region is entered:

$$\mu(c) = \begin{cases} \mu(c') + 1 & c' \notin G_r(\boldsymbol{w}_{\boldsymbol{\mu(c')+1}}) \wedge c \in G_r(\boldsymbol{w}_{\boldsymbol{\mu(c')+1}}) \\ \mu(c') & otherwise, \end{cases}$$

where $c = \kappa(x)$, $c' = \kappa(x')$, $x = f_{\Delta t}(x, u)$ for some action $u \in U_f$.

Goal Region

Once we find a trajectory starting in the initial state x_0 such that the sequence of states passes all the waypoints in the correct order, we found the trajectory we were looking for. The last state of the trajectory x_k must be a state in which the trajectory enters the region of the last waypoint, i.e. $\mu(\kappa(x_k)) = l$. At the same time, any trajectory which contains a state x such that $\mu(\kappa(x)) = l$ necessarily passes all the waypoints in the correct order thanks to the way we defined the transition function in the previous section.

The goal region X_g of our search problem for a sequence of l waypoints will be a set of all states which pass the l-th waypoint:

$$X_g = \{x \in X_{free} \mid \mu(\kappa(x)) = l\}.$$

Problem Formulation

In the previous sections, we described and defined the individual components of the trajectory planning problem. We can now formulate it formally:

Definition 12 (Trajectory Planning Problem)**.** *A trajectory planning problem for a fast moving car is a tuple* $(X, U_f, f_{\Delta t}, c, x_0, X_{\hat{w}})$*:*

- *A non-empty continuous set* $X \subseteq \mathbb{R}^n$*, which is the state space of the vehicle. Each state of the vehicle is represented by* $n \in \mathbb{N} : n \geq 4$ *state variables and it contains the dimensions of the configuration of the vehicle in a two dimensional plane, a discrete dimension for the number of passed waypoints, and other state variables as defined by the vehicle model which is used.*
- *A non-empty finite set* U_f *of actions, such that* $\forall x \in X, \forall u \in U_f(x) : c(f_{\Delta t}(x, u)) = 0$*, where* c *is a collision detection function.*
- *A system simulator* $f_{\Delta t} : X \times U_f \to X$ *for a fixed time period* $\Delta t > 0$*.*
- *An initial state of the vehicle* $x_0 \in X$*.*
- *A goal region* $X_g \subset X$ *for some fixed sequence of waypoints* $\hat{w} = \langle w_0, w_1, w_2, \ldots, w_l \rangle, l > 0$*.*

Time-Optimal Feasible Solution

The vehicle starts in an initial state x_0. It is controlled through an input sequence of actions applied over time at a constant rate, because we chose a fixed time interval $\Delta t > 0$ between application of any two consequent actions. We can calculate the expected evolution of the state of the vehicle by integrating the velocity from a state transition function. If we take snapshots of the state every Δt interval, we will get a sequence of states which we call a state trajectory:

Definition 13 (Feasible State Trajectory)**.** *We say that a sequence of states* $\hat{x_k} = \langle x_0, x_1, x_2, \ldots, x_k \rangle, x_i \in X$ *is a feasible state trajectory starting in the state* x_0 *when for a fixed time interval* $\Delta t > 0$*:*

$$\forall i \in \{0, \ldots, k-1\}\, \exists u \in U_f(x_i) : x_{i+1} = f_{\Delta t}(x_i, u).$$

We say that the state trajectory is feasible to emphasize the fact that it is collision-free. Remember that for each state x we do not allow any actions which would lead to a crash to be in $U_f(x)$.

This sequence contains all the information we need to reconstruct the full continuous trajectory by finding the corresponding sequence of actions. As we mentioned earlier, we do not expect the robot to be able to follow any plan perfectly, so we cannot execute the actions one by one. It is important for us to know in which the state the robot should at any given moment. The solution to our problem is therefore not a sequence of actions, but a state trajectory:

Definition 14 (Feasible Solution)**.** *For an instance of a trajectory planning problem for a fast moving car* $(X, U_f, f_{\Delta t}, c, x_0, g_{\hat{w}})$*, a feasible state trajectory* $\hat{x_k} = \langle x_0, x_1, \ldots, x_k \rangle$ *is a time-optimal feasible solution to the problem if:*

- *the final state is in the goal region:* $x_k \in X_g$*,*
- *and for every other feasible state trajectory* $\hat{y_j}$*, which ends in the goal region, it takes longer or the same time to reach the goal (i.e.* $j \geq k$*).*

State Trajectory Sub-sampling The choice of the sampling time interval Δt greatly impacts the performance of the planning algorithm. The longer the interval is, the more decisions the planning algorithm has to make to find a solution. On the other hand, a longer sampling interval could find invalid trajectories, because due to a long distance between two poses of the vehicle, the vehicle could "jump through walls" as the collision detection algorithm, which tests only the samples from the state trajectory, would give false positives.

To counter this problem, we keep the planning step Δt, but we uniformly subdivide this interval into n smaller steps and we check collision for each of these sub-samples. This technique will give us better estimate of the motion of the vehicle and reduce the inaccuracy of numerical integration but at the same time it will not increase the computational complexity of the planning algorithm because we do not increase the number of explored sequences of actions. A good pair of Δt and n has to be found experimentally to provide good precision while keeping good performance of the algorithm on real hardware.

3.3 Trajectory Planning Algorithms

With a complete formulation of the trajectory planning problem for a fast moving car, we can design a planning algorithm which will find a time-optimal feasible solution trajectory starting at any given initial state and which will pass any sequence of waypoints it is given. Our planning algorithm must respect the differential constraints of our vehicle represented by the state transition function and avoid collisions with any obstacles by utilizing the collision detection function.

In this chapter we will introduce several well-known sampling-based algorithms and their variants which we considered for solving our trajectory planning problem. We will describe the main ideas behind these algorithms and discuss if and how they could be used in our specific use case.

3.3.1 Rapidly-Exploring Random Trees (RRT)

The Rapidly-Exploring Random Trees (RRT) is a simple randomized algorithm designed by Steven M. LaValle with differential constraints of mobile robots in mind [20]. The core idea of this algorithm is to build a tree structure rooted in the initial state x_0 with branches representing state trajectories by randomly sampling the state space and using the system simulator function to find feasible trajectories in the state space.

The tree is grown iteratively by randomly sampling a state $x_{rand} \in X$ using some sampling scheme and choosing an action u from U_f which will steer the robot to a new state x_{new} towards x_{rand} without colliding with an obstacle from an existing tree node x_{near} which is the closest to x_{rand} according to some metric ρ on X. The new node is added as a vertex to the tree and it is connected via an edge to its parent node x_{near}. This process is repeated until a state trajectory which ends in the goal region is found or until a fixed number of iterations is exceeded, in which case the search ends with a failure. The pseudocode of RRT is included in Algorithm 1.

Algorithm 1: The RRT algorithm.

Input: Trajectory planning problem $(X, U_f, c, f_{\Delta t}, x_0, X_g)$, a metric on the state space ρ
Output: State trajectory or *null*
Parameter: Maximum number of iterations $K \in \mathbb{N}$

$T \leftarrow (\{x_0\}, \emptyset)$
for $k = 1 \dots K$ **do**
  $x_{rand} \leftarrow$ `SelectRandomSample(`X`)`
  $x_{near} \leftarrow$ `FindNearestNode(`T, x_{rand}, ρ`)`
  $u_{best} \leftarrow \arg\min_{u \in U_f(x_{near})} \rho(x_{rand}, f_{\Delta t}(x_{near}, u))$
  $x_{new} \leftarrow f_{\Delta t}(x_{near}, u_{best})$
  `AddVertex(`T, x_{new}`)`
  `AddEdge(`T, (x_{near}, x_{new})`)`
  if $x_{new} \in X_g$ **then**
    return `TrajectoryFromBranch(`T, x_{new}`)`
  end
end
return *null*

Algorithm Analysis

Nearest Neighbor Search The algorithm searches for the closest state to the sampled state among the nodes in the tree according to the metric ρ in every iteration of the algorithm and this search can have considerable impact on the performance of the algorithm.

A naïve solution would be to go over all tree nodes and find a node with minimum distance to the sampled state, but this approach would have linear time complexity with respect to the number of nodes of the tree. A better solution is to use a space partitioning data structure which is designed for nearest neighbor

search. A good choice is a balanced k-d tree [21], where k is the dimension of the partitioned space, in our case the dimension of X.

Time Complexity The algorithm will run for at most K iterations. During every iteration, a random state will be sampled, nearest node will be found, an action will be selected, and the new state will be inserted into the already explored states tree structure.

We can assume that selecting a random sample of the state space does not depend on the already explored parts of the state space and that this operation time complexity is constant.

The complexity of finding the nearest neighbor and inserting a new node into a k-d tree with a fixed number of dimensions is logarithmic with respect to the number of nodes in the tree on average, in our case this means $O(\log K)$.

Finding an action which contributes to the largest progress towards the goal will be implemented by evaluating all the $m = |U_f|$ options and measuring the distance between the resulting state and the target state. The time complexity of this operation is $O(m)$.

Checking if a state is a goal state is a constant operation because we only have to check if μx_{new} equals to the number of waypoints.

The time complexity of a single iteration of the algorithm is $O(\log K + m)$ and so the total time complexity of the algorithm is $O\left(K(\log K + m)\right)$.

Memory Complexity In order to capture the state of the algorithm, we need to keep the tree of all already explored states. As we discussed earlier, we use a k-d tree because of its nearest neighbor search queries. The memory complexity is linear with the number of nodes stored in the tree. In each iteration of the algorithm we add at most one new node to the tree which will lead to a tree with at most K nodes. The memory complexity of the whole algorithm is therefore $O(K)$.

Completeness All nodes added to the tree are added as leaves and an edge between two nodes in the tree always corresponds to application of some action $u \in U_f$. None of the nodes added to the tree corresponds of a state which would collide with an obstacle.

All branches of the tree T correspond to some feasible trajectory in the state space. The algorithm will stop as soon as the tree contains a branch which corresponds to a feasible solution trajectory or if the number of iterations exceeds the limit K.

RRT is proven to be probabilistically complete for K approaching infinity [22]. This means that if we sample the state space long enough and make our tree denser and denser, we are guaranteed to eventually find some solution. Because we chose the discrete-time model, we would have to also make the time discretization finer after each failure to achieve completeness.

Optimality The main difference between our modified algorithm and the original algorithm proposed by Steven LaValle is the early termination of the algorithm when the first solution is found [20]. We do not try to find a better solution and

we do not grow our tree in a way which would guarantee that the first solution we find will be an optimal solution.

RRT does not have any mechanism for finding optimal trajectories. On contrary, it is proven that the paths found by RRT are optimal with the probability of 0 [22].

There are modifications to the RRT algorithm which utilize the randomized sampling approach and probabilistic completeness of the algorithm, but which are optimal. One of these algorithms is called Optimal RRT (RRT*) [22]. Unfortunately, this algorithm is not easily usable for us, because it is not designed for problems with differential constraints. The algorithm uses a "rewiring" procedure to optimize the tree edges after a new node is added. This procedure needs to be able to connect two states $x_1, x_2 \in X$ with an action trajectory. This would be near impossible in the discrete-time model we chose and even with variable time, this operation would be very expensive and impractical.

Sampling Schema

The sampling schema is the most important part of the algorithm when it comes to the rate of convergence to a solution. The sampling function can also have an impact on the completeness of the algorithm.

Uniform sampling is the simplest form of random node selection. The value for each dimension of the state vector is selected uniformly from the set of valid values.

Goal biasing is a technique where we choose a state representing the final configuration with probability p and we uniformly sample the space otherwise. This method can be generalized for several goals which can then be useful to bias the exploration of the state space in the direction of several waypoints. The goal of this method is to drive the search towards the goal faster, but at the same time keep the probabilistic completeness of the algorithm, as uniform sampling can help avoid getting stuck in dead ends.

Path biasing is an extension of the goal biasing sampling scheme. We first find a reference path using a relaxed variant of the problem (e.g., a grid search algorithm without considering the differential constraints) and choosing the points along the path as intermediate goals for the grown tree. This variant of the algorithm is referred to as *RRT-Path* [23].

Conclusion

The RRT is a very interesting algorithm. One of its biggest advantages is its simplicity. The algorithm is easy to understand and easy to implement. There are many extensions of the original idea which improve the rate of conversion to the goal. The algorithm is complete, if we allow it to search for a solution long enough. On the other hand, the algorithm finds a trajectory without any guarantee of optimality. The optimal extension of the algorithm RRT* is hard to adapt to our use case, when we must consider the differential constraints of the vehicle.

3.3.2 Hybrid A*

Due to the infinite number of states reachable from an initial state $x_0 \in X$ through actions from the infinite set of actions U and the transition function $f_{\Delta t}$, it is impossible to exhaustively search for an optimal trajectory in the configuration space. Even if we limit the number of actions to a finite set of motion primitives U_f and limit the maximum length of a trajectory, uninformed search for a trajectory will be impractically slow. In order to converge to a goal faster, informed search algorithms estimate graph nodes based on knowledge of the nature of the searched space.

A* is a complete and optimal informed graph search algorithm [24]. It visits the most promising nodes of the graph first, but it will eventually visit all the reachable nodes, until it finds a path to a goal state, or until every node was visited. We will describe a modified version of the A* algorithm which reduces the searched state space even further by discretizing the original continuous state space into an n-dimensional grid while still producing feasible trajectories in the configuration space.

This modified algorithm is referred to as *Hybrid-State A* Search*. It was successfully used for path planning by the Stanford Racing Team in their vehicle *Junior* during the DARPA Urban Challenge in 2007 [25]. This algorithm was utilized for navigation in unstructured environments when it was not possible to follow road center line such as parking lots and for other complex maneuvers such as U-turns or overtaking of other vehicles. It has been explored and successfully used by several other researchers since then [26].

State Space Discretization

The advantage of A* is that it visits every state at most once and whenever we reach a graph node, we have an assurance that the path through which we came from the initial node is the best possible one and we can ignore this node in the future if we find a different route to it. This works well for discrete search spaces where it is easy to compare two nodes between themselves and determine whether it was already visited or not. In a continuous state space, this can be problematic. Even though we come very close to an already visited node, it does not have to be equal to the node. Let us consider the following example.

Example. Imagine we have a robot which moves in a two-dimensional plane and it is subject to differential constraints. The state space will be the configuration space $X = C$. We will focus on one action u which moves the robot forward and steers to the left. In a fixed time of $\Delta t = 1$ second the robot should travel around a quarter of a circle and turns by ninety degrees ($\frac{\pi}{2}$ radians). If we start from an initial state $x_0 = (0, 0, 0)$ then after applying u four times in a row, we would expect to go through states $x_1 = \left(1, 1, \frac{\pi}{2}\right), x_2 = (0, 2, \pi), x_3 = \left(-1, 1, \frac{3\pi}{2}\right)$, and arrive to a final state $x_4 = (0, 0, 0) = x_0$ after 4 seconds. If we try to implement and run this example in a simulation, we might get a slightly different result caused by accumulated rounding errors and other numerical approximations. For example, we might use only a finite number of the decimals of the number π and arrive to a heading angle of $\theta < 2\pi$. The A* algorithm will consider x_0 and x_4 as

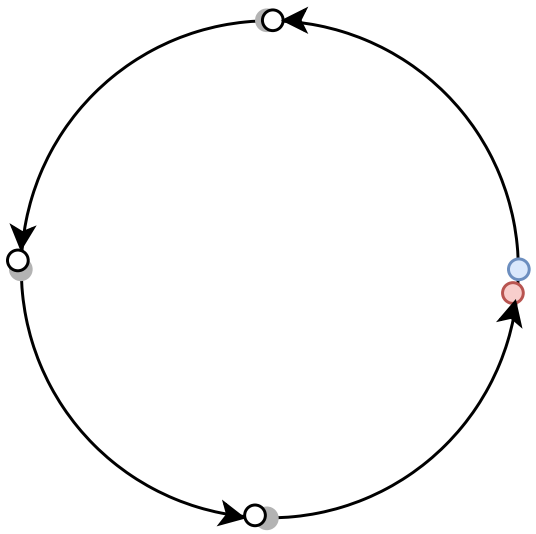


Figure 3.4: We would expect the car to return to its initial position after the action u was applied 4 times. Due to rounding errors and errors of numerical integration, we might reach a different state close to the initial state. The figure shows how the initial state, represented by the blue circle, gradually drifts from the expected position until it reaches the fourth state, represented by the red circle, which is different from the initial state.

different nodes of the graph and it will continue with the expansion of x_4. This example is visualized in Figure 3.4.

*Hybrid A** divides the state space into a grid of similar states to avoid this problem. Before we expand a state from the configuration space, we will check if the grid cell it falls into is already closed or not. If the cell is closed, we will consider the state to be closed as well and we will not expand it again.

We can select an appropriate resolution for every dimension of the space base on the semantics of each dimension. If we wanted to ignore some dimension of the state space for searching, we can set the resolution for the dimension to ∞. We can convert the continuous dimensions of a state vector x into a discrete grid coordinate with cell sizes vector $\boldsymbol{\sigma}$ with a state space discretization function $d : X \to X$:

$$d(x) = \left\lfloor \frac{x}{\boldsymbol{\sigma}} \right\rfloor .$$

The *hybrid A** algorithm stores the exact state through which we entered every visited discrete grid cell. The motion is continued by applying the action from the original continuous state, not the discretized one.

There is one problem with the discretization which is worth solving. Depending on the resolution of the grid, the next state obtained by the state transition function might fall into the same cell as the source state. This could be a significant problem especially at low speeds, for example when the vehicle is accelerating from a standstill. To solve this problem, we will repeat the same action k times, until the resulting state falls into a different cell. We will then store the resulting state for future expansion with an appropriate cost.

Heuristic Function

The A* algorithm is driven towards the goal by a heuristic function. Designing a good heuristic function can help us converge towards the goal faster by expanding the most promising nodes first. The heuristic function estimates the remaining cost to go from any state to the goal. Our cost function represents time it takes to drive from one state to another. Our heuristic function therefore must estimate the remaining time it will take to reach the goal from a given state.

To guarantee optimality of A* for graph searching, we require the heuristic to be admissible and monotonous. This means that along any direct path from the initial node and a goal node the value of the heuristic must not increase and therefore maintain a form of triangle inequality.

The inspiration for our heuristic functions were taken from the Stanford Racing Team entry in the DARPA Urban Challenge [25].

Euclidean Distance One of the simplest heuristics is calculating the distance between the position of the vehicle and the position of the goal. Our vehicle must drive past several waypoints. We must calculate the distance from the current position of the vehicle to the next waypoint and then add the distances between the remaining waypoints to obtain the minimum remaining distance d.

To calculate the time to reach the last waypoint, we must factor in the velocity of the vehicle. To maintain admissibility of the heuristic, we must assume that the vehicle will travel at maximum velocity. We will calculate the minimum time necessary by dividing the distance d by the maximum possible velocity of the vehicle v_{max}.

This heuristic is admissible and monotonous, because as we move closer to the goal along any path, the Euclidean distance decreases and our estimate is always lower or equal to the real capabilities of our vehicle.

We ignore the driving characteristics of our vehicle and we also do not account for collisions with any of the obstacles or the boundary of the racing track. On the other hand, the value of the heuristic is easy to calculate, and it does not need any additional memory.

Shortest Path Through Waypoints We discretize the two-dimensional plane into a grid with the same resolution as we do for the position dimensions of the state space. We will then compute the distance to the goal in an eight-connected grid using dynamic programming. We ignore grid cells which are outside of the racing track or the ones which overlap with some obstacle. Given a distance, we can calculate the minimum time to reach the goal in the same way as in the case of Euclidean distance heuristic.

This heuristic ignores the capabilities of the vehicle and the constraints on steering. Nevertheless, it will guide the search along the course of the racing track, and it will help us avoid dead ends.

Combination of Heuristic Functions To take advantage of several different heuristics $h_1, h_2, \ldots, h_n$, we can create a new heuristic, which takes the maximum estimated cost-to-go:

$$h(x) = \max_{i \in 1,2,\ldots,n} h_i(x).$$

This heuristic will dominate all the original heuristics and it will give us better estimates of the remaining cost. We will the combination of both the euclidean distance and shortest path heuristics as our final heuristic for the A* algorithm.

Search Records

We will conduct search over the state space of our vehicle. Besides the state vector, we need to remember additional information: the cost-to-come from the initial state to the current state and a pointer to the state from which we came to the current state. We will keep this information in records (x, g, v_p), where $x \in X$ is the state, $g \in \mathbb{R}$ is the cost-to-come, and v_p is the parent search record. The initial state does not have any parent search record. By traversing the search records, we can track back the states to the initial state and reconstruct the state trajectory which ends in any search record.

Cost-to-come The cost of a trajectory accumulates over time and it is non-negative. The cost-to-come for the initial state is 0. We fixed the time of execution of each action to Δt seconds. If we start in a state with cost-to-come g after applying an action $k \in \mathbb{N}$ times, the cost-to-come of the new state will be $g + k\Delta t$.

Total Cost Estimate

The sum of the cost-to-come and the estimate of the cost-to-go from the heuristic function gives us an estimate of a total cost from the initial state to a goal state. The states with a lower total cost estimate are considered *more promising* and we will explore these states first in order to speed up conversion to the goal region.

Algorithm Analysis

In this section we will analyze the pseudo-code shown in Algorithm 2. We will explain how it differs from the original version of the A* algorithm [24] and we will then also analyze the time and memory complexity of the algorithm based on our choice of data structures.

The main procedure of the algorithm `HybridAStar` is very similar to what one could expect from the A* algorithm. We initialize an *open set* O with the initial state search record and also an empty *closed set* C. We will say that a state is *opened* if it has a record in O and that it is *closed* when the cell which it falls into is in C.

In the main loop, we obtain the most promising state $x \in O$ by comparing the total cost estimate of the open states. If x reaches the goal region, we reconstruct the state trajectory by traversing the parent search records using a helper function `ReconstructTrajectory`. We test if the cell in which the state belongs has not been closed yet and only if it has not and the cost-to-come does not exceed the limit do we continue to expansion of the state. We close the cell into which the state x belongs so that we will not expand any further states from this cell.

The expansion of a state is implemented in the procedure `Expand`. We try to explore the outcome of every possible action in x. We simulate the application of each action using the function `Unfold`. If the application of the action did not result in a collision, we will add the final state x' to O if the cell $d(x')$, which

Algorithm 2: The Hybrid A* algorithm.

Input: Trajectory planning problem $(X, U_f, c, f_{\Delta t}, x_0, X_g)$, maximum trajectory cost limit g_{max}

Output: State trajectory or *null*

Parameter: State space discretization function $d : X \to X$, heuristic function h

Algorithm `HybridAStar`(d, h)**:**
  $O \leftarrow \{(x_0, 0, null)\}$
  $C \leftarrow \emptyset$
  while $O \neq \emptyset$ **do**
    $v \leftarrow \arg\min_{(x', g', v_p) \in O} g' + h(x')$
    $O \leftarrow O \setminus \{v\}$
    if $x \in X_g$ **then**
      return `ReconstructTrajectory`(v)
    end
    $(x, g, v_p) \leftarrow v$
    if $d(x) \notin C \wedge g < g_{max}$ **then**
      $C \leftarrow C \cup \{d(x)\}$
      `Expand`(v, O, C, d)
    end
  end
  return *null*

Procedure `Expand`(v, O, C, d)**:**
  foreach $u \in U_f(x)$ **do**
    $(x, g, v_p) \leftarrow v$
    $(x', k) \leftarrow$ `Unfold`(x, u, d)
    if $x' \neq null \wedge d(x') \notin C$ **then**
      $g' \leftarrow g + k\Delta t$
      $O \leftarrow O \cup \{(x', g', v)\}$
    end
  end

Procedure `Unfold`(x_{start}, u, d)**:**
  $(x_{last}, k) \leftarrow (x_{start}, 0)$
  repeat
    $x_{prev} \leftarrow x_{last}$
    $x_{last} \leftarrow f_{\Delta t}(x_{last}, u)$
    if $c(x_{last}) = 1$ **then return** $(null, -1)$
    $k \leftarrow k + 1$
  until $x_{last} = x_{prev} \vee d(x_{start}) \neq d(x_{last})$
  return (x_{last}, k)

was reached, has not been closed yet. In other versions of A*, it is common to check if the new state is currently in O. If x' already is in O, in case when x' was reached with a lower cost-to-come, the old g-value and the ancestor $p(x')$ would be replaced with the new values. We decided not to do this and instead keep both records in O. If any of these records for the same state are expanded before the algorithm finds a solution, its cell will be closed and so the consequent records of this state (with a higher cost-to-come), will be skipped when they are removed from the open set.

The simulation of the movement of the vehicle happens in the `Unfold` procedure. This function uses the system simulator function $f_{\Delta t}$ repeatedly until the vehicle stops (none of the properties of the vehicle state does not change after an action is applied), or until the vehicle leaves the cell of the parent state in the discretized state space.

Time Complexity

The time complexity of the A* algorithm depends mainly on the number of nodes which are expanded. We must also consider the complexity operations we perform whenever we expand a node. This involves an analysis of the way we implement the data structures for the open and closed nodes, which we vaguely expressed as set operations in the Algorithm 2.

The number of expanded states The maximum length of a path in the search space we will explore is limited by the cost limit $g_{max} \in \mathbb{R}$. This limit alone defines a continuous bounded region of states which are reachable from the initial state. There are two factors, which affect the number of nodes that can be expanded: the discretization of time into Δt second intervals and the discretization of the state space into a grid via the discretization resolution vector $\boldsymbol{\sigma}$. Both of these discretizations limit the number of states which are reachable. The total number of reachable states is then the minimum of these two limits.

Since our cost function is constant for all actions, we will only expand nodes at the ends of action sequences of at most $g_{max}/\Delta t$ steps. Due to the implementation of the `Unfold` procedure which might apply a single action multiple times in a row, the length of the sequence and therefore the number of expanded nodes can be even smaller. The expanded nodes form a tree with at most $l = \lceil g_{max}/\Delta t \rceil$ levels with a branching factor of $b = |U_f|$. Each of the states x will be expanded at most once, because if it were to be expanded the second time, its cell would already be closed ($d(x) \in C$). This means that in the worst case when there are no obstacles and each state would be in a different cell, the number of expanded nodes would be $O(b^l)$.

The number of reachable states in the discretized state space also limits the number of expanded states, as at most one state will be expanded per each discrete cell. If the state space has d dimensions and the maximum number of discrete cells in the individual dimensions is $\boldsymbol{n} = (n_1, n_2, \ldots, n_d) \in \mathbb{N}^d$, the upper limit for the number of cells is $\max(\boldsymbol{n})^d$.

Open set operations In order to keep track of which element should be expanded next, we need to keep the data in the open set arranged in a priority

queue by their total cost estimate $g + h(x)$. We use three operations of the priority queue: adding an element, and getting and removing an element with the minimum key. When implemented as a binary heap, all of these operations will have time complexity of $O(\log n)$, where n is the number of nodes in the queue[1]. This can be at most b^l and so the time complexity would be $O(\log b^l) = O(l \log b)$.

Closed set operations In order to keep track of all the cells which have been closed, we need a data structure with an efficient insertion and element retrieval operation. This can be achieved with a hash table with an amortized time complexity of $O(1)$.

Heuristic function Our heuristic function is implemented as a maximum of several heuristics, which can all be implemented as lookup tables and so the time complexity will be $O(1)$.

The `Unfold` procedure The time complexity of "unfolding" depends on the number of repetitions of the action, before the state leaves the original cell. Each evaluation of the system simulator function $f_{\Delta t}$ is a non-trivial operation, which includes numerical integration, but asymptotically this operation is still $O(1)$. The collision detection function performs a single query into the occupancy grid with inflated obstacles. This is a simple operation with the complexity of $O(1)$. We will assume that in most cases, the number of repetitions is low and it is bound by some small constant. We will assume that the `Unfold` procedure as a whole is constant.

The `Expand` procedure The expansion tries to apply all available actions and adds all collision-free extensions of the previous state to the open set, unless the cell is already closed. For each action, we call the `Unfold` procedure, check if an element is already in the closed set, and we may insert an element into the open set. This gives us a total time complexity of $O(bl \log(b))$.

Total time complexity For each expanded state we remove the minimum item from the heap and we run the `Expand` procedure. The total time complexity is therefore $O(b^l \cdot l \log b \cdot bl \log b) = O(l^2 b^{l+1} \log^2 b)$. The dominating factor is the exponential b^{l+1}. This is expected, because motion planning was shown to be PSPACE-hard [19, Chapter 6.5.1].

Memory Complexity

Each of the states requires only a constant amount of memory. The only additional information stored in memory are the lookup tables of the heuristic functions which require at most linear space in the size of the discretized search space, which is at most $O(b^l)$. The graph itself does not have to be explicitly held in memory as we traverse it through calling the system simulator. The memory complexity of the open set O (implemented as a priority queue) and the closed

[1]The complexity of insertion can be reduced to constant time with an advanced heap, such as a binomial or Fibonacci heap. The choice of the binary heap is motivated by the priority queue in the C++ STL, which implements a binary heap.

set C (implemented as a hash table) is linear with the number of items stored in the data structure. The number of open or closed nodes can be at most the number of all expanded states. The memory complexity of the algorithm in the worst case is therefore $O(b^l)$.

Conclusion

The *Hybrid A** algorithm is a good option for trajectory planning. The size of the state space is enormous and the heuristic function helps us to focus on the most promising directions first and only explore the seemingly bad trajectories if we run to a dead end. The algorithm is systematic and if a solution exists, it will eventually find it. The order in which the algorithm explores the state space gives us a guarantee that the first solution it will find will be the optimal solution. The selected discretization of time Δt and of the size of the cells of the state space $\boldsymbol{\sigma}$ can have an impact on the existence of a solution. In theory, by making the discretization finer, we are guaranteed to find a solution. In practice, we could fine-tune the size of the cells $\boldsymbol{\sigma}$, to balance the performance and the ability to find solutions during the first search in most cases.

3.3.3 Space Exploration Guided Heuristic Search (SEHS)

Chao Chen presents a different approach to the discretization of the state space. In his dissertation, he presents a novel Space Exploration Guided Heuristic Search (SEHS) algorithm and several extensions of this algorithm [27]. The author points out that for systems subject to nonholonomic constraints, uniform discretization is suboptimal. Instead, a guidance path consisting of overlapping circles with their centers on the circumferences of each other and with their radii based on the distance to the nearest obstacle from the start position to goal region is found. The path is then used as a heuristic to guide the search and the circles are used to discretize the position dimensions of the state space as a Voronoi diagram around the centers of these circles. This algorithm reduces the number of expanded nodes when compared to both RRT or *Hybrid A**, which is demonstrated by the author on a series of experiments. In this section, we will describe this search method more in depth.

Space Exploration (SE)

Space exploration is a pre-processing step of this planning algorithm. We are looking for the shortest path of overlapping circles in $\mathbb{R}^2$ form the initial state position to the goal location. The center of each circle lies on the circumference of the previous circle along the path. The radii of the circles must be within the closed interval $[r_{min}, r_{max}] \subset \mathbb{R}^+$ and each radius is chosen so that it is as large as possible, but none of the circles can overlap a cell with an obstacle in the occupancy grid map of the track. The path of circles does not give us only the shortest path from the initial state to a goal, but it also avoids paths where the vehicle would not fit due to its size.

The exploration step can be implemented with a basic A^* search starting at the initial state location. During the expansion step of A^*, we will create a fixed number of new circles $n \in \mathbb{N}$ evenly spaced on the circumference of the

expanded circle. The radius for each new circle will be calculated by measuring the distance to the closest obstacle and trimming this value to be within the interval of accepted radii. We will not open any circles with a smaller radius than r_{min}. The search will end once we expand a circle which includes some endpoint $\boldsymbol{x_{goal}} \in \mathbb{R}^2$. The cost function will be the length of the path measured by the distances between the centers of the circles, which is incidentally just the sum of the radii of the circles along the path. The heuristic function $h_{\boldsymbol{x_{goal}}}$ will be a simple euclidean distance between the center of a circle and the goal point $\boldsymbol{x_{goal}}$.

Our goal is defined by a sequence of waypoints. We will start from the position of the initial state and use the center of the first waypoint as a goal. From the final circle in this path, we will run the algorithm again with the center of the second waypoint as a goal. The result will be a path through all of the waypoints, which is exactly what we need.

The pseudo-code of space exploration is shown in Algorithm 3. The main method "stitches" together the path from paths between the individual waypoins, which are obtained using the `FindPathBetween` procedure. This procedure implements the space exploration algorithm itself. It is an implementation of the standard A* algorithm and it shares most of the logic with Algorithm 2. There are only a few aspects, which deserve additional commentary:

1. A point is considered to be *closed*, when it is an internal point of an already expanded circle. This is not a simple operation which could be implemented using a hash-table. We can assume that we would perform a check against every circle in C and so the complexity would be linear with the size of C.

2. The function `PointsOnCircumference` calculates n polar coordinates $p_i = (2\pi i/n, r)$, where r is the given radius, and converts them into Cartesian coordinates with respect to the given center of the circle.

3. We implement the `DistanceToClosestObstacle` function by generating a fixed number of rays around the starting point and using a ray-marching algorithm to find the distance along the ray at which we enter a cell which is occupied by an obstacle. The number of rays must be high enough so that the risk of missing an obstacle between two rays within the radius r_{max} is low. We assume that the obstacles marked in the map are fairly large and that their surfaces are smooth, without long spikes. This approach is inspired by the way a Light Detection And Ranging (LIDAR) works.

The path of circles can be further optimized to find a shorter path between the centers of the circles, while not reducing the clearance from obstacles. The space exploration algorithm might not find the shortest path, because it considers only a fixed number of directions when a circle is expanded. The author of the algorithm proposes a simple iterative algorithm which adjusts the center of each circle $(\boldsymbol{x_i}, r_i)$ to be on the line between $\boldsymbol{x_{i-1}}$ and $\boldsymbol{x_{i+1}}$ in the ratio of $r_{i-1} : r_{i+1}$, if the distance to the nearest obstacle from this point is greater or equal to r_i. This process is repeated until some of the circles can be adjusted. At the end of these adjustments, the centers of the circles might not be on the circumferences of other circles, but the circles will still be overlapping. The single optimization step is visualized in Figure 3.5 and further details of this process are described in [27, Section 2.2.3].

Algorithm 3: The Space Exploration algorithm.

Input: Occupancy grid $O_r^{m\times n}$, initial position $\boldsymbol{x_0} \in \mathbb{R}^2$, list of waypoints $\hat{w} = \langle \boldsymbol{w_1}, \boldsymbol{w_2}, \ldots, \boldsymbol{w_l} \rangle \in \mathbb{R}^l$, range of valid radii $[r_{min}, r_{max}] \subset \mathbb{R}_{>0}$, number of child circles $n \in \mathbb{N}$.

Output: Path of circles or *null*

Algorithm `SpaceExploration`($O_r^{m\times n}$, $\boldsymbol{x_0}$, $\hat{w}$, $[r_{min}, r_{max}] \subset \mathbb{R}_{>0}$, $n \in \mathbb{N}$)**:**
- $\boldsymbol{x_{start}} \leftarrow \boldsymbol{x_0}$
- **for** $i = 1 \ldots l$ **do**
  - $c_i \leftarrow$ `FindPathBetween`($\boldsymbol{x_{start}}$, $\boldsymbol{w_i}$)
  - **if** $c_i = null$ **then return** *null*
  - $\boldsymbol{x_{start}} \leftarrow$ center of last circle of c_i
- **end**
- **return** $c_1 + c_2 + \ldots + c_l$

Procedure `FindPathBetween`($\boldsymbol{x_{start}}$, $\boldsymbol{x_{goal}}$)**:**
- $r_{start} \leftarrow$ `DistanceToClosestObstacle`($O_r^{m\times n}$, $\boldsymbol{x_{start}}$, r_{max})
- $O \leftarrow \{(\boldsymbol{x_{start}}, r_{start}, 0, null)\}$
- $C \leftarrow \emptyset$
- **while** $O \neq \emptyset$ **do**
  - $v \leftarrow \arg\min_{(\boldsymbol{x'}, r', g', v_p) \in O} g' + h_{\boldsymbol{x_{goal}}}(\boldsymbol{x'})$
  - $O \leftarrow O \setminus \{v\}$
  - $(\boldsymbol{x}, r, g, v_p) \leftarrow v$
  - **if** $\|\boldsymbol{x} - \boldsymbol{x_{goal}}\| < r$ **then**
    - **return** `ReconstructTrajectory`(v)
  - **end**
  - **if** $\forall\, (\boldsymbol{x'}, r') \in C : \|\boldsymbol{x} - \boldsymbol{x'}\| \geq r'$ **then**
    - $C \leftarrow C \cup \{(\boldsymbol{x}, r)\}$
    - `Expand`(v, O, C)
  - **end**
- **end**
- **return** *null*

Procedure `Expand`(v, O, C)**:**
- $(\boldsymbol{x}, r, g, v_p) \leftarrow v$
- **foreach** $\boldsymbol{x'} \in$ `PointsOnCircumference`(n, $\boldsymbol{x}$, r) **do**
  - $r' \leftarrow$ `DistanceToClosestObstacle`($O_r^{m\times n}$, $\boldsymbol{x'}$, r_{max})
  - **if** $r' \geq r_{min} \wedge \forall\, (\boldsymbol{x''}, r'') \in C : \|\boldsymbol{x'} - \boldsymbol{x''}\| \geq r''$ **then**
    - $g' \leftarrow g + r$
    - $O \leftarrow O \cup \{(\boldsymbol{x'}, r', g', v)\}$
  - **end**
- **end**

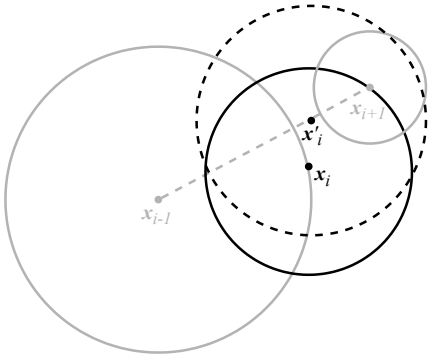


Figure 3.5: The optimization step of the path of circles moves the center of a circle only if the length of the path through the centers of the circles decreases and the radius of the circle can be increased or it stays the same. The center of the new circle might not be placed on a circumference of another circle.

Heuristic Search (HS)

The second phase of the SEHS algorithm is identical to *Hybrid A** as we defined it in Algorithm 2. We must only define a state space discretization function $d_{SEHS} : X \to X$ and a heuristic function $h_{SEHS} : X \to \mathbb{R}$. Both of these parameters will be derived from the path of circles obtained from the pre-processing phase.

State Space Discretization We will use the centers of the circles $X_c = \{\boldsymbol{c_1}, \boldsymbol{c_2}, \ldots, \boldsymbol{c_k}\}, \boldsymbol{c_i} \in \mathbb{R}^2$ obtained from the pre-processing to define the discretization of the x and y dimensions of the configuration space C. Any point $\boldsymbol{x} \in \mathbb{R}^2$ will be projected on the closest circle in X_c. The function $d_{x,y} : \mathbb{R}^2 \times X_c$ effectively defines the Voronoi diagram:

$$i(\boldsymbol{x}) = \underset{j \in \{1,2,\ldots,k\}}{\arg\min} \|\boldsymbol{x} - \boldsymbol{c_j}\|$$
$$d_{x,y}(\boldsymbol{x}) = c_{i(x)}.$$

Determining which center of a circle $\boldsymbol{c_i}$ is closest to a point $\boldsymbol{x}$ is an instance of a nearest-neighbor problem, which we already discussed when we described the RRT algorithm. It turned out that simple linear search is more efficient than a space-partitioning data structure when we ran the algorithm. This is most likely due to the fact that the number of circles k was usually low and that the array of all waypoints can be loaded into the cache of a processor and it does not require any expensive branching operations, unlike a search in a tree.

The path of circles does not give us any information which could be used to discretize the heading angle of the vehicle θ or any other additional state variables. For these state variables, we have to use the same uniform cell discretization as in *Hybrid A**.

This space discretization has one big advantage over the uniform grid discretization. When the path goes through a long wide corridor, it will consist of a small number of circles with large radii. Going through this part of the track will not require too many control inputs. When the path goes through a narrow passage between obstacles or through a corner, the radii of the circles will be smaller and so the number of circles in this area will be larger. Going through

this region will probably require finer control over the vehicle. We will be able to explore more different control input sequences through this area because the discretization will be denser here.

Heuristic The heuristic function estimates the remaining time which it could take to reach the goal. As in *Hybrid A**, we will estimate the remaining distance from the current state to the goal region and divide this distance by the maximum possible velocity of the vehicle v_{max}. For some point $\boldsymbol{x} \in \mathbb{R}^2$ we will find the circle closest to it and add the distance to the center of the next circle and the sum of the distances between the rest of the circles until the end region:

$$m(\boldsymbol{x}) = \left\| \boldsymbol{x} - \boldsymbol{c_{i(x)+1}} \right\| + \sum_{j=i(\boldsymbol{x})+1}^{k-1} \|\boldsymbol{c_j} - \boldsymbol{c_{j+1}}\|$$

$$h(\boldsymbol{x}) = \frac{m(\boldsymbol{x})}{v_{max}}.$$

Conclusion

The SEHS algorithm is a simple yet efficient way of searching a state space for a car-like vehicle. It promises significantly faster convergence towards a solution than *Hybrid A**, but the implementation itself is very similar. The main difference is in the discretization of the state space, which is much denser when the desired path is near obstacles and less dense when there are no obstacles around. We will implement both *Hybrid A** and SEHS and compare the results of these two algorithms.

3.4 Track Segmentation

A natural way of splitting the track into smaller segments is to find the corners of the track. We can then plan the trajectory for the very next turn in front of the vehicle and for one or two consecutive ones, and imitate the behavior of a human racing driver, as we described it in Section 2.1. To prevent long segments for long straight stretches, we can set a limit for the length of a segment and split long segments into multiple shorter ones.

Another benefit of splitting the track into smaller segments and planning the trajectory just for a fixed number of them is that the total length of the track does not affect the performance of the algorithm anymore. The length of the segments is limited and so the actual time it will take to calculate a trajectory for the next n segments on real hardware should be similar for different parts of the track. We will have to test this hypothesis experimentally.

In this section we will describe an algorithm we chose to find points close to corners of a racing circuit. We will use this algorithm just once before a race starts.

The definition of a racing circuit as an occupancy grid $O_r^{m \times n}$, initial configuration of the vehicle $c_0 \in C$, and a sequence of at least two checkpoints $\boldsymbol{p_i} \in \mathbb{R}^2$ which defines the direction in which the vehicle has to drive along the circuit. If the circuit is more complicated and there are multiple ways of completing the circuit, for example when the track intersects itself at some point (e.g., a circuit

in the shape of the number eight as shown in Figure 3.6b), it might be necessary to define more than two waypoints to force correct completion of the circuit.

The question is, how do we find a corner of a track in an occupancy grid? We can make a simple observation and derive an algorithm which will produce good approximate solutions.

Intuitively, a corner is point where the track bends and we are not able to keep driving in a straight line anymore. We can relax our problem for a moment and imagine, that robot has a differential drive. This allows it to move straight in the direction where it is heading, then stop, rotate around its vertical axis, and then start moving in the new direction. If we found a collision-free path for this robot in the occupancy grid through all the checkpoints of the track, the pivot points of the robot would be in the vicinity of the corners. It might be possible that it would require several turns within a short distance of each other to go around the corner, (e.g., a so-called "hairpin" corner as shown in the bottom left corner of Figure 3.6c) and so not every change of direction could be simply interpreted as a corner and some of them should be merged.

We will implement an algorithm based on this thought experiment which will have three steps:

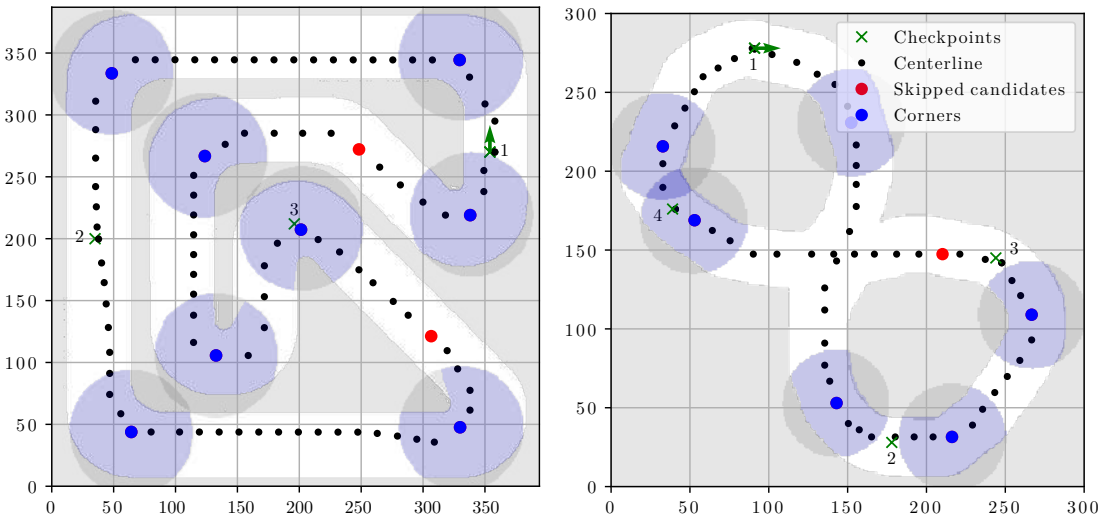
- We will find the pivot points in a path for the differential drive robot through the occupancy grid which starts at the initial position of the vehicle and which goes through the checkpoints.
- We will eliminate pivot points, where the direction does not change significantly.
- We will find clusters of the remaining pivot points along the path and keep only one of them, this point will be one of the final waypoints.

3.4.1 Finding Pivot Points

The shortest path can be found in several different ways. We can reuse the *Space Exploration* algorithm which was described in Section 3.3.3. The final path will consists of steps where the robot picks one of the fixed number of possible directions, and it will move forward in this direction, before it picks another direction. If we set the minimum radius of the circles to the half of the track width, then we will find a line resembling the center line of the track. Because the width of the track can vary along the circuit, we allow the radii of the circles to be between 60 % and 200 % of half of the width of the track at the starting position.

The outcome of the `FindPivotPoints` algorithm, the pseudo-code of which is shown in Algorithm 4, is a sequence of pivot points such that two adjacent points in the sequence are *directly visible* and the distances between these points are as long as possible. We use the `SpaceExploration` algorithm to find the "center line" $\hat{q}_l$ consisting of l points. We then go around the track and we try to merge these steps into longer straight lines.

We start from the initial position and pretend that this is the last marked pivot point. We find the furthest point along $\hat{q}_l$, which is directly visible from the last waypoint, and mark it as an index of the next waypoint. We then repeat this process from next waypoint until we reach the end of the track.



(a) Circuit “Race Track”

(b) Circuit “Eight”

(c) Circuit “Tornado”

(d) Circuit “Zurich”

Figure 3.6: This figure shows several different tracks on which we applied the track segmentation algorithm. The tracks have different layouts and feature different density and types of corners. The numbered green × marks show the checkpoints which define the circuit. The blue circles represent the detected corners.

Algorithm 4: Find Pivot Points

Input: Occupancy grid $O_r^{m\times n}$, initial state x_0, directly visible points relation D, sequence of checkpoints $\hat{p}_k = \langle \boldsymbol{p_1}, \boldsymbol{p_2}, \ldots, \boldsymbol{p_k} \rangle$

Output: Sequence of pivot points

Algorithm `FindPivotPoints`($O_r^{m\times n}$, x_0, $\hat{p}_k$, r_{track})**:**
  $\hat{q} \leftarrow$ `FindCenterline`($O_r^{m\times n}$, x_0, $\hat{p}_k$, r_{track})
  $W \leftarrow \emptyset$
  $i_{last} \leftarrow 1$
  foreach $i \in \{2, 3, \ldots, |\hat{q}|\}$ **do**
    $(W, i_{last}) \leftarrow$ `ProcessNext`(W, i_{last}, i)
  end
  return `SelectSubsequence`($\hat{q}$, W)

Procedure `FindCenterline`($O_r^{m\times n}$, x_0, $\hat{p}_k$, r_{track})**:**
  return `SpaceExploration`($O_r^{m\times n}$, $\kappa_{x,y}(x_0)$, $\hat{p}_k$, $[0.6r_{track}, 2r_{track}]$)

Procedure `ProcessNext`(W, i_{last}, i)**:**
  if $(\boldsymbol{q_{i_{last}}}, \boldsymbol{q_i}) \notin D(O_r^{m\times n})$ **then**
    $i_{last} \leftarrow i - 1$
    $W \leftarrow W \cup \{i_{last}\}$
  end
  return (W, i_{last})

Procedure `SelectSubsequence`($\hat{q}$, I)**:**
  $\hat{s} \leftarrow \langle\rangle$
  foreach $i \in \{1, 2, 3, \ldots, |\hat{q}|\}$ **do**
    if $i \in I$ **then**
      $\hat{s} \leftarrow \hat{s} + \langle \hat{q}_i \rangle$
    end
  end
  return $\hat{s}$

No Obstacles Edge Case In the case, when there are no obstacles in the occupancy grid, all points will be visible from each other and the set W will be empty and consequently the list of pivot points would be empty. We can ignore this edge case and assume that the racing circuit has a non-trivial layout with at least two corners (a narrow oval).

3.4.2 Estimating Positions of Corners

Once we have a sequence of pivot points, we can estimate the positions of corners based on these pivot points. Whenever the track curves, we expect a pivot point in the vicinity.

At some pivot points, the direction of the robot changes only very little. This means, that the angle between the previous point, the current point, and the next point along the path, will be close to π. The first pass of our algorithm removes pivot points, at which the angle is above some angle threshold value α_{max}. To calculate the angle between three consecutive points along the path $\boldsymbol{p_{i-1}}, \boldsymbol{p_i}, \boldsymbol{p_{i+1}} \in \mathbb{R}^2$, we use the law of cosines to calculate the angle between the vectors $\boldsymbol{x} = \boldsymbol{p_{i-1}} - \boldsymbol{p_i}$ and $\boldsymbol{y} = \boldsymbol{p_{i+1}} - \boldsymbol{p_i}$:

$$\alpha(\boldsymbol{x}, \boldsymbol{y}) = \arccos\left(\frac{\boldsymbol{x} \cdot \boldsymbol{y}}{\|\boldsymbol{x}\| \, \|\boldsymbol{y}\|}\right),$$

where $\cdot$ represents the scalar product between the two vectors.

It might be possible, that the `FindPivotPoints` algorithm returns a path, where there are several pivot points close to each other. We will try to merge these close pivot points into a single point, to reduce the number of very short segments. This complete algorithm is summarized as pseudocode in Algorithm 5.

Algorithm Analysis

The algorithm will always be complete and return a sequence of points such that for any of two consecutive points the distance between these points is at least d_{min}.

If during an iteration in `MergeClosePoints` no two consecutive points which are closer than d_{min} are found in $\hat{p}$, the algorithm stops. If at least one cluster of points is merged, then at least two points from the original sequence are removed and only one point is added. In every iteration the length of the sequence therefore decreases. After at most $k - 1 = |\hat{p}| - 1$ iterations, the number of points in the sequence will drop to a single point, and the algorithm will stop.

In every iteration, we find the biggest cluster of points along the sequence and we find the indices of these close points. These points form a consecutive sub-sequence of the original sequence $\hat{p}$. We remove all of these points, except for the point with the most significant change of direction or in other words, the point with the most acute angle along the path.

Time Complexity At the beginning of the algorithm, we iterate over all k pivot points and remove the points with large obtuse angles. Calculating the angle is a constant operation and so this part requires $O(k)$ time.

Algorithm 5: Find Corners

Input: Sequence of points $\hat{p}_k = \langle \boldsymbol{p_1}, \boldsymbol{p_2}, \ldots, \boldsymbol{p_k} \rangle$, minimum distance between corners $d_{min} \in \mathbb{R}_{>0}$

Output: Sequence of locations of corners

▷ `We will use the notation` $\boldsymbol{p_{i-1}}$ `and` $\boldsymbol{p_{i+1}}$ `to reference the previous and next point of the` i`-th point in the sequence. The sequence of points represents a loop. The previous point for` $\boldsymbol{p_1}$ `is the very last point` $\boldsymbol{p_k}$ `and the next point after` $\boldsymbol{p_k}$ `is` $\boldsymbol{p_1}$`.`

Algorithm `FindCorners`$(\hat{p},\ \alpha_{max},\ d_{min})$**:**
  $R \leftarrow \{i \in \{1, \ldots, |\hat{p}|\} \mid \alpha(\boldsymbol{p_{i-1}} - \boldsymbol{p_i}, \boldsymbol{p_{i+1}} - \boldsymbol{p_i}) \leq \alpha_{max}\}$
  $\hat{p}' \leftarrow$ `SelectSubsequence`$(\hat{p},\ R)$
  return `MergeClosePoints`$(\hat{p}',\ d_{min})$

Procedure `MergeClosePoints`$(\hat{p},\ d_{min})$**:**
  $\hat{p}' \leftarrow \hat{p}$
  repeat
    $\hat{p} \leftarrow \hat{p}', k \leftarrow |\hat{p}|$
    $i_{max} \leftarrow \arg\max_{i \in \{1,\ldots,k\}} |$`Neighbourhood`$(i,\ \hat{p},\ d_{min})|$
    $N \leftarrow$ `Neighbourhood`$(i_{max},\ \hat{p},\ d_{min})$
    $i_{corner} \leftarrow \arg\min_{i \in N} \alpha(\boldsymbol{p_{i-1}} - \boldsymbol{p_i}, \boldsymbol{p_{i+1}} - \boldsymbol{p_i})$
    $\hat{p}' \leftarrow$ `SelectSubsequence`$(\hat{p},\ \{1, \ldots, k\} \setminus N \cup \{i_{corner}\})$
  until $\hat{p}' \neq \hat{p}$
  return $\hat{p}$

Procedure `Neighbourhood`$(i,\ \hat{p},\ d_{min})$**:**
  $N_{ahead} \leftarrow$ `Walk`$(i,\ \hat{p},\ d_{min},\ i \mapsto i-1)$
  $N_{after} \leftarrow$ `Walk`$(i,\ \hat{p},\ d_{min},\ i \mapsto i+1)$
  return $N_{ahead} \cup \{i\} \cup N_{after}$

Procedure `Walk`$(i,\ \hat{p},\ d_{min},\ f_{step})$**:**
  $N \leftarrow \emptyset$
  $j \leftarrow f_{step}(i)$
  while $\|\boldsymbol{p_i} - \boldsymbol{p_j}\| < d_{min} \wedge j \neq i$ **do**
    $N \leftarrow N \cup \{j\}$
    $j \leftarrow f_{step}(j)$
  end
  return N

As we discussed earlier in this section, the main loop in `MergeClosePoints` will run at most $k - 1 = |\hat{p}|$ times. In each iteration, we first find the point with the highest number of close neighbors and then merge this cluster into a single point. Finding the point with the highest number of neighbors is a quadratic operation‘.

The total time complexity of the proposed algorithm is quadratic in the length of the size of the input sequence $O(k^3)$. In practice, we expect the number of pivot points k to be low, and so even its third power will be very low. Also, the algorithm is used only once at the very beginning of the race. This algorithm is sufficient for this task and its time complexity is insignificant when compared to `FindPivotPoints`.

Memory Complexity The only thing our algorithm has to remember is the input sequence of points. This requires $O(k)$ of memory. The total memory complexity is therefore linear.

3.5 Differential Constraints

In order to make the trajectory planning possible at all, we must be able to predict the behavior of the vehicle based on its state and the actions we execute. Without a good understanding of how the state of the vehicle evolves over time, we might collide with an obstacle, plan a sub-optimal trajectory, or even plan a trajectory which cannot be followed by the physical vehicle.

Car-like robots are typically underactuated. The configuration space C has three degrees of freedom: the x, y position and the heading angle θ. On the other hand, the actions which control the vehicle, have only two degrees of freedom: a steering angle of the front wheels and the motor throttle. The vehicle moves only in the direction in which it is pointing and it changes the heading direction only while it is moving. Systems with such limited directions of motion are called non-holonomic in literature.

In this chapter, the task is to describe all the properties which define the state of the vehicle at any given moment and model the rate of change of the state when a specific input command is supplied to the actuators. In other words, we must describe a state transition function f which calculates a derivative of the vehicle state with respect to time $\dot{x} = f(x, u)$. By integrating $\dot{x}$ over time, we will be able to predict how the state evolves and describe the trajectory of the vehicle.

The state of our vehicle consists of several parts:

- the configuration of the vehicle $(x, y, \theta) \in C$,
- the state of the actuators,
- the state variables of the kinematics and dynamics of the vehicle.

Actuators model The vehicle we will be modeling has two actuators which influence the motion of the vehicle: a steering servo and a brushed DC motor. These actuators can be controlled by setting a target value of the steering angle δ_t and the throttle position τ_t, which affects the speed of the motor. In general, the

vehicle can have more actuators and other mechanical components, which would have to be modeled and whose state we would have to capture. These could be an internal combustion engine, brakes, or a clutch and a gearbox.

Kinematic models are a simple way of modeling the behavior of a vehicle. All the different forces acting on the body of the vehicle are ignored, and we rather calculate the movement only based on the geometry of the vehicle and the velocity of the vehicle. This simplification yields accurate predictions for low speeds when the effects of inertia are negligible, but the error between the predictions and the actual movement of the vehicle will grow larger as the speed of the vehicle increases and the tires start to slip.

Dynamic models on the other hand describe the forces which act on the vehicle and which result in translation and rotation of the body of the vehicle. We model the effect of the torque created by the engine which is then transmitted through the power train to the wheels. The friction force between the rotating tires and the road causes the vehicle to accelerate or decelerate and at the same time it holds the vehicle on a curve while turning. At a certain speed and wheel angles the lateral force which keeps the vehicle on the curve will not be enough to keep the vehicle on the curve and the car will start drifting sideways.

The dynamic model can give us more accurate predictions at the handling limits of the vehicle. As a trade-off, the dynamic model is more complex and harder to implement correctly. We must identify many parameters of the vehicle, such as the properties of the tires and the characteristics of the power train. The dynamic model also requires more arithmetic operations to predict the next state after some period Δt which can make planning with this model slower.

3.5.1 Actuators Model

In this section, we will describe the behavior of the actuators of our vehicle and we will propose a mathematical model of the state of these actuators. The model is tailored to the experimental vehicle which we used and which is described in more detail in Chapter A.

Steering Servo Model

Steering Angle to PWM Mapping The position of the steering servo is transferred to the front wheels through mechanical linkages. The inner wheel is steered more than the outer wheel, because the outer wheel will circumscribe a circle with a larger radius than the inner wheel. When the wheels are rolling perfectly against the surface at slow speed, the vehicle will keep going around a circle with a constant radius. PWM is the electrical impulse we use to control the steering servo. More information about this signal is available in Appendix A.1 and Figure A.2.

We measured the radius R of the inner wheel as it circumscribes a circle, with the steering servo set to several constant PWM values. From this radius, we can calculate the steering angle δ of a virtual front wheel in the middle of the front axis from the geometry of the vehicle:

$$\delta = \arctan\left(\frac{L}{R + W/2}\right), \tag{3.4}$$

where L is the length of the wheelbase and W is the width of the axles. The geometry is captured in Figure 3.7. More information about Ackermann steering geometry can be found in [28, Chapter 2].

Using the least squares method, we identified a line which approximates the relationship between a steering angle and the PWM signal which corresponds to this angle:

$$PWM(\delta) = 12.701 \cdot \delta + 1476.686, \tag{3.5}$$

where $\delta \in [-21.16°, 26.57°]$ is the steering angle expressed in degrees and $-21.16°$ is the maximum steering angle to the right and $26.57°$ is the maximum steering angle to the left. The relationship between the steering angle and the PWM value is shown in Figure 3.8. This measurement was in part inspired by the experiments in [13] and [29].

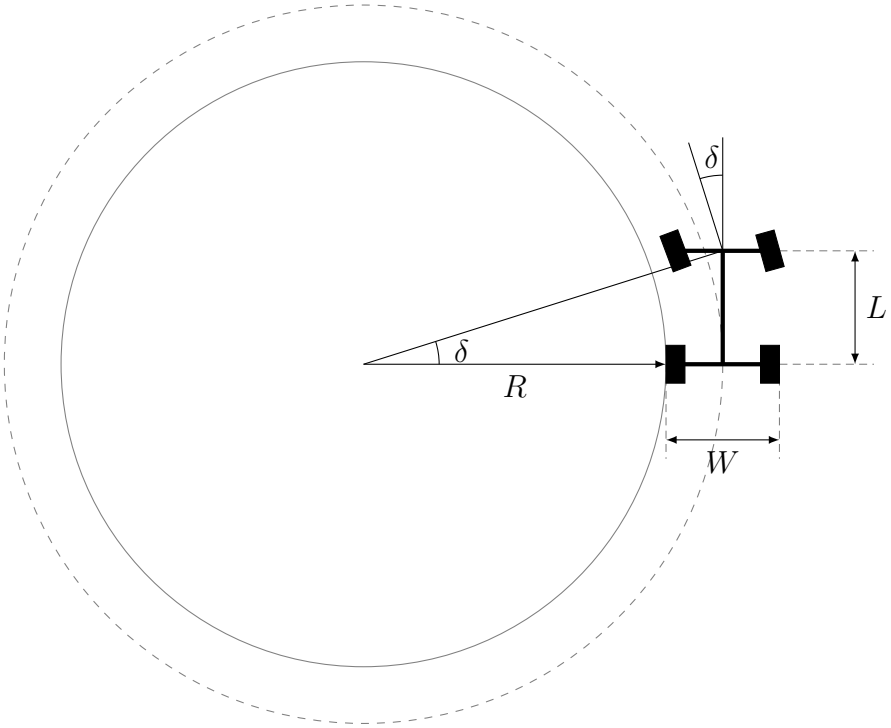


Figure 3.7: For a fixed steering angle δ of the virtual front center wheel, the vehicle will move along a circle of radius R, unless the wheels start skidding. The proportions of the vehicle in the picture are the same as on the experimental vehicle, where $W = L = 0.3\,\mathrm{m}$.

Servo Setting Time It takes non-trivial time for the steering servo to move from its previous position to a newly set position. The steering angle changes continuously as the servo adjusts and it affects the direction in which the vehicle travels in the meantime. In this experiment, we measure the setting time of the servo between different steering angles and based on the data, we will find a

relationship between the distance between the servo position and the time it will take to set.

The position of the servo is set through a PWM value between 1200 µs (rightmost position), 1500 µs (center position), and 1800 µs (leftmost position). In this procedure, the vehicle switches between a right position r and a left position l a hundred times while the vehicle is stationary and it is placed on a flat surface. The servo produces a noise during the whole setting period and is mostly quiet during the periods between adjustments.

We record the sound the servo makes through a microphone. The audio track is then edited using *Audacity*[2] to remove background noise and low frequency "buzzing" of the servo and to use the "Sound Finder" analysis tool to label periods of sound in the cleaned audio track (see Figure 3.9). The start and end times of the labeled intervals are then exported into a text file for further analysis.

We carried out this procedure for five pairs of PWM values: $(1800, 1200)$, $(1750, 1250)$, $(1700, 1300)$, $(1650, 1450)$, $(1600, 1400)$. Each of these pairs is symmetrical around the center position 1500 µs and the *distance* between the two positions decreases from 600 µs to 200 µs. This might have biased the results and decrease the accuracy of our result.

We used the method of least squares to find a best fitting line in the form of $t = ad + b$, where t is the setting time in seconds and d is the distance between the PWM values. Based on the data we recorded, the relationship between the distance to the new servo position and the setting time is visualized in the Figure 3.10 and in the following equation:

$$\begin{aligned} t(d) &= 0.000329 \cdot d + 0.1174 \\ t(\alpha, \beta) &= t(|PWM(\alpha) - PWM(\beta)|), \end{aligned} \tag{3.6}$$

where $\alpha, \beta \in [\delta_{min}, \delta_{max}]$.

Steering Angle Model With the knowledge of how long it takes to adjust the steering angle of the front wheels, we can formulate the rate of change of the steering angle $\dot{\delta}$ in radians per second based on the current steering angle δ and target steering angle δ_t:

$$\begin{aligned} \Delta\delta &= \min\left\{|\delta - \delta_t|, 2\pi - |\delta - \delta_t|\right\} \\ \dot{\delta} &= {}^{\operatorname{sgn}\Delta\delta}/_{t(\delta,\delta_t)}. \end{aligned} \tag{3.7}$$

DC Motor Model

We need to be able to model the angular velocity of the motor shaft. The motor is directly connected to the wheels. Therefore, if we can predict the velocity at which the motor shaft rotates, we can predict how fast the wheels will turn. If the motor spins at angular velocity of ω, then the angular velocity of the wheels ω_w will be equal to

$$\omega_w = G\omega,$$

[2] `https://www.audacityteam.org/`

where G is a combined gear ratio of the gearbox and the differential. For a vehicle with multiple gears, this relationship would be a function of the selected gear. Because our experimental vehicle will have a single fixed gear, which is typical for electric cars, our gear ratio G will be constant.

This, of course, does not give us all the information to determine the longitudinal velocity of the vehicle, because the wheels might skid when the friction forces between the tires and the road surface are exceeded.

Our goal is to create a model of the engine RPM as a function of the previous RPM and the control inputs. We measured the RPM of the motor shaft using a Hall effect encoder while we were manually controlling the vehicle on dry asphalt at a temperature of around 15 °C. We also recorded the throttle and steering angle inputs from the remote control. We use the collected data to identify the parameters of our model and to validate the accuracy of the model.

The torque of the motor depends directly on the speed at which its axle rotates. For a DC motor, the relationship can be approximated with a linear function which depends on two parameters: the maximum angular velocity ω_{max} and stall torque T_{stall} as it is shown in Figure 3.11. The maximum angular velocity is measured with full throttle when there is no load on the motor and the stall torque is reached when the vehicle starts moving from standstill.

The throttle control input τ_t limits the amount of torque transferred to the wheels. The torque is counteracted by the load on the motor, e.g., friction or air resistance. While driving the experimental car manually, we learned that the steering angle δ of the front wheels is one of the most significant factors limiting the speed of the motor. When the car is at its maximum speed and the throttle is released, the car will travel a significant distance before it comes to a full stop. When the steering angle is large, the car stops almost immediately. The vehicle reaches a lower speed when it travels around a circle compared to going straight when full throttle is applied. The rate of change of angular velocity of a rigid body is equal to the total torque applied to the body divided by its moment of inertia.

We model the rate of change of the angular velocity of the motor with the following set of parametric equations, which are inspired by the mechanics of the motor:

$$\begin{aligned} T_{max} &= \left(1 - \frac{\omega}{\omega_{max}}\right) x_1 \\ T_{drive} &= T_{max} \cdot \tau_t \\ T_{load} &= \omega^{x_2} \left(x_3 + x_4 |\delta_t|\right)^{x_5} \\ \dot{\omega} &= \frac{T_{drive} - T_{load}}{x_6}. \end{aligned} \tag{3.8}$$

By integrating the $\dot{\omega}$ numerically over a time period Δt, we will obtain the change in the RPM of the vehicle. The parameters $x_i \in \mathbb{R}$ are obtained using a numerical optimization algorithm which minimizes mean squared error between the predicted RPM and RPM measured using the experimental vehicle. The parameter x_1 replaces T_{stall} and x_6 replaces the moment of inertia of the power train. Nevertheless, the values assigned to these parameters by the optimization algorithm must not be interpreted as the values of these two properties.

An example prediction of our model with the fitted parameters as stated in

Table 3.1 is shown in Figure 3.12 for a time period of almost two minutes. The RPM shown in this figure is relative to the maximum RPM of the motor which we identified to be 15 500 RPM.

x_1	$7.330\,167\,01 \times 10^{2}$
x_2	$8.586\,268\,96 \times 10^{2}$
x_3	$7.407\,399\,69 \times 10^{-1}$
x_4	$7.682\,488\,46 \times 10^{1}$
x_5	$2.051\,903\,02 \times 10^{2}$
x_6	$1.165\,842\,76$

Table 3.1: Fitted parameters of the DC motor model.

We also tried training a neural network with three input neurons and one output neuron to predict $\dot{\omega}$ or ω directly. We experimented with supervised learning and with reinforcement learning using neuro-evolution (NEAT) but we were not able to obtain a neural network with better estimates of ω than the predictions from the model described in (3.8).

3.5.2 Kinematic Bicycle Model

The name of the bicycle model, sometimes also referred to as one-track or single-track model in literature, comes from the fact, that we combine the effects of the two front wheels and the two rear wheels into a single virtual wheel located at the centers of the axles. Our kinematic model is based on a model described by Dr. Rajesh Rajamani [28, Chapter 2] and it is visualized in Figure 3.13.

We assume that the wheels are rolling in the same direction in which they are pointing. When the front wheels are turned to a steering angle $\delta \neq 0$, our vehicle will be going around a circle with a constant radius. We will also assume that the wheels are rolling against the road surface and that the vehicle never loses traction.

These simplifications will cause our model to be imprecise during high speed cornering, when the car can start "drifting" when the wheels lose grip and the vehicle will start moving sideways.

Since the motor is directly connected to the wheels and we assume that the wheels are rolling perfectly against the surface, the speed of the vehicle is directly proportional to the engine angular velocity:

$$v = \omega_w r,$$

where r is the radius of the wheels and ω_w is the angular velocity of the wheels.

Based on the no-slip condition, all the wheels are moving at the same speed v but in a different direction. The rear virtual wheel is moving in the heading direction of the vehicle θ and the front wheels in the direction they are pointing $\theta + \delta$. Therefore, the center of gravity will not move in the direction of the orientation of the vehicle, but it will be slightly offset by the steering angle δ. From the geometry of the vehicle, we can determine the direction, in which the center of gravity will move:

$$\beta = \arctan\left(\frac{l_r}{l_r + l_f}\tan\delta\right).$$

The linear and rotational velocity of the body of the vehicle in the global reference frame is described by these equations:

$$\begin{aligned}\dot{x} &= v\cos(\theta+\beta)\\ \dot{y} &= v\sin(\theta+\beta)\\ \dot{\theta} &= \frac{v\cos\beta\tan\delta}{l_r+l_f}.\end{aligned}$$

The kinematic model does not require any extra state variables on top of the configuration and the state variables of the actuators. The final state space X would therefore have five continuous dimensions and one discrete dimension for the number of passed waypoints:

$$X = \{(x, y, \theta, m, \delta, \omega) \mid x, y, \omega \in \mathbb{R}, m \in \{0, 1, 2, \ldots, l\}, \theta, \delta \in [0, 2\pi)\}.$$

3.5.3 Dynamic Bicycle Model

Forces are the cause of linear and angular acceleration of every rigid body. The dynamic bicycle model models the forces between the tires of the vehicle and the road surface and describes the resulting accelerations. To describe the translation and rotation of the vehicle on a two dimensional plane, we need to know the values of two forces: the longitudinal forces F_x, which cause the motion forward or backward, and lateral forces F_y, which allow the vehicle to turn and which cause rotation of the vehicle. An overview of the forces and other important variables is shown in Figure 3.14.

If we knew the longitudinal forces of the wheels F_{xr} and F_{xf}, we could determine the total traction force acting at the center of gravity of the vehicle and the subsequent acceleration $\dot{v}$ of the vehicle:

$$\begin{aligned}F_{traction} &= F_{xr} + F_{xf}\cos\delta - F_{yf}\sin\delta\\ \dot{v} &= \frac{1}{m}\left(F_{traction} - F_{load}\right).\end{aligned}$$

where $m \in \mathbb{R}_+$ is the mass of the vehicle and F_{load} is a longitudinal load force on the vehicle, consisting of an aerodynamic drag force and rolling resistance.

The vehicle moves in the direction of the slip angle β and it rotates around its z axis at the angular velocity $\dot{\theta}$. With the knowledge of the lateral forces F_{yf} and F_{yr} we can model the behavior of these state variables:

$$\begin{aligned}\dot{\beta} &= \frac{1}{mv}\left(F_{yf}\cos\delta + F_{yr}\right) - \dot{\theta}\\ \ddot{\theta} &= \frac{1}{I_z}\left(l_f\left(F_{yf}\cos\delta - F_{xf}\sin\delta\right) - l_r F_{yr}\right),\end{aligned}$$

where I_z is the angular momentum of the vehicle around the z axis which is the rotational equivalent of mass.

The change of the coordinates center of gravity of the vehicle in the two dimensional plane is then calculated the same way as we did in the kinematic model:

$$\dot{x} = v \cos(\theta + \beta)$$
$$\dot{y} = v \sin(\theta + \beta).$$

Tire Model

A tire is an interface between the road and the vehicle. The engine produces torque which is transferred to the wheels through the drive train and this torque is translated into a force at the contact patch between the tires and the road. This force is what puts the vehicle into motion, and what allows the vehicle to turn by changing the steering angle.

There are several mathematical tire models, such as Brush, Fiala, Dugoff, or Pacejka models [28, 30], which describe how to calculate the tire forces F_x and F_y. The problem with these models is that they require a knowledge of many parameters, which must be obtained through experimental measurements of the properties of the actual vehicle.

This turned out to be a problem for us. Even though some students performed identification of the Pacejka magic formula parameters for a similar RC car model [29, 31], we were not able to perform the measurements and obtain good quality tire model for our custom experimental vehicle. We are therefore unable to utilize the properties of the dynamic bicycle model to describe our experimental vehicle.

3.5.4 Discussion

We are now able to describe the movement of the vehicle based on the throttle level and target steering angle inputs. The models of the steering servo and of the DC motor were obtained empirically by measuring the responses of the hardware to the control inputs. Especially the model of the DC motor is interesting, because it predicts the RPM of the motor not only based on the throttle level, but also on the angle of the wheels. This reflects how the vehicle loses its kinetic energy during cornering.

Due to our inability to precisely measure the parameters of tire models, we have to use a simplified kinematic model with a no-slip assumption for our experiments. The planner might be too optimistic find trajectories which are too aggressive and the vehicle will not be able to follow them precisely. We would be improve the trajectories found by our planner if we obtained the dynamic model in the future or if we were able to try our algorithm on a different vehicle for which we knew the parameters of a tire model. On the other hand, the dynamic model has two additional dimensions of the state space (the slip angle β and the yaw rate $\dot{\theta}$), which will make the computation task harder.

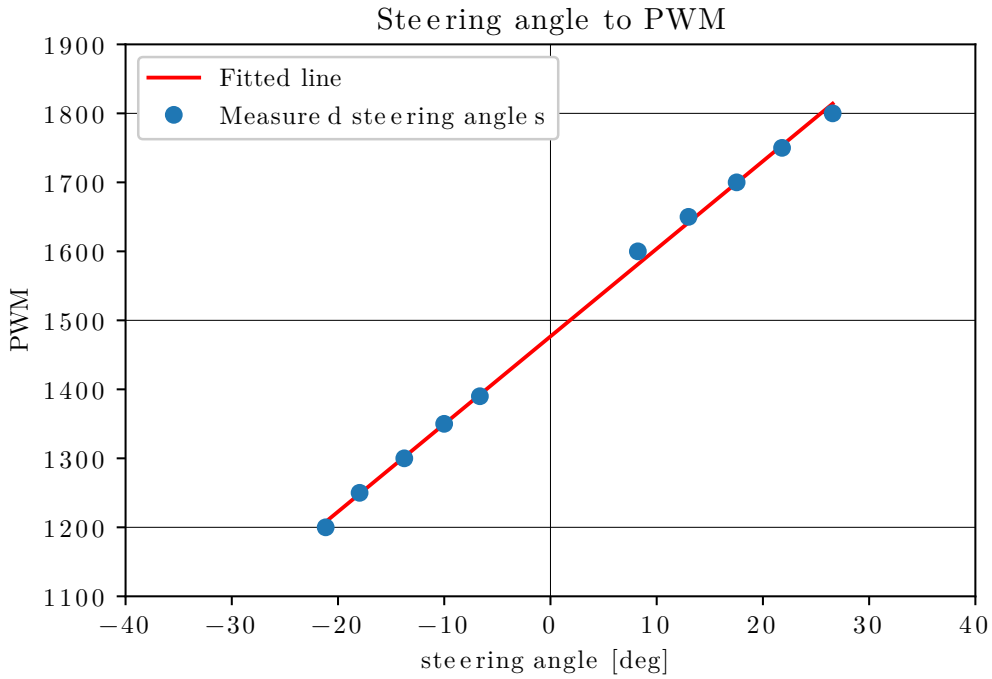


Figure 3.8: The measurement shows the imperfections in the hardware construction. The vehicle is able to go around circles with smaller radii when it is turning left (positive steering angles) than when it is turning right (negative steering angles).

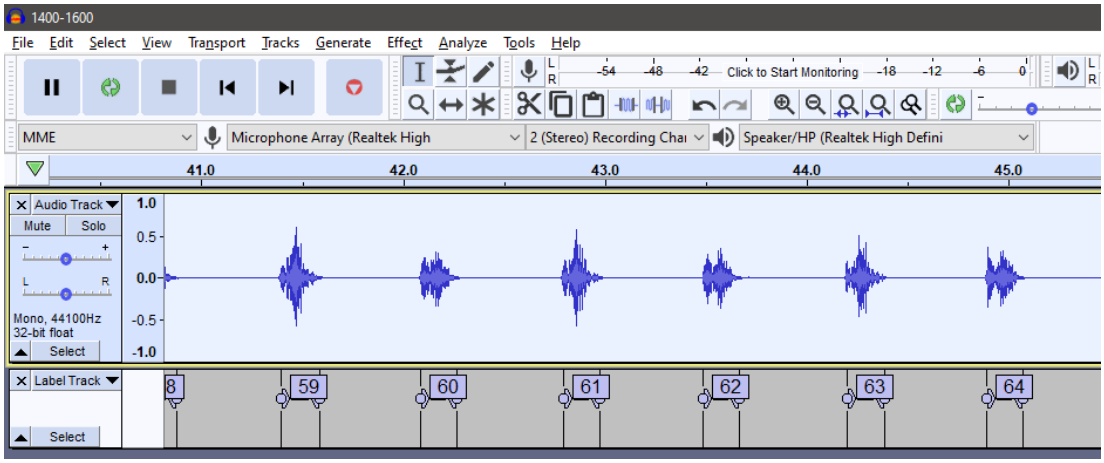


Figure 3.9: The servo setting periods are clearly identifiable in an audio recording after background noise and low frequency sounds are removed. This figure shows the interface of the *Audacity* tool used to analyze a recording of adjustments between the PWM signal of 1400 µs and 1600 µs.

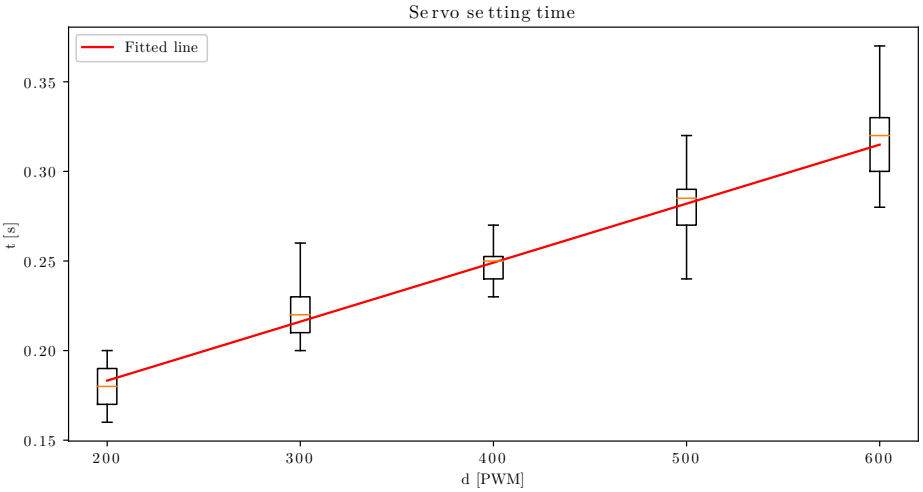


Figure 3.10: The setting time of the steering servo can be estimated with a linear function of the distance between the start and end PWM values.

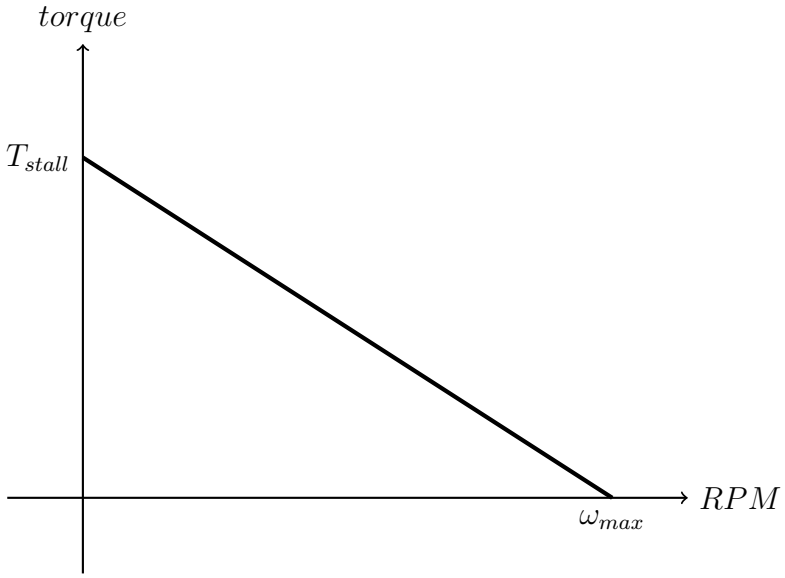


Figure 3.11: Linear approximation of a DC motor torque curve. This linear model has two parameters: the maximum RPM without any load ω_{max} and stall torque T_{stall}.

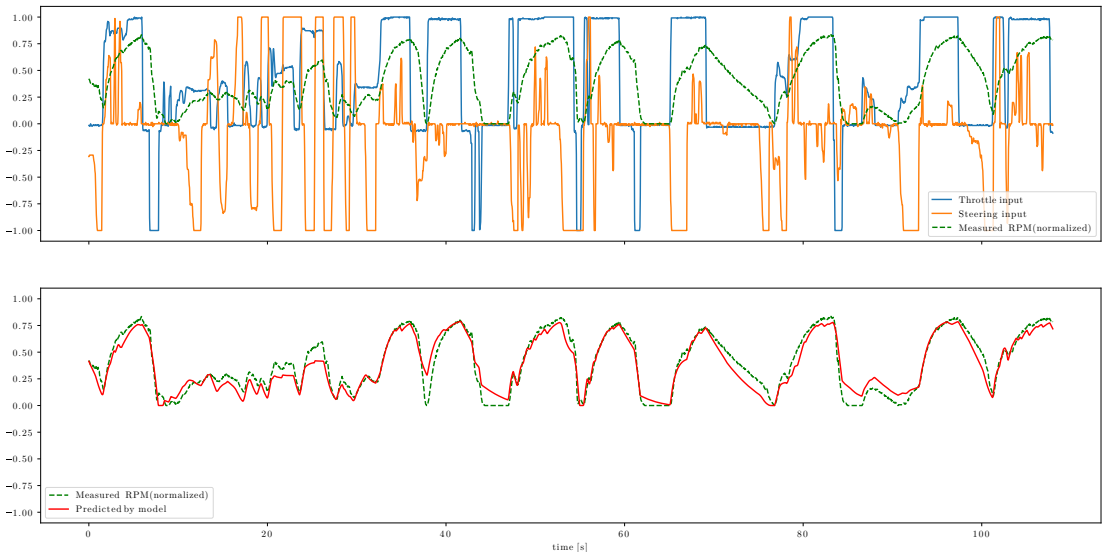


Figure 3.12: The first chart shows the throttle and steering inputs as well as the measured RPM (normalized). The second chart compares the measured and predicted RPMs for the same time period using the fitted parameters of our model.

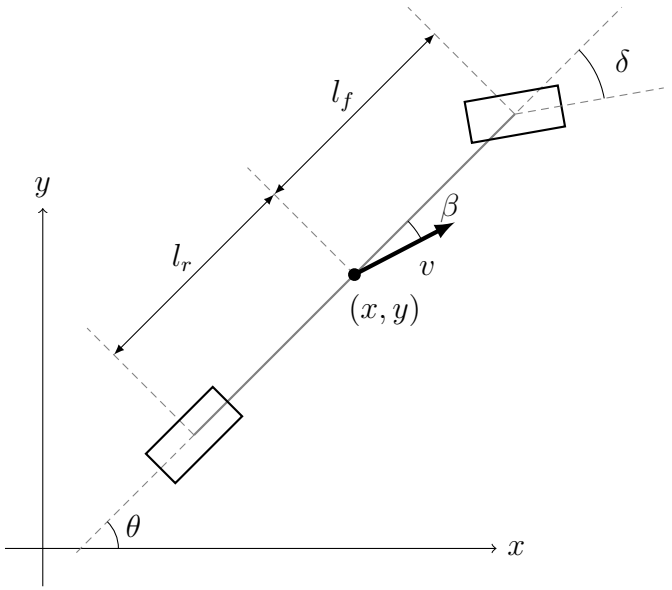


Figure 3.13: The two front and two rear wheels are merged into a single wheel at the center of each axle for simplification. The (x, y) position of the center of gravity moves at the speed v in the direction $\theta + \beta$ in the global reference frame.

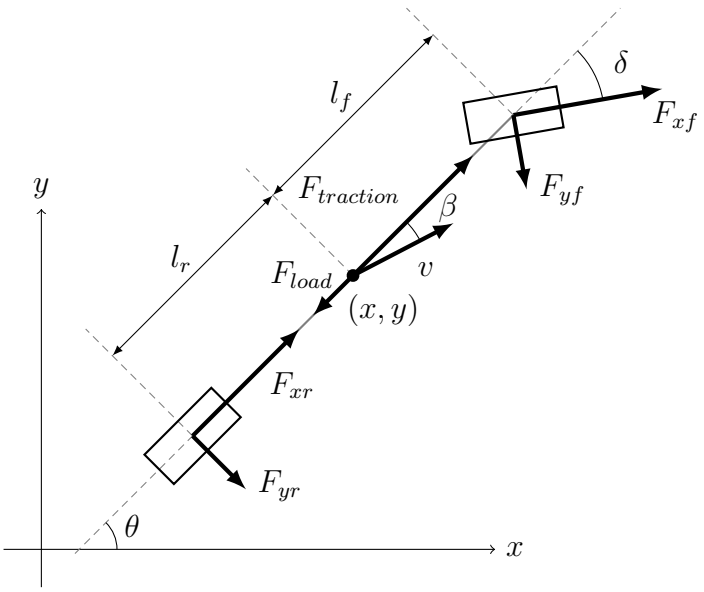


Figure 3.14: The dynamic model calculates the acceleration and the yaw rate based on the forces which act on the body of the vehicle. The configuration (x, y, θ) represents the position of the center of gravity of the vehicle and the heading angle in the global reference frame.

4. Trajectory Following

Given a reference state trajectory $\hat{x}_k = \langle x_0, x_1, \ldots, x_k \rangle$ obtained for a fixed time step Δt and a current state of the vehicle estimated from sensor readings using a localization and odometry algorithm, our task is to define a function which will select the next action $u \in U$ such that the car will move in a way in which it will follow the reference trajectory "as closely as possible". We will also consider avoiding any obstacles which were not known during the planning phase and which might have been hit if we had followed the trajectory blindly.

In this chapter, we will first formally define what we mean by following the trajectory "as close as possible". We will then describe algorithms which we will later implement and try on the experimental vehicle.

4.1 Reference State and Sub-trajectory

First, we must consider the timing of the next control input we will use. The last known vehicle state estimation $x_t \in X$ at time t could be used directly to calculate the error from the reference trajectory. The decision process will take some time $t_d > 0$, during which the vehicle will continue to move subject to the last control input $u \in U$. As a consequence of this delay, at the time $t + t_d$ it might not be appropriate to use the action selected by the trajectory tracking algorithm anymore.

To achieve better accuracy, we can estimate the decision delay t_d by averaging the reaction times of the selected algorithm gathered during experiments. We can use our system simulator to estimate the state of the vehicle at the time $t + t_d$. when the action should be applied. We will therefore use the predicted state $x_{pred} = f_{t_d}(x_t, u)$ to find a corresponding state on the reference trajectory to calculate an error.

We must determine what part of the trajectory we have already followed and what is still remaining to be followed. For the predicted state of the vehicle x_{pred}, we will select its reference state along the reference trajectory $\hat{x}_k$ as the state, which is closest to the current location. In the ideal case, this would be one of the states along the reference trajectory or a state between two adjacent states. In practice, the vehicle will deviate from the reference trajectory due to sensing and steering imperfections, but we assume that it is still near the reference trajectory. When the vehicle deviates too much, we assume that the trajectory will be re-planned with respect to the current state of the vehicle. The following function describes how we find the reference state x_{ref}:

$$x_{ref}(x_{pred}, \hat{x}_k) = \arg\min_{x_i \in \hat{x}_k} \left\| \kappa_{x,y}(x_{pred}) - \kappa_{x,y}(x_i) \right\|, \tag{4.1}$$

where $\kappa_{x,y} : X \to \mathbb{R}^2$ is the function we defined in Section 3.2.2 which returns the position vector in the given state.

We will consider the reference state to be the point which splits the trajectory into the already traveled part and the rest. If i is the index of the reference state in $\hat{x}_k$, then the reference sub-trajectory which should still be followed is $\hat{x}_{ref} = \langle x_i, x_{i+1}, \ldots, x_k \rangle$.

4.2 Trajectory Tracking Error

To evaluate which control input to select and to evaluate the choice of this input, we must define an error function $e : X \times X \to [0, 1]$ which tells us how far apart the two states are from each other. This function will allow us to compare vehicle states between each other and determine which one of them is better. We will also define a second error $\hat{e}_{\Delta t} : X^* \times X^* \to [0, 1]$ function to calculate an error between two sequences of states.

4.2.1 Single State Error

We will calculate the error between two states by combining three components: position error, heading error, and velocity error.

Position Error

The position error is simply the distance from the desired location:

$$e_p(x, x_{ref}) = \frac{\|\kappa_{x,y}(x) - \kappa_{x,y}(x_{ref})\|}{w}, \tag{4.2}$$

where $w \in \mathbb{R}$ is a normalization constant which represents the maximum possible position error for the given track.

Heading Error

The heading error is the misalignment of the current heading angle and the heading angle in the reference state:

$$\begin{aligned} \Delta\theta(x, x_{ref}) &= |\kappa_\theta(x) - \kappa_\theta(x_{ref})| \\ e_\theta(x, x_{ref}) &= \frac{\min\{\Delta\theta(x, x_{ref}), 2\pi - \Delta\theta(x, x_{ref})\}}{2\pi}. \end{aligned} \tag{4.3}$$

Velocity Error

Being close to the reference velocity is crucial to avoid understeering and oversteering in the corner ahead of the vehicle. We will use the following function to calculate the velocity error:

$$e_v(x, x_{ref}) = 1 - \begin{cases} v(x_{ref})/v(x) & \text{when } v(x_{ref}) < v(x), \\ v(x)/v(x_{ref}) & \text{otherwise.} \end{cases} \tag{4.4}$$

Combined Error Each of the three error functions described by functions (4.2), (4.3), and (4.4) provide an error in the range of $[0, 1]$. From these three errors, we can calculate a weighted combined error:

$$e(x, x_{ref}) = \frac{\alpha e_p(x, x_{ref}) + \beta e_\theta(x, x_{ref}) + \gamma e_v(x, x_{ref})}{\alpha + \beta + \gamma},$$

where $\alpha, \beta, \gamma \in \mathbb{R}_{\geq 0}$ are the weights of the individual components of the error. By setting any of the value to 0, we can simply ignore the error. By selecting

different values, we will change the significance of each components and we will also compensate for the differences in typical values of the respective errors. For example, if w is very high, the position error will often be close to zero, even though the position error is non-negligible.

4.2.2 Trajectory Error

Finally, we can define an error between a state trajectory $\hat{x} = \langle x_1, x_2, \ldots, x_l \rangle$ and the reference trajectory $\hat{x}_{ref} = \langle x'_1, x'_2, \ldots, x'_l \rangle$. The idea is to reward trajectories which start off the original trajectory but they re-align with the reference. Because we are comparing discrete sequences of samples, it is important to compare only trajectories where the samples are taken at the same constant time step Δt:

$$\hat{e}_{\Delta t}(\hat{x}, \hat{x}_{ref}) = \frac{\sum_{i=1}^{l} i \cdot e(x_i, x'_i)}{l(l+1)/2}.$$

4.3 Geometric Controller

There are several algorithms which will allow us to track the reference trajectory based only on the geometry of the vehicle and the error between the current state and the desired state. We will use two algorithms, a Proportional-integral-derivative controller (PID) controller to regulate the target velocity of the vehicle and a *Pure Pursuit* controller to control the steering angle.

An advantage of the geometric controller is its simplicity and very straightforward implementation. On the other hand, we must also consider how to avoid collisions with obstacles.

4.3.1 Proportional Integral Derivative (PID)

PID is a classical controller used in many different areas for regulating the output of a system based on reference values. The name stands for Proportional, Integral, and Derivative terms, which are multiplied by predefined gains $k_p, k_i, k_d \in \mathbb{R}$ and combined into an output value which will be directly used as the control input for an actuator.

The proportional term is equal to the error between the current value and the reference value. The derivative term is the change in the error since the previous iteration. Finally, the integral term is the accumulated error over time since the last time there was no error.

We can use PID to calculate the target throttle control input τ_t from the velocity error e_v:

$$\begin{aligned} e_i &= e_v(x_i, x_{ref,i}) \\ \tau_t(e_i) &= k_p \cdot e_i + k_i \cdot \sum_{j=i-n}^{i} e_j + k_d \cdot (e_i - e_{i-1}), \end{aligned}$$

where $x_i, x_{ref,i}$ are the i-th state of the vehicle and the reference state, $n \leq i$ is the number of steps since the error was zero ($e_{i-n} = 0 \wedge \forall j > i - n : e_j > 0$).

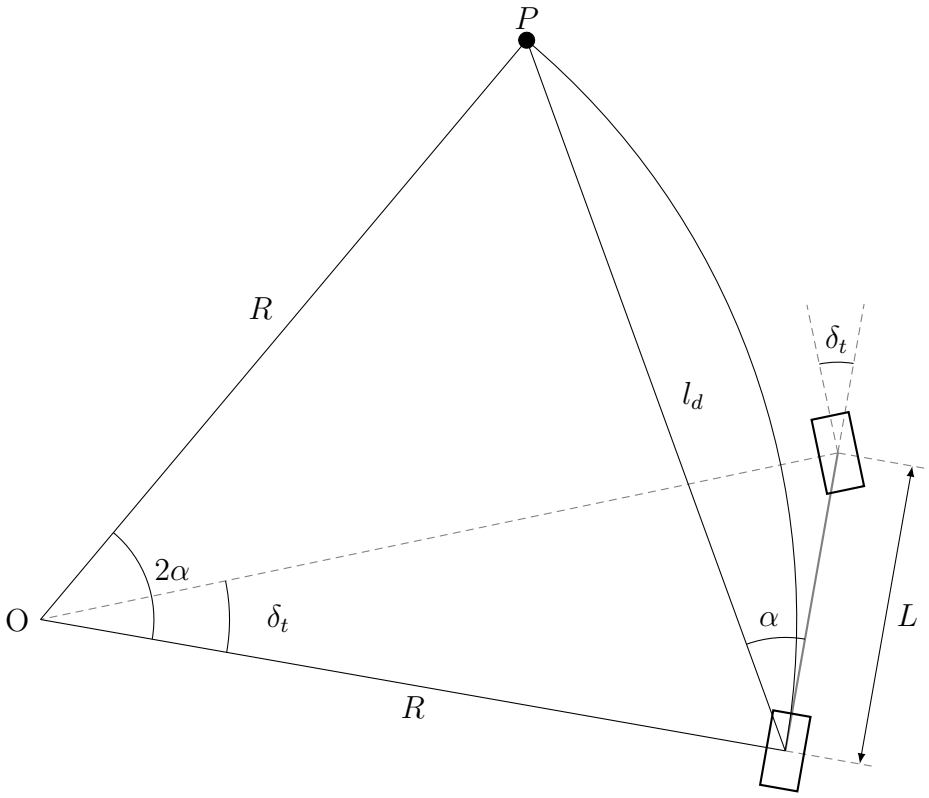


Figure 4.1: The reference point P can be imagined as a "carrot" which the vehicle is pursuing. The lookahead distance l_d can be variable based on the velocity of the vehicle.

The values of the gain parameters must be tuned experimentally for the specific vehicle. When a gain is set to 0, it means that the corresponding error is not used. The controller can then be referred to as for example just P, PI, or PD. The control input value must be within the range of $[-1, 1]$ so in the case that the PID controller calculates a greater value, we must use the closest valid value.

4.3.2 Pure Pursuit

A PID controller which regulate the steering angle based on the heading misalignment e_θ could be easily defined as well. There are although other algorithms, which are more advanced, yet very simple as well. The Pure Pursuit algorithm is a purely geometric algorithm used for determining the steering angle of the vehicle based on a target state at a lookahead distance from the rear axle wheel along the trajectory. This target point is sometimes referred to as to a "carrot", because the vehicle pursues this target state, but it moves along with the vehicle and it will never reach it. This algorithm was used for self-driving car steering by the researches at The Robotics Institute at the Carnegie Mellon University in the *NavLab* project [32].

The lookahead distance can greatly affect the performance of the algorithm. Short lookahead distance can cause oscillation whereas long lookahead distance can lead to slow convergence to the path. Instead of using a constant lookahead distance, we will calculate its value from the current velocity of the vehicle:

$$l_d = \begin{cases} Kv \\ l_{min} & v < v_{min}, \end{cases}$$

where K is a constant gain parameter which must be determined experimentally.

From the geometry of the vehicle and a no-slip assumption of the kinematic vehicle model, we can calculate the steering angle which will keep the vehicle on a circle with the radius R, which can be easily calculated if we obtain the angle α between the longitudinal axis of the vehicle and the location of the target state:

$$R = \frac{l_d}{2\sin\alpha}$$
$$\delta_t = \arctan\frac{L}{R} = \arctan\frac{2L\sin\alpha}{l_d} = \arctan\frac{2L\sin\alpha}{Kv}.$$

The v in the denominator will ensure that the steering angle is not very large when the vehicle is traveling at high speeds which could lead to sharp turns and loss of control over the vehicle. The geometry is visualized in Figure 4.1. The vehicle has a limited range of valid steering angles. We must clamp the target angle to the limiting angle δ_{min} or δ_{max} which will try to bring the vehicle as close to the desired radius as possible.

The vehicle itself cannot change the steering angle instantaneously. This will inherently lead to tracking errors. The lookahead distance gain K must be tuned well so that the vehicle does not start turning too early or too late at a given velocity so that it always stays within the bounds of the track, especially at high speeds.

4.4 Dynamic Window Approach (DWA)

Another approach to trajectory following is to select the next control input based on a prediction of the effects in a short horizon. While during trajectory planning we had time to plan several corners ahead, the controller needs to be very fast and select control inputs with very little delay after every measurement of the state of the vehicle. In this section, we will consider two model-based algorithms with different properties.

At each time when we want to select the next control input for our vehicle, the DWA algorithm [33] samples the control space of the robot and simulates the trajectories which the robot would travel if the action was applied for a short time period $T = k \cdot \Delta t, k \in \mathbb{N}$. The resulting trajectories, which do not result in a collision, are scored and the action of the best trajectory is used. The outline of the algorithm adapted for our problem can be seen in Algorithm 6 and it is visualized in Figure 4.2.

The function `Unfold` simulates the motion of the vehicle from the state x_0 using the control command u in the next k steps and returns the state trajectory which is obtained.

The function `EmergencyStop` is used when there is no valid trajectory and it selects a control input from the set of possible actions which will slow down the vehicle as much as possible before hitting the obstacle.

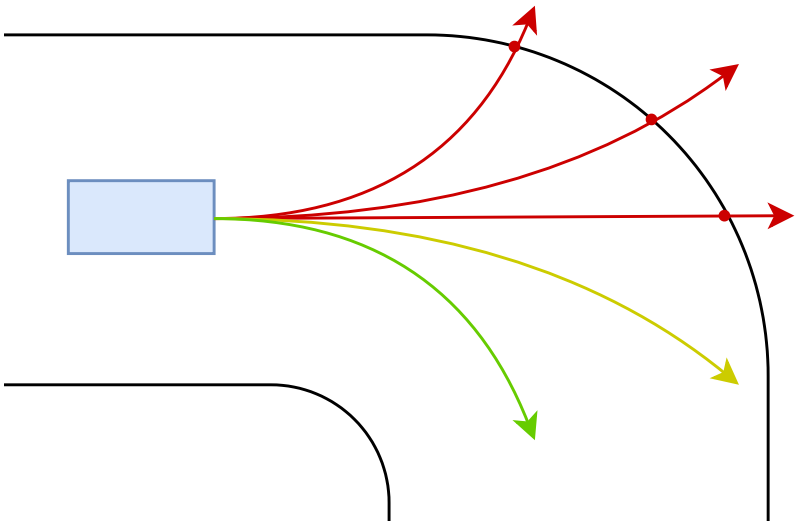


Figure 4.2: Application of different actions can be visualized as a set of "tentacles" around the vehicle. Each "tentacle" is a prediction of the positions of the vehicle during the prediction horizon period. Each of the predicted trajectories is scored and the action which produces the best trajectory is selected.

Algorithm 6: Dynamic Window Approach

Input: Collision detection function c, time step Δt, finite set of actions U_f, horizon $k \in \mathbb{N}$

Output: An action $u \in U_f$

Parameter: Initial state x_0, reference trajectory $\hat{x}_{ref}$

Algorithm `DWA`$(x_0,\ \hat{x}_{ref})$**:**
- $T \leftarrow \emptyset$
- **foreach** $u \in U_f$ **do**
  - $\hat{x}' \leftarrow$ `Unfold`$(x_0,\ u,\ k,\ \Delta t)$
  - **if** $\forall x \in \hat{x}' : c(x) = F$ **then**
    - $T \leftarrow T \cup \left\{\hat{x}'\right\}$
  - **end**
- **end**
- **if** $T = \emptyset$ **then**
  - **return** `EmergencyStop()`
- **end**
- **return** $\arg\min_{\hat{x} \in T} \hat{e}_{\Delta t}(\hat{x})$

The time complexity of the algorithm is $O(k \cdot n)$ and it requires $O(k \cdot n)$ of memory, for $n = |U_f|$. Since both values are small constants, the algorithm itself can be very fast and the delay t_d will be low.

The algorithm prefers trajectories which will bring the vehicle closer to the reference trajectory while avoiding collisions with any obstacles along the trajectory which might not have been known during the planning phase.

5. Experiments

5.1 Full Lap Planning

In this experiment we test the Hybrid A* and SEHS algorithms as we described them in Section 3.3 on several different circuit maps. For each circuit, the algorithms should find a near time-optimal trajectory from a starting position through a series of waypoints until the last waypoint of the circuit is reached. We then compare the quality of the solutions, how much of the state space the algorithm had to explore before it found the solution, and how long it took to calculate the result on the testing computer.

5.1.1 Test Setup

Both Hybrid A* and SEHS are dependent on the level of discretization. The finer the discretization is, the closer the results are to the optimal solution. At the same time, the size of the searched state space increases. In order to compare our implementation of the two algorithms, we ran them with several different combinations parameter values.

The heading angle (θ) values were arranged into 18, 36, or 52 uniform sectors. The range of the RPM values of the motor was split into 20, 40, and 80 subranges. Additionally, the Hybrid A* algorithm divided the xy plane into a grid of square cells with the side lengths of $4r$ or $5r$, where r is the radius of the vehicle. The SEHS algorithm discretized the xy plane by finding a path of circles in the Space Exploration step. We require the radii of the circles to be between r and $5r$. The actions available to the planner were obtained as a cross product of 5, 9, or 21 throttle levels and 11, 21, or 31 different steering angles. The discrete time step δt was 0.04 s (25 Hz) at all times.

Circuits are represented by an occupancy grid with a resolution of 0.05 m per grid cell. For each circuit, we first perform track segmentation, as described in Section 3.4, to detect corners which we use as waypoints. We then let both algorithms find the near time-optimal trajectory from the initial configuration defined for each circuit to the last waypoint. The parameters of the vehicle model are selected to match the behavior of the F1/10 simulated vehicle model in the Gazebo simulator which we used also for simulated racing in section 5.2.3.

We measure the number of search nodes the algorithm opens during search and the time it takes to find the solution on a testing computer. Because both of the algorithms are deterministic, they always find the same solution for the same problem and they always open the same number of search nodes. For the SEHS algorithm, we do not measure the time of the *Space Exploration* step and we only measure the actual *Heuristic Search*. The execution times differ slightly and so we repeat the measurement ten times and calculate the average execution time. We used a desktop computer with an AMD Ryzen 7 3700X CPU running at the base clock of 3.6 GHz (4.4 GHz boost) and 32GB of DDR4 RAM at 3200 MHz. The source code was written in C++17 and it was compiled using GCC 9.2 with the `-O3` and `-ffast-math` optimization flags.

5.1.2 Results

The results of our experiments are shown in Figures 5.2, 5.3, 5.4, 5.5, 5.6, and 5.7. Each figure contains a table with a solution with the fastest lap time and the solution which took the least amount of time to compute for the Hybrid A* algorithm and for the SEHS algorithm. We show also the discretization parameters which were used to obtain these result as well as the visualization of the trajectory that was found in the shortest time period. The rest of the results obtained with all the different combinations of parameters are available in CSV files in the attachments of this thesis[1].

The curve which represents the trajectory of the vehicle has a color ranging from red to green. The color corresponds to the speed of the vehicle at the given position. The segments where the vehicle moves slowly is shown in red and the faster the car is moving the closer the color is to bright green. The trajectory is plotted over a map of the circuit with white parts showing the road and gray parts the boundaries of the track. As we mentioned earlier, the occupancy grids for the circuits all have resolution of 0.05 m. Every 20 grid cells correspond to 1 m in real world. Therefore for example the circuit in Figure 5.7 represents an area of 25 m × 25 m.

Each trajectory is accompanied by three charts, showing the control inputs for the vehicle, the predicted state of the motor and the steering servo, and the speed profile of the trajectory. The speed profile has the top speed and the average speed marked with horizontal dashed lines.

From our results, it is hard to declare a winner of the two algorithms. Both preformed similarly and each of them found a better solution than the other for at least one of the test circuits. In the thesis in which Chao Chen introduces the SEHS algorithm, it clearly outperforms the Hybrid A* algorithm [27]. This is probably due to differences in the internal implementation and the choice of discretization parameters. It is clearly possible to tune the Hybrid A* algorithm to perform similarly or better than SEHS with the same parameters. The SEHS algorithm also failed to provide a solution within a time limit of 10 s while our implementation of the Hybrid A* algorithm managed to solve each test circuit.

We analyzed the impact of the parameters on the performance and the quality of the results of the algorithms. The data show a clear trend of an increased computation time as the number of actions and the discretization resolution increases. What we don't see though, is a significant increase in the quality of the solutions. As the Figure 5.1 shows, good solutions are found both by a coarse discretization and low number of actions and a fine discretization and a high number of actions.

5.2 Autonomous Race

To further test the algorithms, we implemented the complete Artificial Racing Agent on top of the ROS and we built a custom hardware inspired by the F1/10 platform. The details of the implementation are described in the appendixes Experimental Vehicle (Appendix A) and Technical Documentation (Appendix B).

[1] The results for each circuit are in a separate CSV file in the `/experiments/-algorithm-testing/planning-benchmark-results` directory

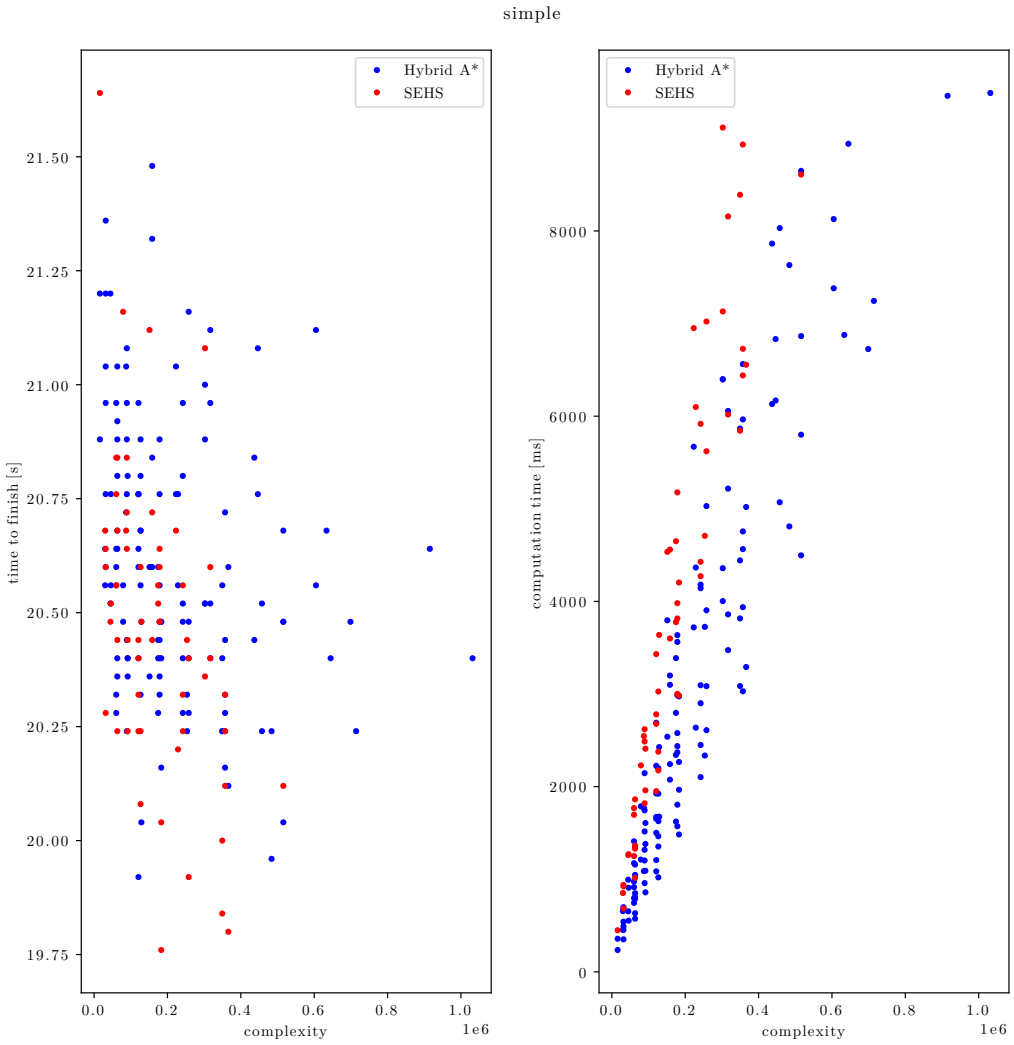
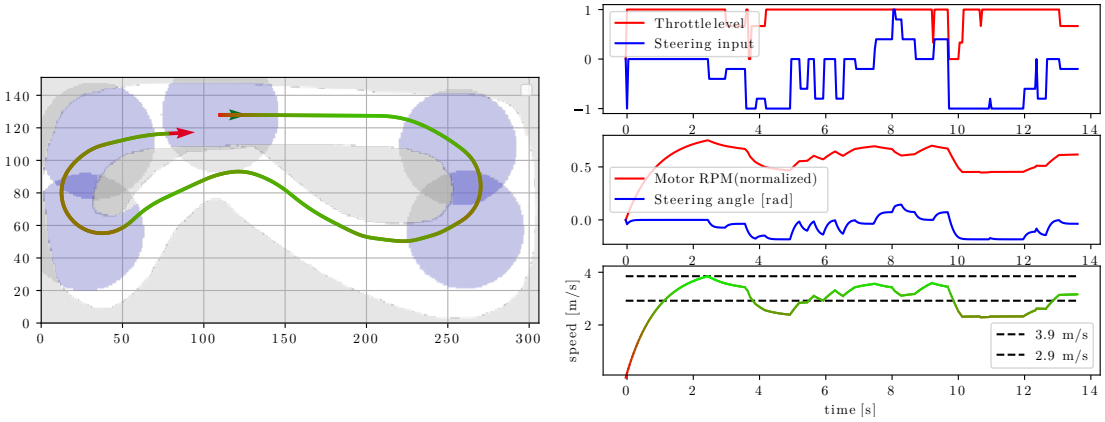
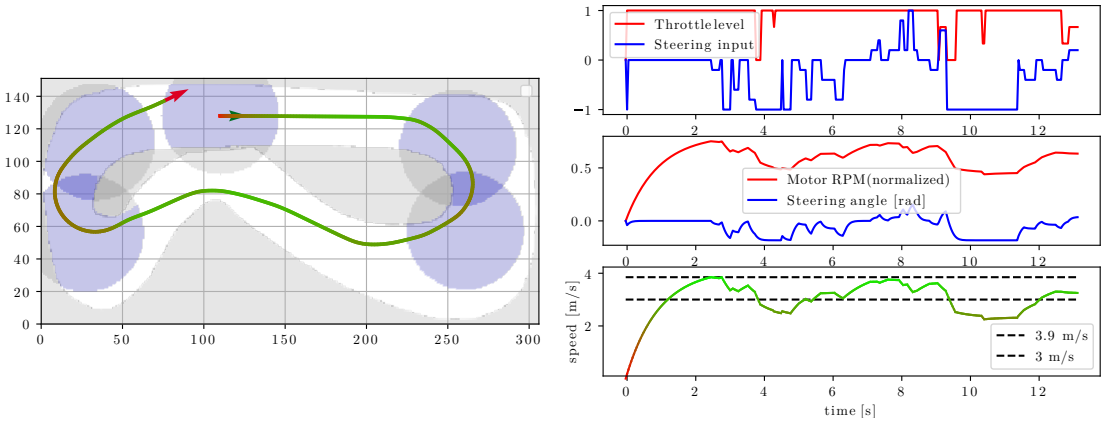


Figure 5.1: This figure visualizes the impact of the complexity of the parameters on the performance of the algorithm and on the quality of the results for the circuit "Simple". The *complexity* is calculated as a product of the number of actions, the number of motor discretization levels, and the number of heading angle discretization angles.



(a) Solution found by Hybrid A* in 192.30 ms



(b) Solution found by SEHS in 253.60 ms

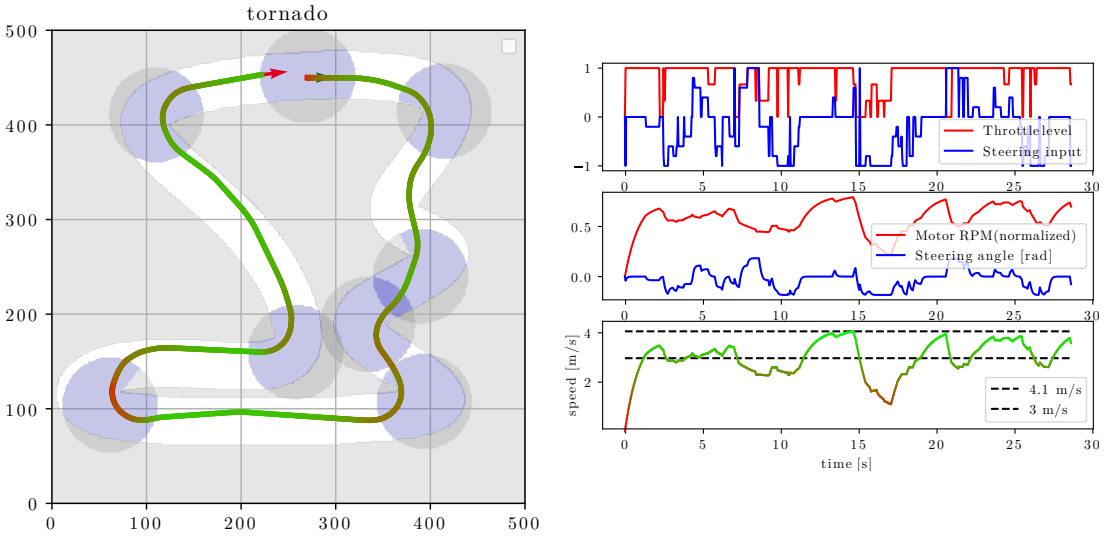
Available actions	Discretization xy [m]	RPM	θ	Opened nodes	Search time [ms]	Lap time [s]
44	1.80	20	18	**67 301**	**192.30**	13.64
248	1.44	20	36	838 926	2544.00	**12.28**

(c) Hybrid A* found both a trajectory in the least amount of time and also a trajectory with the best lap time.

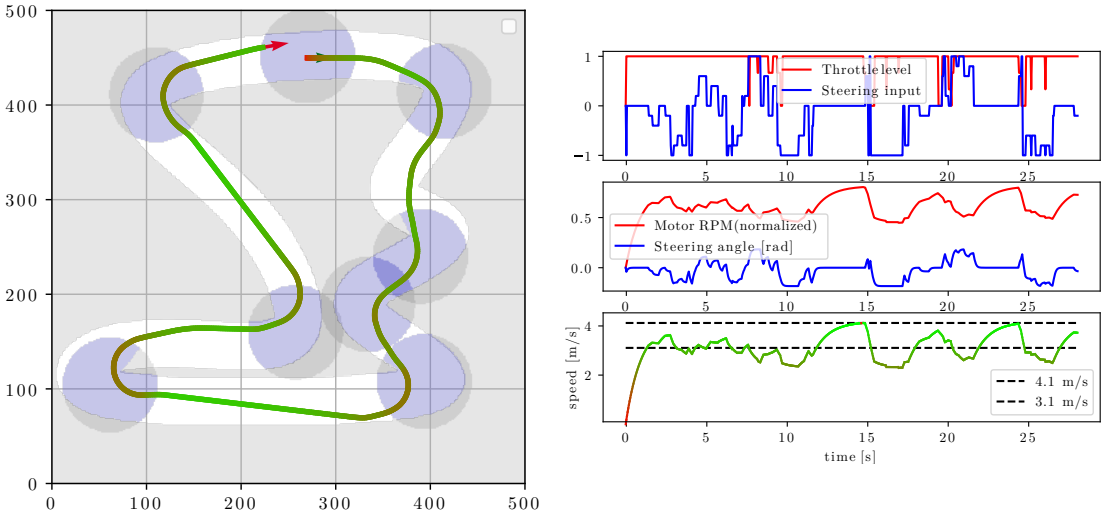
Available actions	Discretization Circles	RPM	θ	Opened nodes	Search time [ms]	Lap time [s]
44	71	20	18	80 763	253.60	13.16
84	71	80	36	1 054 427	2171.60	12.48

(d) SEHS could not find a solution faster nor could it find a better solution than Hybrid A* with any of the tested combination of parameters.

Figure 5.2: Circuit “Porto” is a simple track which was inspired by the track used in the 2018 F1/10 competition in Porto.



(a) Solution found by Hybrid A* in 494.50 ms



(b) Solution found by SEHS in 551.30 ms

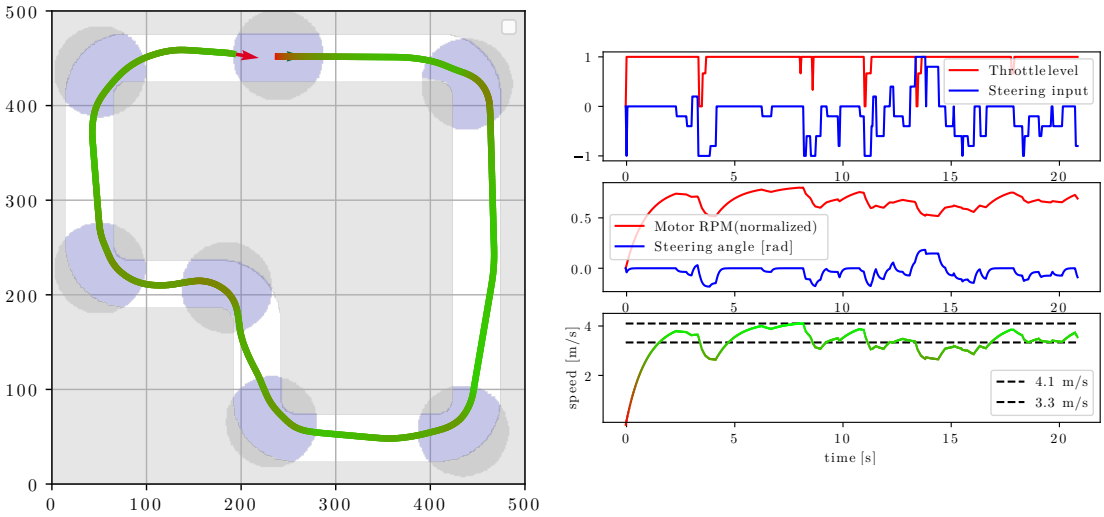
Available actions	Discretization xy [m]	RPM	θ	Opened nodes	Search time [ms]	Lap time [s]
44	1.44	20	18	177 778	**494.50**	28.64
84	1.44	80	52	3 464 766	6611.70	**26.24**

(c) Hybrid A* results

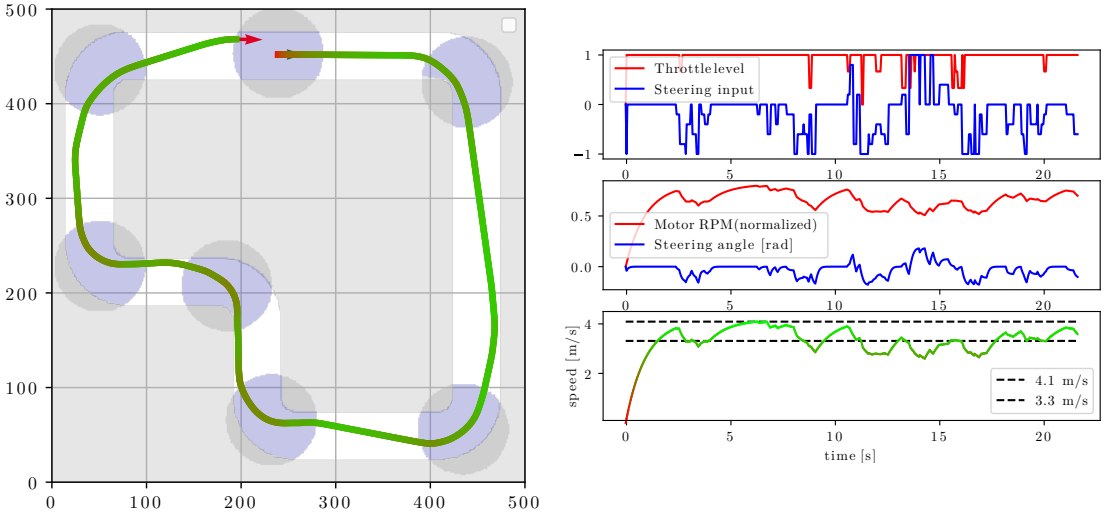
Available actions	Discretization Circles	RPM	θ	Opened nodes	Search time [ms]	Lap time [s]
44	126	20	18	**144 864**	551.30	28.08
84	126	80	52	2 827 948	6397.60	26.84

(d) SEHS results

Figure 5.3: Circuit “Tornado”



(a) Solution found by Hybrid A* in 236.20 ms



(b) Solution found by SEHS in 449.20 ms

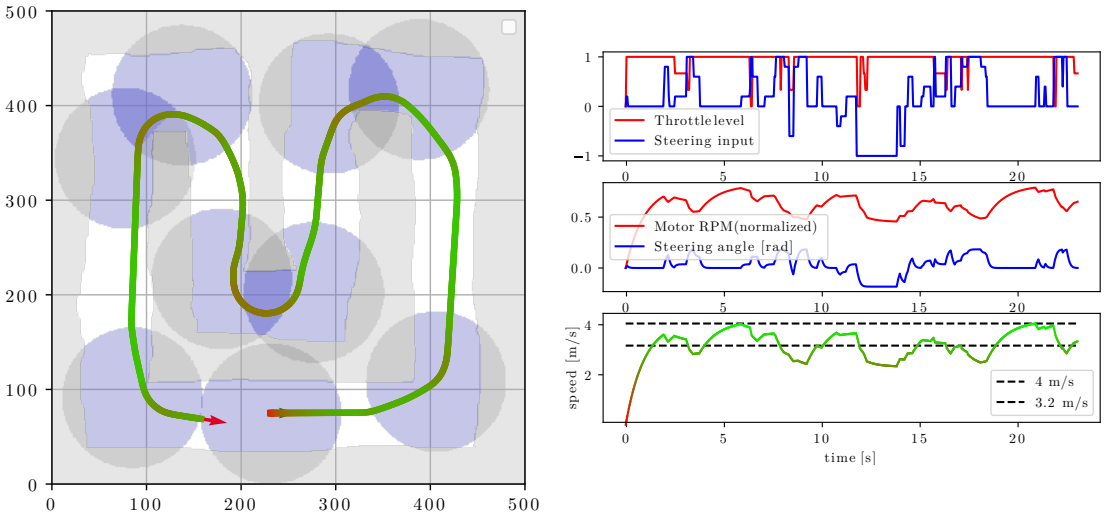
Available actions	Discretization xy [m]	RPM	θ	Opened nodes	Search time [ms]	Lap time [s]
44	1.80	20	18	**78 401**	**236.20**	20.88
84	1.44	40	36	807 771	1926.50	19.92

(c) Hybrid A* results

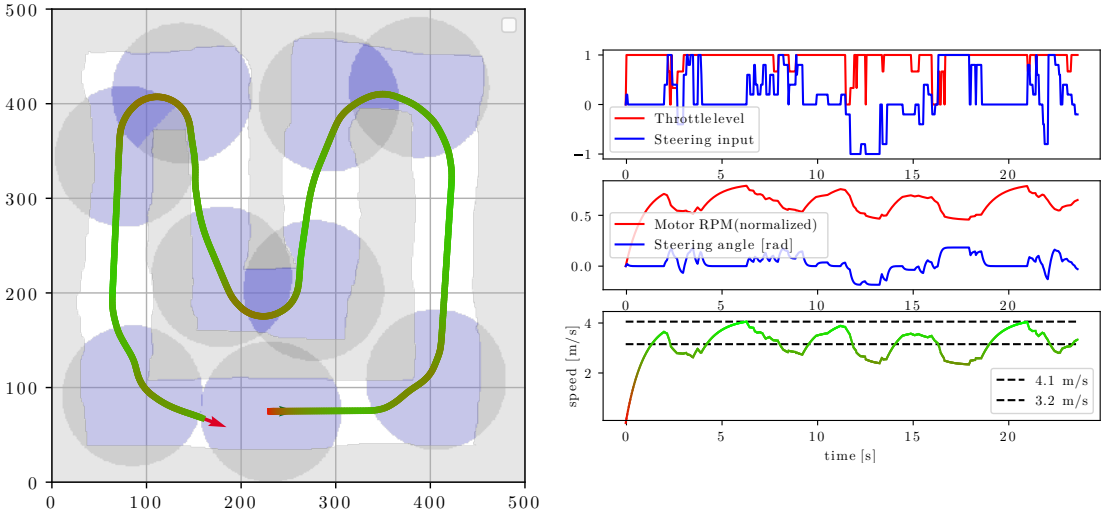
Available actions	Discretization Circles	RPM	θ	Opened nodes	Search time [ms]	Lap time [s]
44	134	20	18	129 796	449.20	21.64
88	134	40	52	1 443 863	4204.30	**19.76**

(d) SEHS results

Figure 5.4: Circuit “Simple”



(a) Solution found by Hybrid A* in 440.10 ms



(b) Solution found by SEHS in 533.60 ms

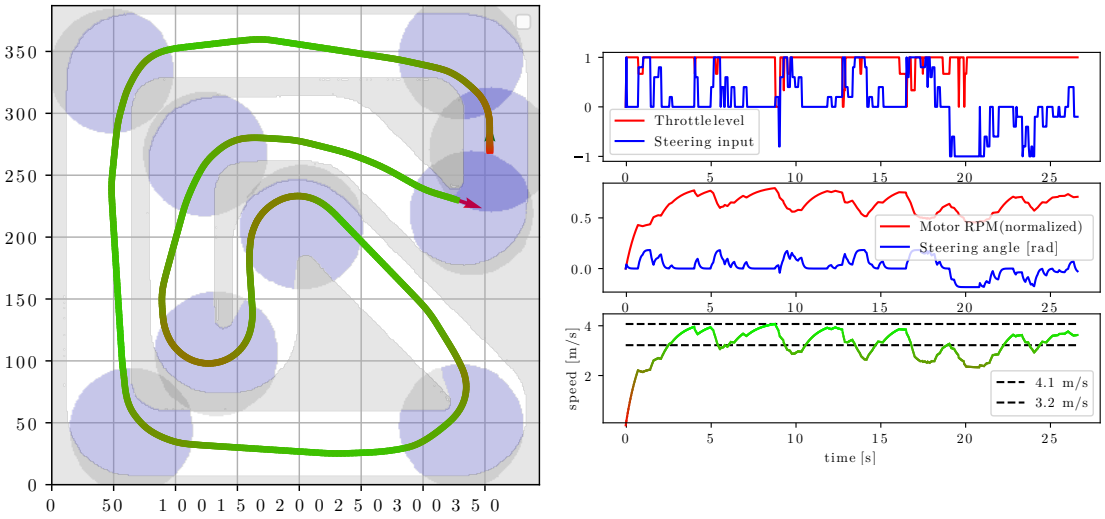
Available actions	Discretization xy [m]	RPM	θ	Opened nodes	Search time [ms]	Lap time [s]
44	1.80	20	18	**145 240**	**440.10**	23.08
84	1.44	40	36	1 442 255	3667.90	21.84

(c) Hybrid A* results

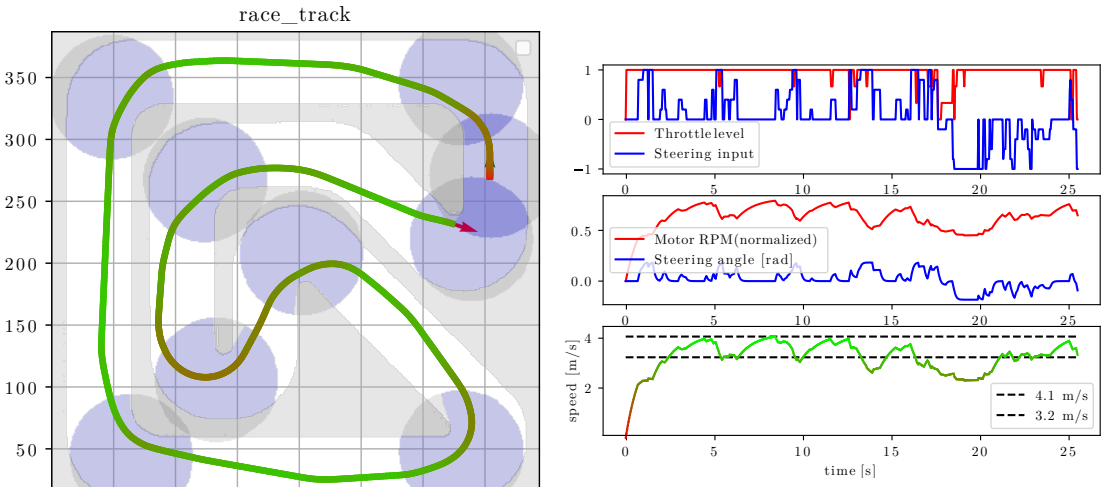
Available actions	Discretization Circles	RPM	θ	Opened nodes	Search time [ms]	Lap time [s]
44	123	20	18	147 605	533.60	23.64
44	123	40	52	829 326	2289.80	**21.60**

(d) SEHS results

Figure 5.5: Circuit “U”



(a) Solution found by Hybrid A* in 570.30 ms



(b) Solution found by SEHS in 598.10 ms

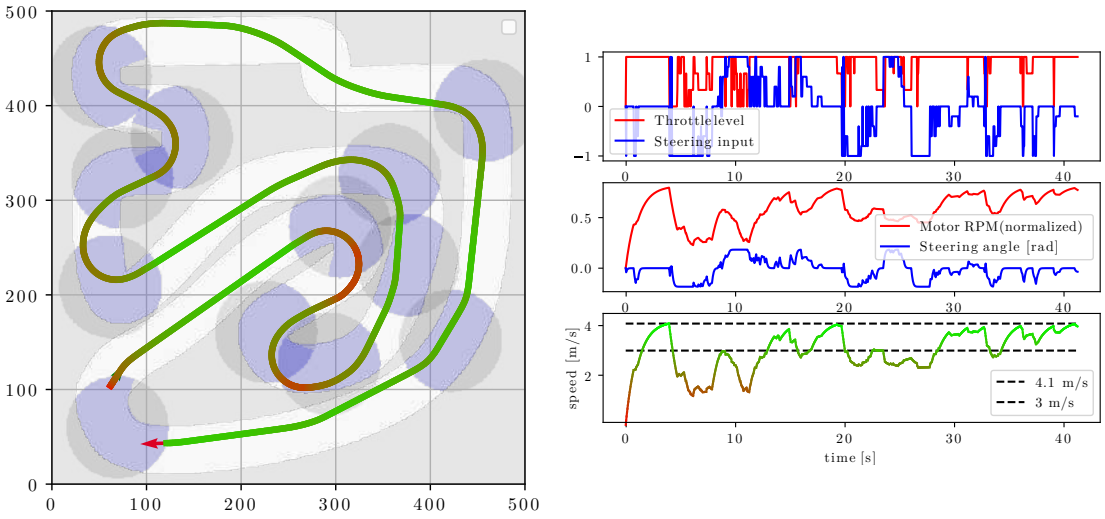
Available actions	Discretization xy [m]	RPM	θ	Opened nodes	Search time [ms]	Lap time [s]
44	1.80	20	36	195 755	**570.30**	26.64
84	1.80	80	36	1 450 767	2749.70	25.48

(c) Hybrid A* performed worse when compared to SEHS for this track.

Available actions	Discretization Circles	RPM	θ	Opened nodes	Search time [ms]	Lap time [s]
44	164	20	18	**169 002**	598.10	25.52
124	164	20	52	1 277 382	4809.20	**25.04**

(d) The solution found by the fastest lap time. Of the two solutions with the lowest computation time, the solution found by SEHS has a significantly better lap time.

Figure 5.6: Circuit “Race Track”



(a) Solution found by Hybrid A* in 1041.10 ms

Available actions	Discretization xy [m]	RPM	θ	Opened nodes	Search time [ms]	Lap time [s]
44	1.80	40	36	488 873	1041.10	41.28
88	1.80	80	36	1 922 591	3811.00	38.52

(b) Hybrid A* was the only algorithm of the two which found a solution. The SEHS algorithm did not find any solution for any combination of the parameters.

Figure 5.7: Circuit “Zurich”

In the following experiments, we tried to test the racing capabilities of the vehicle. The vehicle was given a complete map of the circuit at the beginning of the race. The goal was to go around the circuit repeatedly without colliding with walls and obstacles and reaching the best lap times as possible.

The agent repeatedly triggers the trajectory planning algorithm from the latest known state of the vehicle with the latest known state of the map with the obstacles marked in it through the current list of waypoints. At the same time, the vehicle is already driving and following the last published trajectory. Once the planning algorithm comes up with a new plan, it replaces the old plan and it can start planning again with the latest known state. We tested both the SEHS and the Hybrid A* algorithms.

5.2.1 Testing Criteria

For the planning algorithm to be viable, it must calculate the trajectory in a short period of time before the vehicle physically moves to the end of the previous trajectory. We were also interested in the safety of the planned trajectory and the distance the vehicle keeps from the obstacles while following the trajectory. Finally, our goal is to achieve fast lap times and we are interested in the maximum speed our vehicle can achieve while driving safely.

5.2.2 Real World Testing

Initially we wanted to perform tests outdoors on an asphalt road, but it proved problematic because of surface curvature and slope and because of unstable weather conditions. Another unexpected problem we had to solve was a problem with the LIDAR unit. It performed very unreliably in direct sunlight. In the end, we tested our vehicle indoors in a rectangular gym 10 m long and 6 m wide with concrete surface which was available to us for several days. We built several versions of an obstacle course using gym equipment and from cardboard boxes. One such circuit is shown in Figure 5.8.

The experiment consisted of two phases. First, we built a 2D map from the data coming from the LIDAR as the vehicle drove around the track using a remote controller. Using this map, we assembled a configuration file describing the racing circuit. Next, we started all the ROS nodes for vehicle state monitoring, circuit progress monitoring, trajectory planning, and trajectory following. Once the planning algorithm produced the first trajectory, the vehicle started following it. We were closely monitoring the movement of the vehicle visually and using telemetry data on a laptop through the `Rviz` program[2]. We were ready to take control of the vehicle using the remote control to prevent damage to the vehicle and to its surroundings at all times. Any input from the controller would cancel the autonomous mode and switch the vehicle into a remote-control mode. The vehicle then stayed in this mode until a button on the vehicle was physically pressed.

We struggled with unreliable odometry data for several days. We tested several different localization libraries, adjusted data we collected from the sensors,

[2] `http://wiki.ros.org/rviz`

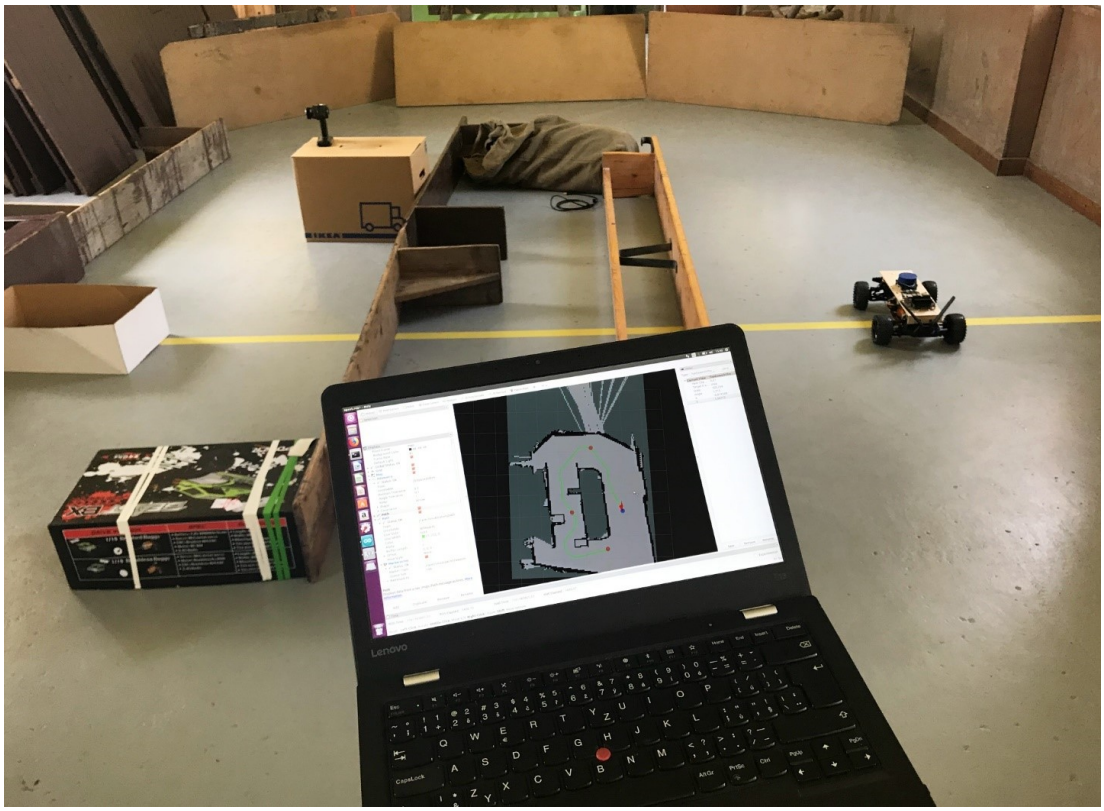


Figure 5.8: The planning algorithm finds a trajectory from the last known state of the vehicle through the waypoints ahead of it while avoiding the obstacles marked in the map.

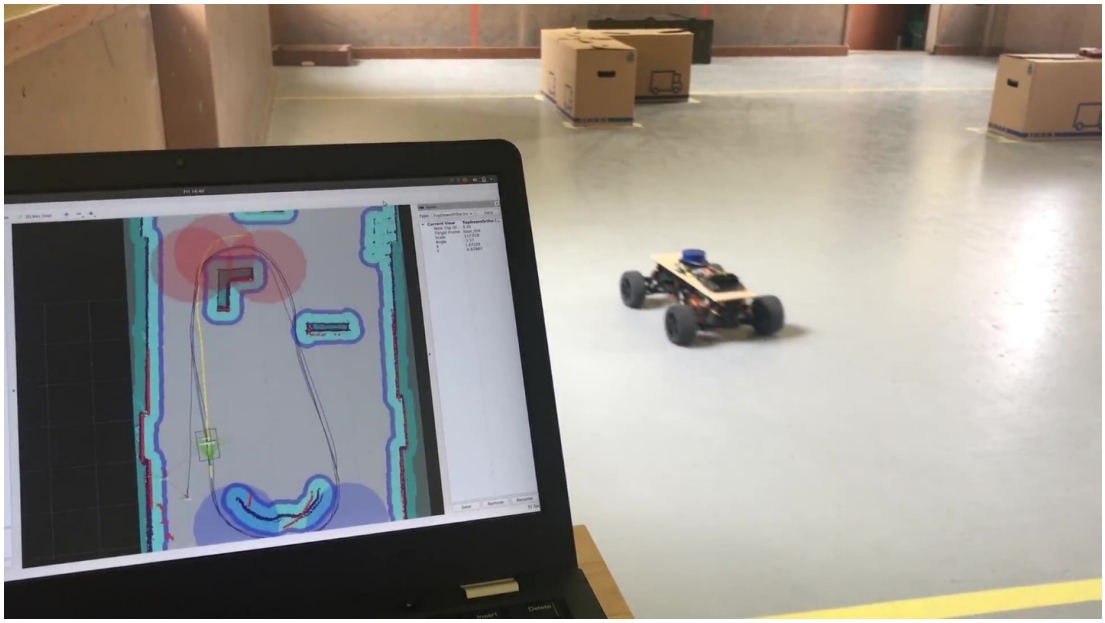


Figure 5.9: The vehicle is slowly driving along a circuit in a gym. The telemetry from the vehicle is monitored on a notebook connected to the vehicle over Wi-Fi.

and tweaked the parameters of the libraries until we were able to get usable odometry data at least at low speeds. At higher speeds, the vehicle would quickly lose track of its actual position and orientation on the map and it would collide with a wall or with an obstacle.

By the end of the testing, the vehicle was able to reliably circle around the track without hitting obstacles and without the odometry significantly diverging from the actual state of the vehicle. The car repeatedly completed a circuit which was approximately 20 meters long in 15 seconds at the steady speed of $1.3\,\mathrm{m\,s^{-1}}$ ($4.68\,\mathrm{km\,h^{-1}}$) which corresponds to the speed of $46.8\,\mathrm{km\,h^{-1}}$ of a full-size vehicle. At this speed, the limitations of the kinematic model do not manifest, and the vehicle had no issues undershooting or overshooting turns. A photo from this test is shown in Figure 5.9 and a video recording of this experiment is available in the attachment of this thesis [3].

The trajectory planning algorithm performed well even with just the low-powered ARM processor and the vehicle had never reached the end of the planned trajectory before a new extended trajectory was published. The Dynamic Window Approach (DWA) algorithm kept the vehicle on the track and avoided all cardboard boxes placed along the circuit.

Although the robot was able to navigate autonomously without any collisions, the speeds we were able to reach, without losing track of the position of the vehicle, were low and did not reach the speed at which we would start seeing any effects of high speed maneuvering, such as wheel slipping and skidding and overshooting corners. We were also unable to test evasion of dynamic obstacles because the obstacle detection library produced too many ghost obstacles from our noisy and low-frequency LIDAR scans.

[3]The videos are locaded in the `/experiments/experimental-vehicle-gym` folder in the thesis attachment.

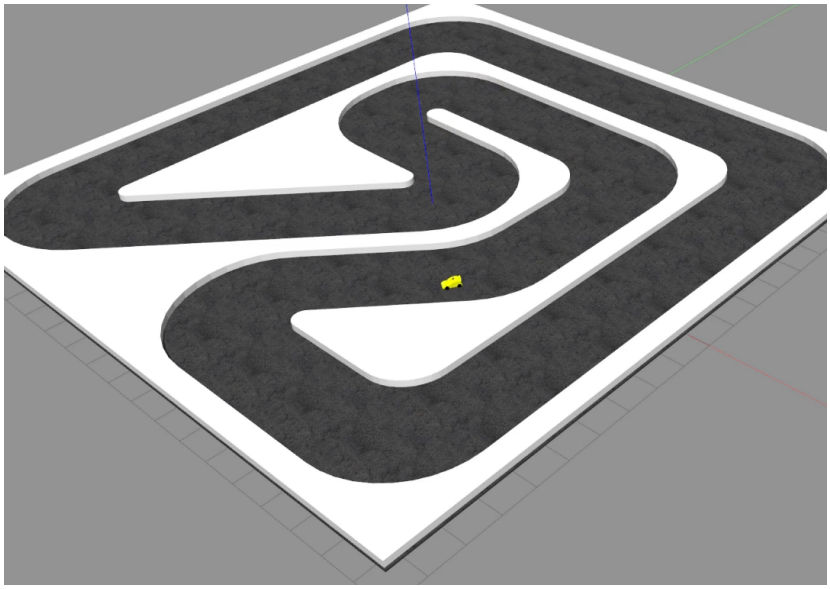


Figure 5.10: The testing track contains several long straights and several challenging turns. The Gazebo simulator allows us to monitor and modify the 3D scene.

5.2.3 Simulator

Due to the hardware issues we ran into and due to other factors, which did not allow further improvements and testing of the vehicle during the spring of 2020, we conducted further experiments only with the Gazebo simulator [34] configured for the F1/10 platform [35]. This change gave us the opportunity to test the algorithm with perfect odometry, but with the same interface between the algorithm and the simulated actuators of the virtual vehicle. To adapt the algorithm for the simulator, we had to modify our steering servo and motor models. We tweaked the parameters of our models to predict the behavior of the simulated vehicle as closely as possible.

The F1/10 simulator repository contains a 3D map[4] with several different types of corners and it is shown in Figure 5.10. Especially the middle part which consists of a series of left-hand turn followed by a right-hand hairpin turn presents a challenge where the car must slow down to avoid collision while keeping as much speed as possible to still reach a good lap time. We ran the full-circuit planning for the modified vehicle model and for this track.

For this experiment, we turned off the obstacle detection and we focused only on the lap times in a track without any obstacles. We tried a reference implementation against our four different combinations of algorithms and for each of them, we let it run for up to 10 consecutive laps. We ran each algorithm three times and we used the run in which it was able to complete all 10 laps and in which it reached the best average lap time.

[4]To view the state of the repository at the time of writing, see `https://github.com/f1tenth-dev/simulator/tree/140f8bd93b22b59d7cf7bfa753e1fdc73a4a8e59`. The repository has changed since then and the map we used has been removed.

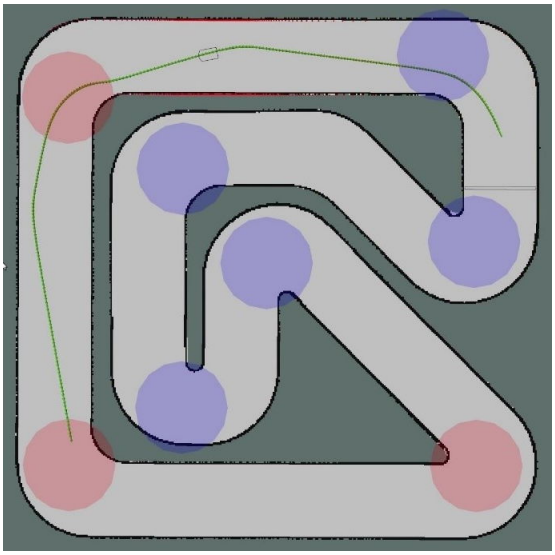


Figure 5.11: This figure captures a planned trajectory and a vehicle on the testing track as it is shown in Rviz. The circles represent the positions of the corners detected by the track analysis algorithm. The circles marked in red are the next three corners ahead of the vehicle. The mostly green line represents the last planned trajectory. Green parts of the trajectory show where the vehicle is expected to drive fast and red parts would show where the vehicle should slow down.

Reference Implementation The F1/10 Simulator repository contains a reference implementation of the Pure Pursuit algorithm which follows a fixed spline path divided sectors marked as "unrestricted", "caution", and "brake", which tell the vehicle how to adjust its speed. We used this implementation as a baseline and we will compare our results against this implementation. This algorithm cannot react to any unexpected obstacles.

SEHS/Hybrid A* and Pure Pursuit The first trajectory following algorithm we tested was the Pure Pursuit controller. We tested different values of the lookahead distance, and we settled on an adaptive look ahead distance linearly proportional to the speed of the vehicle. The lookahead was set to 1.0 m for a stopped vehicle and 1.5 m for the theoretical top speed. Lower lookahead distances caused the vehicle to oscillate and higher lookahead distances often resulted in the vehicle colliding with the apexes of the corners. This algorithm has no mechanism of avoiding unexpected obstacles which were not considered by the planning algorithm and pure pursuit would blindly hit an obstacle if the reference trajectory went through it.

SEHS/Hybrid A* and DWA The other controller we tested was the DWA algorithm. This algorithm has several parameters which we tuned until we reached satisfying results. First parameter is the prediction horizon which determines for how long we predict the movement of the vehicle from the current measured state with a single examined action. Short prediction horizon does not give enough room to reveal differences between all the applicable actions and a long prediction horizon is unrealistic, because we change the action several times per

second and it also eliminates many actions, because fixating one action for too long would likely lead to a collision. We achieved best results by setting the prediction horizon to 0.6 s.

The other parameters are the weights we assign to the error of the position of the vehicle, the error of its heading angle, the error of the speed of the vehicle, and the proximity to the obstacles ahead of the vehicle and nearby obstacles. In the end, the highest weight was assigned to the distance from obstacles and the second largest to the position error.

Results The reference implementation follows a predefined trajectory almost flawlessly and during a test run of 10 consecutive laps it reached an average lap time of 26.410 s with the top speed reaching 4.08 $\mathrm{m\,s^{-1}}$ and the best lap time in this run was 25.911 s. For detailed statistics of lap times and speeds achieved by this algorithm, see Table 5.1.

Our trajectory analysis algorithm correctly detects eight corners in the circuit. We set the planning algorithm to plan for the next 3 corners ahead of it. The planning time on the desktop computer ranged between 100 ms and 300 ms depending on the initial state and on the length and complexity of the circuit segment ahead of the vehicle.

The results of our simulated races can be found in Tables 5.2, 5.3, 5.4, and 5.5. The average lap times and the best lap times are shown in the following table:

Planning alg.	Following alg.	Avg. lap time [s]	Best lap time [s]	Max. speed [$\mathrm{m\,s^{-1}}$]
SEHS	DWA	**25.577**	**23.815**	4.04
Hybrid A*	DWA	25.802	24.295	4.05
SEHS	Pure Pursuit	27.168	25.850	4.03
Hybrid A*	Pure Pursuit	27.484	25.667	4.01

The SEHS algorithm produces trajectories which are more aggressive than the Hybrid A* algorithm. The trajectory is often very close to the boundary of the track. This results in faster lap times than the Hybrid A* counterpart. On the other hand, the SEHS-guided vehicle crashes into a wall more often than when the vehicle follows trajectories produced by Hybrid A*.

If we compare the results with the trajectories planned ahead of time for the whole track as shown in 5.6, especially the lap times of the first rounds when the vehicle starts from standstill, it is clear that the planning algorithm finds trajectories which the following algorithms cannot follow to the full extent. This is most likely due to the imperfections of the kinematic vehicle model. The vehicle cannot go through some of the corners at the planned speed without hitting the outer wall of the track and so it has to brake more. On the other hand, the planned trajectory is not too far off from the actual trajectory and it serves as a good guidance for the vehicle during the race.

Both the SEHS and Hybrid A* algorithms proved to be fast enough and to produce good reference trajectories even for the vehicle as it was moving at its maximum speed in the simulator. The algorithms were able to produce new plans at a rate of up to 10 Hz. The only problem we were experiencing was a drop of

	Lap time [s]	Traveled distance [m]	Average speed [m s^{-1}]	Maximum speed [m s^{-1}]
1.	27.876	84.72	3.25	4.06
2.	26.208	87.07	3.37	4.08
3.	26.381	86.91	3.38	4.07
4.	26.279	86.75	3.36	4.06
5.	26.484	87.00	3.35	4.05
6.	26.172	86.96	3.37	4.07
7.	26.366	87.03	3.37	4.07
8.	26.227	86.54	3.37	4.08
9.	**25.911**	**85.83**	**3.37**	**4.05**
10.	26.199	86.39	3.36	4.06

Table 5.1: The results of ten consecutive laps achieved with the reference implementation of the Pure Pursuit algorithm following a predefined curve. This implementation was taken from the F1/10 Simulator GitHub repository.

	Lap time [s]	Traveled distance [m]	Average speed [m s^{-1}]	Maximum speed [m s^{-1}]
1.	28.915	83.79	3.19	3.97
2.	25.953	88.10	3.43	3.94
3.	25.671	90.88	3.50	4.01
4.	24.872	85.40	3.44	4.01
5.	25.851	86.18	3.35	4.02
6.	25.880	89.87	3.47	4.02
7.	25.102	86.75	3.47	4.04
8.	**23.815**	83.93	3.52	4.01
9.	24.286	85.48	3.53	4.04
10.	25.428	88.06	3.47	4.02

Table 5.2: The results of ten consecutive laps achieved with the combination of the SEHS and DWA algorithms in the Gazebo simulator.

	Lap time [s]	Traveled distance [m]	Average speed [$m\,s^{-1}$]	Maximum speed [$m\,s^{-1}$]
1.	29.746	85.88	3.17	3.94
2.	25.096	87.10	3.53	4.01
3.	24.478	83.99	3.43	4.05
4.	25.569	86.33	3.39	3.92
5.	25.503	86.49	3.39	3.80
6.	27.935	89.02	3.35	3.92
7.	25.404	86.69	3.42	3.82
8.	**24.295**	86.54	3.56	4.03
9.	24.786	85.21	3.45	4.01
10.	25.209	88.85	3.52	4.00

Table 5.3: The results of ten consecutive laps achieved with the combination of the Hybrid A* and DWA algorithms in the Gazebo simulator.

	Lap time [s]	Traveled distance [m]	Average speed [$m\,s^{-1}$]	Maximum speed [$m\,s^{-1}$]
1.	29.814	84.31	3.11	4.03
2.	**25.850**	85.62	3.32	4.02
3.	26.012	87.41	3.38	4.01
4.	26.325	87.24	3.32	4.00
5.	26.845	87.69	3.28	3.95
6.	28.549	90.89	3.25	4.00
7.	26.863	88.27	3.30	4.00
8.	26.520	86.62	3.29	3.95
9.	26.663	87.56	3.30	4.00
10.	28.242	90.68	3.29	4.00

Table 5.4: The results of ten consecutive laps achieved with the combination of the SEHS and Pure Pursuit algorithms in the Gazebo simulator. During the last lap, the vehicle hit a wall and was slowed down.

	Lap time [s]	Traveled distance [m]	Average speed [m s^{-1}]	Maximum speed [m s^{-1}]
1.	31.119	85.38	3.0	4.0
2.	26.569	87.16	3.32	4.0
3.	27.531	88.36	3.23	4.01
4.	26.057	87.11	3.36	3.98
5.	26.071	84.51	3.29	4.0
6.	33.191	100.34	3.27	4.0
7.	26.529	87.19	3.31	4.01
8.	26.070	85.58	3.31	4.0
9.	**25.667**	86.18	3.35	3.99
10.	26.039	85.07	3.29	4.0

Table 5.5: The results of ten consecutive laps achieved with the combination of the Hybrid A* and Pure Pursuit algorithms in the Gazebo simulator. The vehicle hit a wall during the 6-th lap and it got stuck for a brief moment.

the re-planning frequency in the middle section of the track. The algorithms were sometimes not able to find a trajectory in the series of a tight left run followed-up with a hairpin turn. Each of the trajectories the algorithm tested resulted in a collision with the boundary of the track. Most of the time, the car would slow down enough so that the algorithm could find a trajectory to the next waypoint, but sometimes the car would keep following the trajectory to its end and then it would not have anything to follow. The Pure Pursuit algorithm would immediately crash into the road boundary and get stuck. The DWA algorithm on the other hand can continue driving further just by prefering actions which keep the vehicle away from obstacles and road boundaries. Once the vehicle reaches the apex of the hairpin turn, the planning algorithm is able to find a collision-free trajectory to the waypoints ahead of the vehicle and it can continue the race with only a small impact on the lap time. The Hybrid A* algorithm performed slightly better in this regard and it was mostly the SEHS algorithm which occasionally failed to produce a trajectory. This seems to be in line with the issue the SEHS algorithm had with the complex "Zurich" circuit as shown in Figure 5.7.

Both the Pure Pursuit and the DWA algorithms worked well in the simulator after their parameters were tuned. The Pure Pursuit algorithm performed slightly worse than the reference implementation. The DWA algorithm on the other hand reached the fastest lap times of the three algorithms on average and it also managed to reach the fastest lap time of 23.855 s. On top of that, it was more robust thanks to the ability to continue driving forward even without a guiding trajectory and it is therefore the winner of this race.

5.2.4 Obstacle Avoidance

The vehicle detects the obstacles using the readings from the LIDAR scanner and using the Costmap ROS library. The obstacles are marked into the occupancy grid and the planning algorithms and the DWA following algorithm can account for these obstacles. The DWA algorithm filters out any actions which would likely

lead to a collision if they were being applied during its lookahead period. For the remaining actions, it calculates the tracking error of the projected movement of the vehicle and the reference trajectory and the distance from any obstacle marked in the occupancy grid. It therefore prefers actions, which steer the vehicle away from any obstacles.

The Gazebo simulator allows us to place obstacles, such as cubes and cylinders, into the scene. We tried placing these obstacles in the path of the vehicle and we observed how the vehicle reacts to it. When the vehicle observes the obstacle from afar, it usually has no issue going around it, because the trajectory planning algorithm has enough time to account for the obstacle and go around it. During the next lap, the vehicle will already plan its way around the obstacle even when it is still around a corner or two. An example of the vehicle going around obstacles is shown in Figure 5.12 and Figure 5.13.

The most challenging scenario was several obstacles placed one after the other. The vehicle detects the first one but it hits the second one if the planned trajectory goes through the second obstacle. The vehicle often managed to mark the obstacle into the occupancy grid, but the DWA algorithm was often unable to avoid it. A second challenging scenario was a narrow passage between two obstacles. If the obstacles were not marked precisely, which was often the case, the track would either appear to be blocked to the vehicle or the passage as too narrow for the vehicle to align itself with it and fit inside. The planning algorithm would fail to find a trajectory through the passage. The vehicle would then run out of the planned reference trajectory and it would stop or crash into a wall.

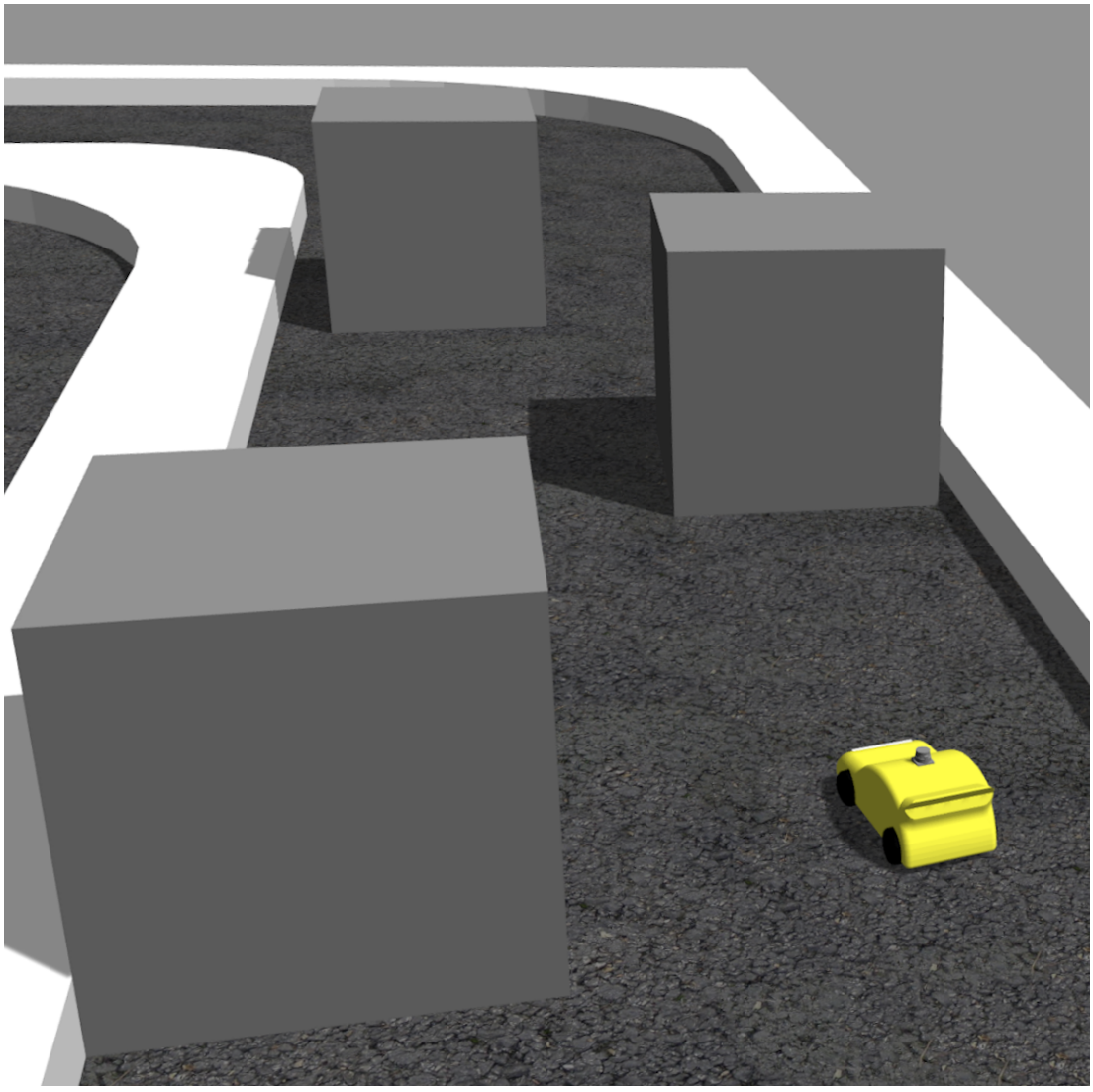


Figure 5.12: The vehicle is passing three cubes in an alternating pattern.

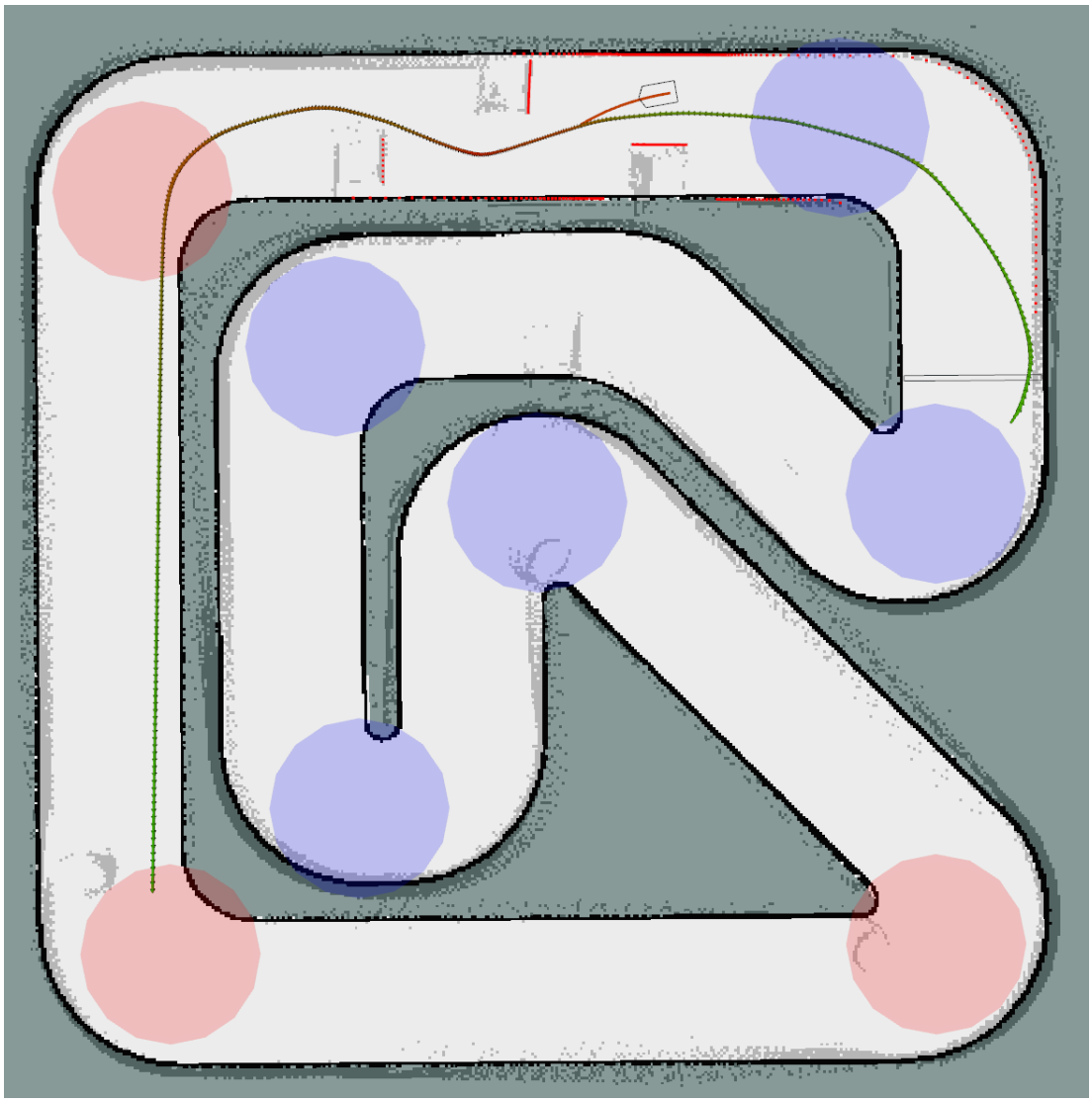


Figure 5.13: This picture shows how the information the vehicle has about its surroundings. The red dots are the readings from the LIDAR, black pixels show the original map of the track and the gray pixels are the detected obstacles which have been marked into the occupancy grid. The planned trajectory already accounted for the obstacles and it guides the vehicle through the narrow passage at a low speed.

Conclusion

In this thesis, we approached the problem of autonomous racing inspired by the F1/10 competition. This problem is the simplification of full-scale autonomous vehicle driving which allowed us to focus solely on a collision-free time-optimal car-like vehicle motion planning without having to take into account other intricacies of the adjacent problems, such as following traffic rules and predicting the behavior of other road users and pedestrians.

Our approach to solving the problem was to create an agent which analyzes the layout of the racing circuit and then plans its motion through the track focusing on just a fixed number of corners ahead of the vehicle. Using the Hybrid A* and the SEHS algorithms, we were able to generate near time-optimal trajectories as the vehicle drives along the racing circuit in almost real-time. Our planning algorithm was able to consider stationary obstacles which were discovered using the sensors of the vehicle in previous laps.

We implemented two trajectory following strategies – the Pure Pursuit controller and the DWA algorithm. Both algorithms enabled the vehicle to move along the planned trajectory, both with their own advantages. The Pure Pursuit algorithm follows the planned trajectory closely, but it follows it blindly and it does not avoid any unexpected obstacles. The DWA algorithm achieves similar comparable results and fast lap times and at the same time it allows the vehicle to avoid any obstacles which are detected as the vehicle moves and which were not considered by the planning algorithm.

We tested our algorithms on a car-like robot we built ourselves inspired by the F1/10 racing platform and on top of the ROS ecosystem of tools and libraries. Although the planning algorithm performed sufficiently even with limited computation power, this test was not a success. Due to the unreliable state estimation caused by the limited capabilities of the sensors, we were able to reach only very limited speeds at which we were able to move around the circuit. At higher speeds the vehicle would quickly lose track of its correct location and it would crash into a wall. When we eliminated the noisy data from the sensors and moved to a simulated environment, the vehicle was able to reach much higher speeds and reliably complete several laps of the circuit in a row.

The experiments we conducted in a simulator show that our approach is viable. We are able to plan trajectories for the next three corners ahead of the vehicle at a frequency of several hertz and achieve good results on a test circuit. The vehicle also has a basic ability to detect and avoid unexpected obstacles. The most obvious shortcoming is the imprecise kinematic vehicle model. If we were able to predict the movement of the vehicle more reliably, the trajectories the planning algorithm produces would be easier to track and we would be able to navigate the vehicle through corners more safely without hitting the outer boundary of a corner. We could achieve further improvements by tuning the discretization parameters of the algorithms and the parameters and weights of the Pure Pursuit and DWA algorithms. It would be interesting to try the planning algorithm for a different F1/10 vehicle with reliable odometry and with identified parameters of a tire model. Since our code uses the standard ROS topic types and the standard transformation tree, the adaptation of our ROS nodes should be possible.

Bibliography

[1] Pete Thomas, Andrew Morris, Rachel Talbot, and Helen Fagerlind. Identifying the causes of road crashes in Europe. *Annals of advances in automotive medicine / Annual Scientific Conference ... Association for the Advancement of Automotive Medicine. Association for the Advancement of Automotive Medicine. Scientific Conference*, 57:13–22, 2013.

[2] Kevin Dowling, R Guzikowski, J Ladd, Henning Pangels, Sanjiv Singh, and William (Red) L Whittaker. NAVLAB: An Autonomous Navigation Testbed. Technical Report CMU-RI-TR-87-24, Carnegie Mellon University, Pittsburgh, PA, nov 1987.

[3] Janosch Delcker. The man who invented the self-driving car (in 1986), 2018.

[4] Dean A Pomerleau. Advances in Neural Information Processing Systems 1. chapter ALVINN: An Autonomous Land Vehicle in a Neural Network, pages 305–313. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989.

[5] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to End Learning for Self-Driving Cars. *CoRR*, abs/1604.07316, 2016.

[6] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. In *Proceedings of the National Conference on Artificial Intelligence*, pages 343–349, 1999.

[7] Dieter Fox. KLD-Sampling: Adaptive Particle Filters and Mobile Robot Localization. 2001.

[8] S J Julier and J K Uhlmann. Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, 92(3):401–422, mar 2004.

[9] Christopher Urmson, Joshua Anhalt, Hong Bae, J Andrew (Drew) Bagnell, Christopher R Baker, Robert E Bittner, Thomas Brown, M N Clark, Michael Darms, Daniel Demitrish, John M Dolan, David Duggins, David Ferguson, Tugrul Galatali, Christopher M Geyer, Michele Gittleman, Sam Harbaugh, Martial Hebert, Thomas Howard, Sascha Kolski, Maxim Likhachev, Bakhtiar Litkouhi, Alonzo Kelly, Matthew McNaughton, Nick Miller, Jim Nickolaou, Kevin Peterson, Brian Pilnick, Raj Rajkumar, Paul Rybski, Varsha Sadekar, Bryan Salesky, Young-Woo Seo, Sanjiv Singh, Jarrod M Snider, Joshua C Struble, Anthony (Tony) Stentz, Michael Taylor, William (Red) L Whittaker, Ziv Wolkowicki, Wende Zhang, and Jason Ziglar. Autonomous driving in urban environments: Boss and the Urban Challenge. *Journal of Field Robotics Special Issue on the 2007 DARPA Urban Challenge, Part I*, 25(8):425–466, jun 2008.

[10] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabriel Hoffmann,

Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, and Sebastian Thrun. Junior: The Stanford Entry in the Urban Challenge. *Journal of Field Robotics*, 25:569–597, 2008.

[11] Y Kuwata, G A Fiore, J Teo, E Frazzoli, and J P How. Motion planning for urban driving using RRT. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1681–1686, 2008.

[12] University of Pennsylvania. F1/tenth.

[13] Martin Vajnar. Model car for the F1/10 autonomous car racing competition. Master thesis, Czech Technical University in Prague.

[14] Jan Dusil. Slip detection for F1/10 model car. Bachelor thesis, Czech Technical University in Prague.

[15] Jaroslav Klapálek. Dynamic obstacle avoidance for autonomous F1/10 car. Master thesis, Czech Technical University in Prague.

[16] Alexander Heilmeier, Alexander Wischnewski, Leonhard Hermansdorfer, Johannes Betz, Markus Lienkamp, and Boris Lohmann. Minimum curvature trajectory planning and control for an autonomous race car. *Vehicle System Dynamics*, 0(0):1–31, 2019.

[17] Stahl, Tim, Wischnewski, Alexander, Betz, Johannes, and Lienkamp, Markus. Ros-based localization of a race vehicle at high-speed using lidar. *E3S Web Conf.*, 95:04002, 2019.

[18] Alexander Wischnewski, Tim Stahl, Johannes Betz, and Boris Lohmann. Vehicle dynamics state estimation and localization for high performance race cars**research was supported by the basic research fund of the institute of automotive technology of the technical university of munich. *IFAC-PapersOnLine*, 52(8):154 – 161, 2019. 10th IFAC Symposium on Intelligent Autonomous Vehicles IAV 2019.

[19] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[20] Steven M LaValle. Rapidly-Exploring Random Trees: A New Tool for Path Planning. Technical report, 1998.

[21] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, sep 1975.

[22] Sertac Karaman and Emilio Frazzoli. Incremental Sampling-based Algorithms for Optimal Motion Planning. *CoRR*, abs/1005.0416, 2010.

[23] Vojtech Vonasek, Jan Faigl, Tomáš Krajník, and Preucil Libor. A Sampling Schema for Rapidly Exploring Random Trees Using a Guiding Path. 2011.

[24] C A Rosen and N J Nilsson. Application Of Intelligent Automata to Reconnaissance. Technical report, Stanford Research Institute, nov 1966.

[25] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, and James Diebel. Practical Search Techniques in Path Planning for Autonomous Driving. In *Proceedings of the First International Symposium on Search Techniques in Artificial Intelligence and Robotics (STAIR-08)*, Chicago, USA, jun 2008. AAAI.

[26] J Petereit, T Emter, C W Frey, T Kopfstedt, and A Beutel. Application of Hybrid A* to an Autonomous Mobile Robot for Path Planning in Unstructured Outdoor Environments. In *ROBOTIK 2012; 7th German Conference on Robotics*, pages 1–6, may 2012.

[27] Chao Chen. *Motion Planning for Nonholonomic Vehicles with Space Exploration Guided Heuristic Search.* Dissertation, Technische Universit{ä}t M{ü}nchen, M{ü}nchen, 2016.

[28] R Rajamani. *Vehicle Dynamics and Control.* 01 2006.

[29] Carlo Ravizzoli. Identification and control of an rc car for drifting purposes. Master thesis, Politecnico di Milano.

[30] H.B. Pacejka and I.J.M. Besselink. Magic formula tyre model with transient properties. *Vehicle System Dynamics*, 27(S1):234–249, 1997.

[31] Filippo Consoli. Identification and validation of a sensored radio controlled drifting car. Master thesis, Politecnico di Milano.

[32] R Craig Conlter. Implementation of the Pure Pursuit Path Tracking Algorithm, 1992.

[33] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The Dynamic Window Approach to Collision Avoidance. *Robotics & Automation Magazine, IEEE*, 4:23–33, 1997.

[34] Open Source Robotics Foundation. Gazebo - Robot simulation made easy. `http://gazebosim.org/`, 2020.

[35] Varundev Suresh Babu and Madhur Behl. ROS F1/10 autonomous racecar simulator. October 2019.

[36] Matthew O'Kelly, Varundev Sukhil, Houssam Abbas, Jack Harkins, Chris Kao, Yash Vardhan Pant, Rahul Mangharam, Dipshil Agarwal, Madhur Behl, Paolo Burgio, and Marko Bertogna. F1/10: An Open-Source Autonomous Cyber-Physical Platform. *CoRR*, abs/1901.0, 2019.

[37] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf. A flexible and scalable SLAM system with full 3d motion estimation. In *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE, November 2011.

Acronyms

AMCL Adaptive Monte Carlo Localization. 103, 105

DC Direct Current. 93, 96, 107

DWA Dynamic Window Approach. 77, 80, 83, 84, 87, 106

EKF Extended Kalmann Filter. 103

ESC Electronic Speed Controller. 93

IMU Inertial Measurement Unit. 5, 10, 95, 96, 100, 103

LIDAR Light Detection And Ranging. 21, 40, 76, 77, 84, 95–97, 100, 101, 103

PID Proportional-integral-derivative controller. 63, 64, 106

PWM Pulse Width Modulation. 24, 50, 51, 93, 95, 96, 106, 107

RC Radio-controlled. 93, 96

REP ROS Enhancement Proposal. 100

ROS Robot Operating System. 21, 69, 77, 87, 98–103, 106, 108

RPM Revolutions Per Minute. 24, 52, 53, 68

RRT Rapidly-Exploring Random Trees. 28, 30, 31, 39

RRT* Optimal RRT. 30, 31

SEHS Space Exploration Guided Heuristic Search. 39, 40, 43, 68, 69, 80, 83, 87, 109

SLAM Simultaneous Localization And Mapping. 101

A. Experimental Vehicle

In order to verify the approach we discuss in this theses, we needed a car-like robot which we could use for real-world testing. We took inspiration from the F1/10 competition [36], the MIT RACECAR[1], AutoRally[2], and other similar projects and we built a robot based on a chassis from an 1:10 scale Radio-controlled (RC) car with an on-board computer and a set of sensors. In this chapter, we will describe the hardware components we used to build a custom experimental fast moving car.

A.1 Chassis

We used chassis, steering servo, Direct Current (DC) motor, Electronic Speed Controller (ESC), and a ratio transmitter and receiver from an off-the-shelf RC car. We removed the plastic cover and some unnecessary parts attached to the chassis and we attached a thin plywood board on top of the chassis. We later mounted all the electronics to the plywood board. Even though the weight of the battery and of the electronics is not negligible, the suspension of the vehicle is stiff enough to carry the weight without any significant roll and pitch changes during acceleration and cornering.

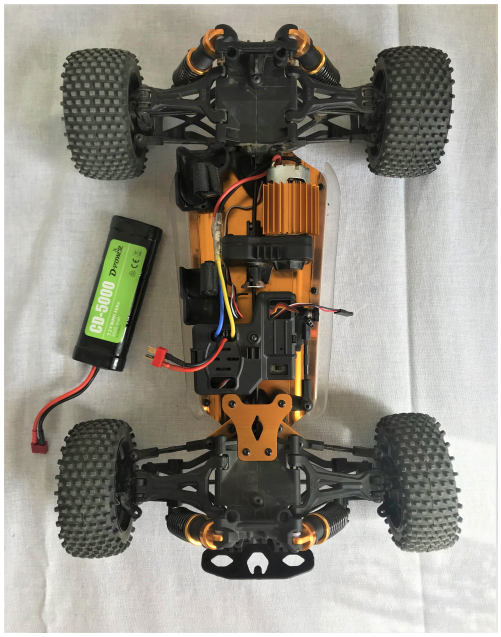
The chassis has the same geometry as a common passenger car with two fixed rear wheels and two front wheels with Ackermann steering geometry. All wheels are driven by the DC motor through a fixed gearbox. The wheels on both front and rear axes can move independently thanks to two differentials, one on each axis. The differentials are necessary for smooth travel during turns, when the outer wheel spins faster than the inner wheel. Photos of the vehicle are shown in Figure A.1 and the our measurements of the chassis are stated in Table A.1.

The steering servo and the ESC, which controls the speed of the DC motor, both have a 3-pin connector which was originally connected to a radio receiver. Two of the pins connect to a DC power supply and the third connects to a signal wire. The signal is originally created by an radio transmitter which generates a PWM signal. The PWM signal consists of pattern resembling a square wave,

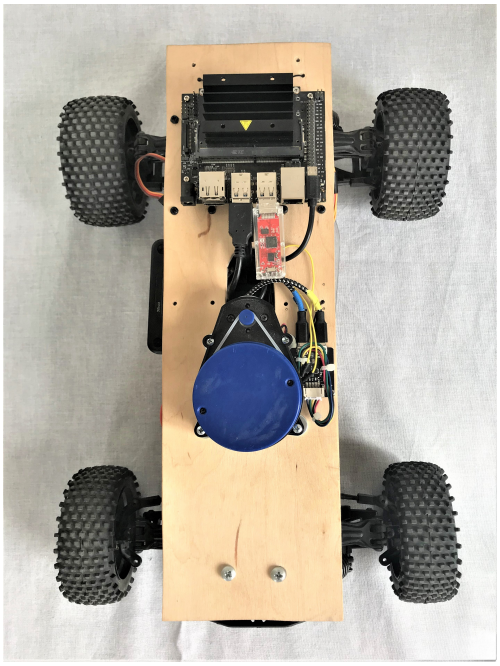
[1] `https://mit-racecar.github.io/`
[2] `https://autorally.github.io/`

Property name	Symbol	Measured value
Wheelbase	L	0.31 m
Center of gravity offset	l_r, l_f	0.155 m
Width	W	0.3 m
Length	H	0.45 m
Wheel radius	r_w	0.05 m
Gear ratio (motor to wheels)	G	9
Reachable steering angles	δ	$[-21.16°, 26.57°]$

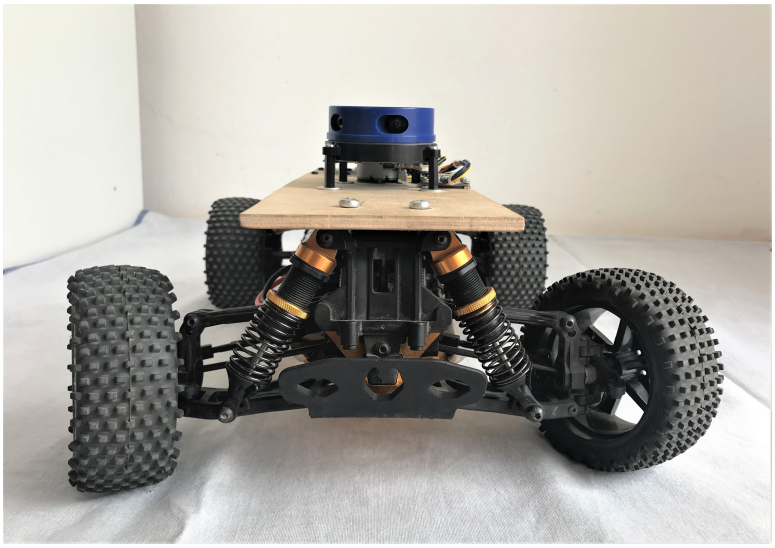
Table A.1: Properties of the chassis of the experimental vehicle.



(a) A photo of the chassis of the vehicle without any additional electronics mounted to it.



(b) A photo of the chassis with the plywood board with all the electronics attached to it. Some of the components are not visible because they are attached to the bottom side of the board.



(c) Front view.

Figure A.1: Photos of the experimental vehicle.

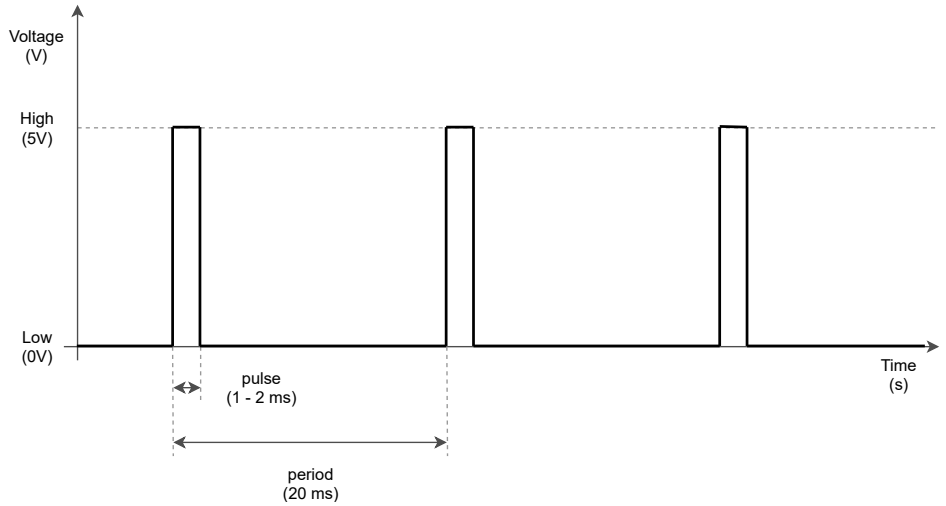


Figure A.2: The PWM signal for the steering servo and the ESC form a square wave with the period of 20 ms and pulse width between 1 ms and 2 ms.

where the voltage measured on the signal wire is high (5 V) for some period of time and then the voltage drops to low (0 V) for the rest of the period. The pulse of high voltage lasts between 1 ms and 2 ms and the period of the signal is 20 ms as it is shown in Figure A.2. We can therefore connect these connectors to a computer which generates appropriate signals to steer the vehicle. The 20 ms period of one PWM cycle gives us an upper limit of 50 Hz at which we can change the commands for the vehicle.

A.2 Sensors

The vehicle uses on-board sensors to track its location on the racing track. We use a combination of three sensors: a LIDAR, an IMU, and a motor encoder. We tried several different sensors from different manufacturers, including `Razor 9DoF IMU` and `Scanse Sweep 2D LIDAR`, but these devices did not yield good results.

Hall Effect Encoder consists of an 8-pole magnetic disc which we attach to the drive shaft of the vehicle, and a Hall effect sensor. The sensor sends a pulse every time it detects a change from a north pole to a south pole or vice-versa. This way we can detect that the motor has made one eigth of a revolution. By counting the number of revolutions and by assuming that the wheels of the vehicle roll perfectly against the road surface, we can estimate the distance that the vehicle traveled. By combining this information with the current steering angle of the front wheels, we can estimate the movement of the vehicle and use it as a source of odometry. We used a "Wheel Encoder Kit from DAGU" from SparkFun[3].

Bosch BNO055 USB Stick is an IMU which contains a triaxial accelerometer, a triaxial gyroscope, a triaxial magnetometer and a thermometer[4]. We use

[3] `https://www.sparkfun.com/products/12629`

[4] `https://www.bosch-sensortec.com/bst/products/all_products/bno055`

the measurements of the gyroscope and the accelerometer to determine the acceleration of the vehicle to improve our odometry from the motor encoder, which cannot detect wheel skidding.

YDLIDAR X4 is a low-cost 360° LIDAR laser scanner. We use it to determine the distance to the surrounding obstacles. This sensor has a range of up to 10 m and it takes 5000 samples every second while spinning at the frequency of almost 12 Hz. This sensor is connected to a computer via a micro-USB port and it is connected to a 5 V power supply through a second micro-USB port.

A.3 On-board Computer

The brains of our autonomous vehicle is the NVIDIA Jetson Nano[5]. This board contains a quad-core ARM A57 with the clock frequency of 1.43 GHz, 4 GB of LPDDR4 RAM and a 128-core Maxwell GPU. This computer is powered through a Micro-USB port with 5 V/2 A[6]. The Jetson has 4 USB ports which are used to connect the LIDAR, the IMU, and two Arduino boards. The board does not include a Wi-Fi antena and therefore we added an Intel Dual Band Wireless-Ac 8265 W/Bt card to the M.2 slot of the board in order to receive telemetry while the vehicle is driving along a circuit.

A.3.1 Microcontrollers

We use two Arduino Nano boards[7] as an interface between the actuators, the radio receiver, and the motor encoder, and the Jetson Nano board.

Hall Effect Encoder is connected to an interrupt pin of one Arduino and this Arduino counts the number of revolutions of the motor. This board also provides power to the Hall sensor.

ESC and Servo are connected to PWM output pins of the other Arduino board and the radio receiver is connected to two interrupt pins. This board contains logic which converts high-level commands for the actuators into corresponding PWM signals. It also contains a safety mechanism which disables autonomous mode of the vehicle and allows the supervisor to take over control of the vehicle when he or she uses the remote controller.

A.4 Power Supply

The DC motor and the servo are powered by 7.2 V 5000 mA h NiMH battery which was included with the RC car. The rest of the electronics of the vehicle is powered by a regular 20 000 mA h power bank which has two output USB ports.

[5] `https://developer.nvidia.com/embedded/jetson-nano-developer-kit`

[6] The board can operate with a lower power consumption of 5 W in a mode in which two of the CPU cores are turned off.

[7] `https://www.arduino.cc/en/Guide/ArduinoNano`

One of these ports can supply 2 A and the Jetson Nano board is connected to it, the other port can supply 1 A and it powers the LIDAR.

This setup ensures that the power in the batteries supplying the motors of the vehicle discharge much sooner than the power bank which supplies the computer and sensors if we start with fully charged batteries. It should never occur that the computer shuts down while the car is being driven autonomously and it will not continue moving uncontrollably. We also do not need to build a custom power delivery board.

B. Technical Documentation

B.1 Attachments

This thesis is accompanied with a number of file attachments. These files include the C++ source code of the algorithms described in this thesis, several Python scripts which were used to identify the parameters of the servo and motor models, and code for integration with ROS and with the hardware of the experimental RC vehicle and with the F1/10 Gazebo simulator. The files are organized in the following directory structure:

```
/
├── arduino ..... C code for two Arduino microcontrollers
│   ├── steering_node
│   └── motor_shaft_encoder
├── catkin_ws ..... ROS Catkin workspace of the Racer project
│   └── src
│       ├── racer ..... our main ROS package
│       │   ├── circuits ..... circuit definiton files (YAML)
│       │   ├── include
│       │   │   ├── racer ..... C++ code under the racer namespace
│       │   │   └── racer_ros ..... code for integration with ROS (C++)
│       │   ├── launch ..... custom launch files
│       │   ├── maps ..... maps recorded using SLAM (YAML+PGM)
│       │   ├── scripts ..... ROS nodes for debugging and telemetry (Python)
│       │   └── src ..... ROS nodes for the racing agent (C++)
│       ├── racer_msgs ..... custom ROS messages
│       ├── racer_sensors ..... sensor adjustments and odometry
│       └── racer_simulator ..... integration with the F1tenth simulator
│           ├── launch ..... launch and Rviz configuration files
│           └── src ..... ROS nodes for integration with the simulator (Python)
├── experiments
│   ├── algorithm_testing .... testing the algorithms outside of ROS (C++)
│   ├── model_fitting ..... identification of actuator models (Python)
│   │   ├── motor_rpm
│   │   ├── servo_pwm_to_steering_angle
│   │   └── servo_setting_time
└── media
    ├── experimental_vehicle_gym ..... Videos of the self-driving RC car
    └── simulator ..... Screenshots from Rviz and Gazebo
```

In rest of this chapter, we will go over the different parts of the project and explain how to use them and how to work with them. We will first describe how to use the "standalone" experiments in the `experiments` directory and later we describe the ROS project located in the `catkin_ws` directory and we go over the steps necessary to build and run the project in the simulator and on custom hardware. Finally, we will describe the ideas behind the C++ code of the algorithms located in the `catkin_ws/src/racer/include/racer` directory.

B.2 Track Analysis and Trajectory Planning

In order to test the track analysis algorithm from Section 3.4 and the SEHS and Hybrid A* trajectory planning algorithms described in Chapter 3, we wrote two C++ programs to test these algorithms on different inputs and to visualize their outcomes.

B.2.1 Requirements and Compilation

In order to compile and run these test programs, we recommend using a PC running a Linux distribution. Both programs were also successfully tested on Microsoft Windows 10 with *Windows Subsystem for Linux* (WSL) 2 and the *X Window System*, such as *Xming X Server*[1], installed. The programs are written using C++17 and they are meant to be compiled using the G++ 9.2 compiler[2].

To visualize the outputs of the algorithms, we use the *Matplotib* library and its C++ binding library `matplotlib-cpp`. The source code of this library is available in a public Git repository under the MIT licence hosted on GitHub[3]. In order to set up this library, install the dependencies of the library first[4] and then clone the repository using `git`[5] and change the repository to the version needed in our code:

```
> cd /experiments/algorithm-testing/include
> git clone https://github.com/lava/matplotlib-cpp
> cd matplotlib-cpp
> git checkout d612b524e10ebdd43d3a8889a95e84c017ad65af
```

To compile and link the source code into executable binaries, use the prepared `Makefile`:

```
> cd /experiments/algorithm-testing
> make
```

This script will create a new subdirectory `bin` and place the newly created binaries in there.

The C++ code includes header files from the `catkin_ws/src/racer/include` directory. If you make adjustments to the directory structure, compilation might fail because `g++` will not be able to find the included files. In such case, edit the `Makefile` and fix the paths of the `-I` options to match your directory structure.

B.2.2 Usage

Track Analysis The track analysis algorithm is easy to test for a circuit:

[1] `http://www.straightrunning.com/XmingNotes/`

[2] `https://gcc.gnu.org/gcc-9/`

[3] To see the code of the libraryt at the time of writing, see `https://github.com/lava/matplotlib-cpp/tree/d612b524e10ebdd43d3a8889a95e84c017ad65af`. Our code might not be compatible with newer revisions of the library.

[4] The instructions are available at `https://github.com/lava/matplotlib-cpp/tree/d612b524e10ebdd43d3a8889a95e84c017ad65af#installation`.

[5] `https://git-scm.com/`

```
> ./bin/track_analysis <circuit-definition-file>
```

The program will output the time each of the steps of the analysis takes and finally it will open a window with the map of the circuit and with marked corners which were detected.

Trajectory Planning To test the trajectory planning algorithm, run this command:

```
> cd /experiments/algorithm-testing
> ./bin/planning_benchmark <n> <timeout> <circ-1> <circ-2>...
```

The parameter `<n>` determines how many times each algorithm with a given set of parameters for a given circuit will be run in order to calculate the average search time. The parameter `<timeout>` determines the time limit for a single search in milliseconds. At least one path to a circuit definition file has to be provided, but the binary accepts multiple of them and it will loop over all of them and execute the search for each of them with all the combinations of the parameters.

The program tests both Hybrid A* and the SEHS algorithms with several different combinations of discretization parameters. The output of the program will be a comma separated values file with the summaries of each test. After each algorithm is tested with each different set of parameters, the found trajectory and the speed profile is shown in a pop-up window.

B.2.3 Circuit Definition File Format

Each circuit is defined with a simple text file with the following structure:

```
<pgm-file>
<occupancy-grid-resolution>
<ix> <iy> <itheta>
<c1x> <c1y>
<c2x> <c2y>
```

The first line contains the path to the occupancy grid file relative to the circuit definition file. The second line contains a decimal number specifying the resolution of the occupancy grid in meters (typical value is 0.05). The third value describes the initial position and orientation of the vehicle. The rest of the file form lines of x and y coordinates of checkpoints along the circuit. At least two checkpoints are expected, but there is no hard upper limit. Examples of circuits can be found in the `/experiments/algorithm_testing/test-inputs` directory.

PGM File Format Limitations The occupancy grid is loaded from a bitmap stored in a PGM file format[6]. We support only the *Plain PGM format* with the magic number `P2` and the pixels represented by ASCII encoded decimal numbers. The numbers must be between 0 and 127. Some bitmap editors, such as GIMP, produce images with pixel values outside of this range, and these files need to be edited manually before they can be supplied to the test programs.

[6] `http://netpbm.sourceforge.net/doc/pgm.html`

B.3 Actuator Models

This section describes the scripts we wrote in order to fit the parameters of the actuator models as described in Section 3.5.1.

B.3.1 Requirements

In order to open and run the actuator model fitting scripts, you need to install Python 3.6[7] and the Jupyter Notebook[8]. The scripts also need the following Python libraries installed on your computer:

- Matplotlib[9]
- Pandas[10]
- Numpy[11]
- SciPy[12]

B.3.2 Usage

To open the scripts, start the Jupyter notebook:

```
> cd /experiments/model_fitting
> jupyter notebook
```

Your browser should open with the Jupyter interface. From here, you can open one of the three subdirectories with the experiments: `motor_rpm`, `servo_pwm_to_steering_angle`, and `servo_setting_time`. Each directory contains a `.ipynb` script which you can view and re-run and the data with the measurements.

B.4 ROS Project

ROS is an open-source set of libraries and tools built on top of Linux. Processes running under this operating system are referred to as nodes and they can be distributed across multiple computers connected over a wired or wireless network or through a serial port. Nodes can be programmed in many different programming languages but the most common and officially supported languages are Python and C++.

Nodes communicate between each other using a publisher/subscriber pattern. A node can advertise any number of topics through which it publishes its outputs in a form of messages. It can also subscribe to topics advertised by other nodes and it can work with the received messages as its outputs. This way it is possible

[7] `https://www.python.org/`
[8] `https://jupyter.org/`
[9] `https://matplotlib.org/`
[10] `https://pandas.pydata.org/`
[11] `https://numpy.org/`
[12] `https://www.scipy.org/`

to create a complex system by combining many small and simple specialized nodes.

Each topic has an assigned message type which defines the structure of the data. There is a large number of standard messages which are used by authors of public libraries. This way it is often possible to replace one library with a different one without additional changes to other nodes. This can be useful for example when one sensor is replaced by a device of the same type but from a different vendor whose ROS node publishes the same standard message type. Programmers can also define their custom message types which are better suited for their application.

ROS comes with a variety of tools for debugging and visualization of the data passed through the topics. One of these tools is `Rviz`. It is a convenient way of visualizing the current state of the system. We use this tool to visualize the telemetry data from the vehicle on a laptop while it is driving along the racing circuit.

We use the latest stable release of ROS at the time of writing this thesis named "Melodic Morenia" which is compatible with Ubuntu 18.04 LTS. All the installation instructions assume that the user is running this version of Ubuntu as their operating system. Further information about ROS and tutorials which describe how to work with it are available on the official website of the ROS project[13].

B.4.1 Architecture of the Distributed System

The setup of our robot consists of the NVIDIA Jetson Nano board which is mounted on the vehicle, two Arduinos which are connected to the Jetson through a serial port, and a notebook which is connected to Jetson over Wi-Fi. Jetson acts as a ROS master machine, Arduinos act as an interface between Jetson and low-level hardware, and the notebook is used only to monitor telemetry data. The graph of this architecture and the connected sensors is depicted in Figure B.1.

The communication between the Arduinos and the Jetson takes place over USB. This is possible thanks to the `rosserial` package[14]. The Arduinos need to include the `ros_lib` library which allows it to subscribe to topics and to publish messages. On Jetson, the communication on each serial port is handled by an instance of a `rosserial_python` node.

Coordinate Frames

ROS includes a very useful way of keeping track of the relative position between different parts of the robot and the position of the robot itself on a map. The package which provides this functionality is called `tf`[15]. `tf` maintains a tree structure of coordinate frames and it keeps track of transformations between these coordinate frames as they change over time. The updates to the transformations are shared through a regular ROS topic called `tf` of the `tf/tfMessage`[16] type. The `tf` package provides libraries for both Python and C++ which make

[13] `https://www.ros.org`

[14] `http://wiki.ros.org/rosserial`

[15] `http://wiki.ros.org/tf`

[16] `http://docs.ros.org/kinetic/api/tf/html/msg/tfMessage.html`

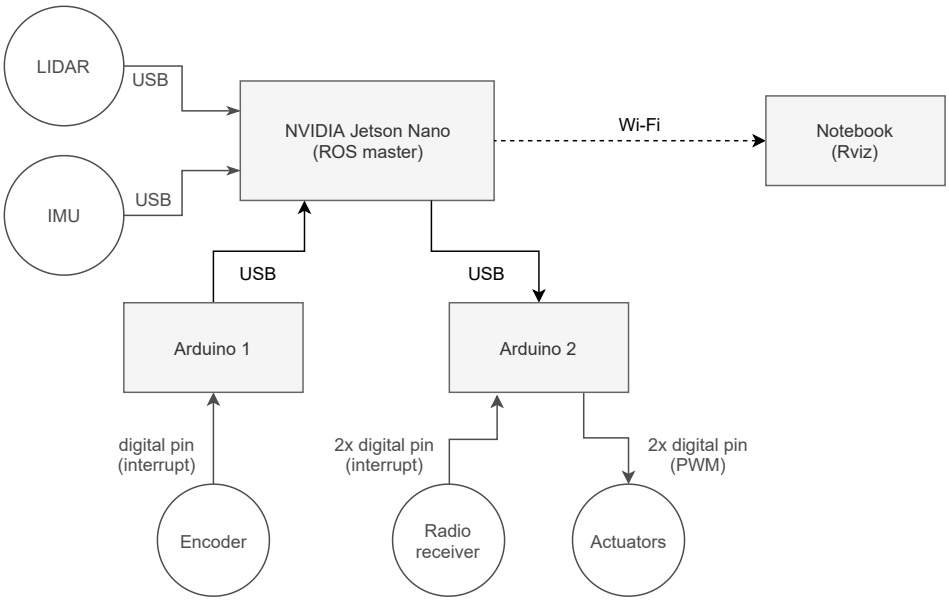


Figure B.1: This figure shows how the computers and sensors which make up the experimental vehicle are connected with each other and the direction of data flow between the devices.

accessing the current state of the whole system and manipulating with the linear transformations easy.

Our implementation adheres to the standard naming convention of coordinate frames as it is defined in ROS Enhancement Proposal (REP) 105[17]. The vehicle has a `base_link` frame attached to it. Each of the sensors has their own coordinate frame and there is a static transformation between the sensor and the base link using the `static_transform_publisher` node from the `tf` package. The IMU gives us the current roll and pitch of the vehicle. We use the `hector_imu_attitude_tf` node[18] to publish this transformation between `base_link` and `base_footprint`, which is the "stabilized" coordinate frame. The root ("fixed") `tf` frame is called `map`. The transformation between the `map` frame and the `base_footprint` represents the pose of the vehicle on the map. This transformation is calculated and published through different nodes in the mapping and racing tasks and we will come back to it later.

Sensors

Each of the sensors connected through USB uses their own drivers for communication over the serial link. Luckily, open-source ROS packages for both the LIDAR and the IMU are available and both of them publish the data in the form of standard `sensor_msgs/LaserScan`[19] and `sensor_msgs/Imu`[20] messages.

The data from the motor shaft encoder is published as a `std_msgs/Float64` message and it contains the number of revolutions since the Arduino was last powered or reset.

[17] `https://www.ros.org/reps/rep-0105.html`
[18] `https://github.com/tu-darmstadt-ros-pkg/hector_slam/tree/catkin`
[19] `http://docs.ros.org/melodic/api/sensor_msgs/html/msg/LaserScan.html`
[20] `http://docs.ros.org/melodic/api/sensor_msgs/html/msg/Imu.html`

The raw data we receive from the LIDAR and the IMU are not readily usable. We had to introduce a layer of preprocessing for both these sensors:

1. The coordinate frame of the IMU is not aligned with the coordinate frame of the vehicle. Even though it should be possible, we were not able to get correct roll and pitch transformations using the `static_transform_publisher` node of the `tf` package. To overcome this issue, we implemented a custom node in C++ called `fix_imu` in the `racer_sensors` package, which publishes the correct transformation. We also set positive values on the diagonals of covariance matrices to make the IMU message valid, because the ROS package for the IMU we use `bno055_usb_stick`[21] does not set them correctly. The configuration for the IMU is stored in `/catkin_ws/src/racer/launch/imu.launch`.

2. The LIDAR is rigidly attached to the vehicle and so the laser scan is affected by the roll and pitch of the vehicle. We use `LaserScanBoxFilter` from the `laser_filters` package[22] to remove points which are too close to the ground (the vehicle might see the ground when it is slowing down or braking) and which are too high above the ideal plane (the vehicle might see the ceiling or over the obstacles when it accelerates). The configuration for the LIDAR is stored in `/catkin_ws/src/racer/launch/ydlidar.launch`.

B.4.2 Mapping

In order to race, our vehicle needs a map of the racing track. The robot can create its own map using an algorithm called Simultaneous Localization And Mapping (SLAM). We use a library called Hector SLAM created by a team from the Technical University of Darmstadt.

To make things simple, we steer the vehicle manually using its remote controller. The robot collects data from the LIDAR as it moves along the track and it performs scan matching between the LIDAR scans and a 2D occupancy grid which it has built so far. For an already built part of the map and a LIDAR scan consisting of n points, the task of the scan matching algorithm is to find a rigid body transformation $\xi = (p_x, p_y, \phi)$, which gives best alignment with the already built map. The authors formulate this problem as an optimization problem for solving in [37]:

$$\xi^* = \arg\min_{\xi} \sum_{i=1}^{n} \left[1 - M\left(S_i\left(\xi\right)\right)\right]^2,$$

where $S_i\left(\xi\right)$ are the world coordinates of scan point s_i after applying the transform ξ and the function M gives the value of the occupancy grid at the given coordinates. After obtaining the transformation, the laser scan is aligned with the occupancy grid and each endpoint of a laser ray is marked as an obstacle into the occupancy grid.

Hector SLAM provides a ROS node `hector_mapping` which subscribes to a topic providing `sensor_msgs/LaserScan` messages and publishes its output in a

[21] `https://github.com/yoshito-n-students/bno055_usb_stick`
[22] `http://wiki.ros.org/laser_filters`

topic of type `nav_msgs/OccupancyGrid`[23]. The node also publishes a `tf` transformation between the `map` frame and the `base_footprint` frame. An important parameter is the resolution of the occupancy grid which we chose to be 5 cm. When the mapping is complete, the final map can be saved into a file using a `map_saver` node from the `map_server` package[24]. We prepared a ROS launch file with the configuration of all nodes necessary for this task. This configuration is included in the attached files in `/catkin_ws/src/racer/launch/create_map.launch`.

B.4.3 Racing

Racing is the most important and the most complicated part of the implementation. It can be split into three groups of ROS nodes: odometry and localization, agent behavior, and the hardware interface.

Map

We use `map_server` node of the `map_server` package to serve the previously recorded occupancy grid to the other nodes. The map is stored in two files: a `.yaml` file with map metadata and a `.pgm` file with the occupancy grid stored as a bitmap. It is necessary that the `.yaml` configuration file contains a correct relative path to the bitmap file. The map server publishes a topic called `map` of type `nav_msgs/OccupancyGrid`. This node also provides a service for one-time retrieving the map without subscribing to the topic.

Costmap In order to prevent collisions, we use a ROS package `costmap_2d`[25] to inflate the areas around obstacles according to a bounding rectangle around the vehicle. The result of this node is another `nav_msgs/OccupancyGrid` topic with cell values marking the "cost" of each cell instead of the regular ternary status (free/obstacle/unknown). The higher the cell cost is, the closer the cell is to an obstacle. Values above certain threshold value are considered "lethal" and if the center of the vehicle were in this position it would be colliding with an obstacle. Collision detection is then just a matter of a single query into the occupancy grid to check if the cell which contains the center point of the vehicle.

This `costmap_2d` can also work directly with the laser scan from the LIDAR and overlay it with the map using the latest `tf` transformation between the `map` and `lidar` coordinate frames and mark obstacles which were not present during mapping into the resulting occupancy grid. Unfortunately, this feature was flaky and we had to disable it in the final version of our configuration when we tested it on the RC car. The laser frame would often be misaligned with the original map for a brief moment and it would mark certain areas of free space as obstacles. The "clearing" feature, which is supposed to remove obstacles from the costmap based on new laser scans, did not work properly and the map soon became filled with ghost obstacles. When we tested this package in the simulator, it yielded better results, but we still observed many ghost obstacles.

[23] `http://docs.ros.org/melodic/api/nav_msgs/html/msg/OccupancyGrid.html`
[24] `http://wiki.ros.org/map_server#map_saver`
[25] `http://wiki.ros.org/costmap_2d`

Odometry and Localization

Odometry is an estimate of how position of the robot changes over time based on data from sensors which measure the movement of the robot. Odometry is supposed to be continuous and there should not be any sudden changes in the position of the vehicle.

Our vehicle has two sources of odometry: the number of revolutions of the motor shaft and the accelerations measured by the IMU. We implemented a simple ROS node called `odometry_node` in C++ which subscribes to the motor shaft encoder and the steering commands supplied to the vehicle. From the number of revolutions and an assumption that the wheels are not skidding we can estimate the distance the vehicle travelled since the last update. From the steering command, we can estimate the steering angle of the front wheels of the vehicle. From these two inputs, we can estimate how the vehicle body translated and rotated using a simple kinematic vehicle model. This approach was inspired by the MIT RACECAR VESC odometry ROS node[26]. Source code of the odometry node is located in the attachments in `/catkin_ws/src/racer_sensors`.

The data from the odometry node and the IMU are then fused into a single estimate using the `ekf_localization_node` node from the `robot_localization` package[27]. This node uses Extended Kalmann Filter (EKF) to estimate the most likely location from the input sources and their covariances. This node produces a `tf` transform between the `odom` and `base_footprint` frames. We also tried to use the ROS package `robot_pose_ekf` which also implements an EKF, but the results were less accurate.

The data from the odometry sensors is inherently noisy and inaccurate and the estimated position of the robot will become more and more inaccurate over time. This phenomenon is referred to as drift. To counteract the drift, we perform a correction using the Adaptive Monte Carlo Localization (AMCL) algorithm. This algorithm uses the data from the LIDAR to determine the position of the vehicle in the map based on the distances to the obstacles around it. Unlike odometry, the sequence of position updates is not expected to be continuous and the resulting location can in theory be a point anywhere on the map. We use an implementation of this algorithm in the `amcl` ROS package[28]. The `amcl` node from this package publishes a `tf` transformation between `map` and `odom` to correct the odometry drift. The `tf` tree is captured in Figure B.2.

Tuning odometry and localization ROS libraries to obtain reliable data at high rates proved to be difficult. In some scenarios, the robot would "get lost" when the localization algorithm would match the scan incorrectly and it would report incorrect location of the robot on the map. This can easily lead into a crash of the robot with a wall or an obstacle. The AMCL algorithm will sometimes manage to find the correct location after a few seconds, but in these cases it was necessary to take over control over the vehicle manually with a remote control and stop the vehicle until the robot corrects its location. The AMCL ROS node has many parameters which we spend many hours tuning them in order to get the best possible results. The final configuration is available in `/catkin_ws/src/racer/launch/amcl_localization.launch`.

[26] `https://github.com/mit-racecar/vesc`

[27] `http://docs.ros.org/melodic/api/robot_localization/html/index.html`

[28] `http://wiki.ros.org/amcl`

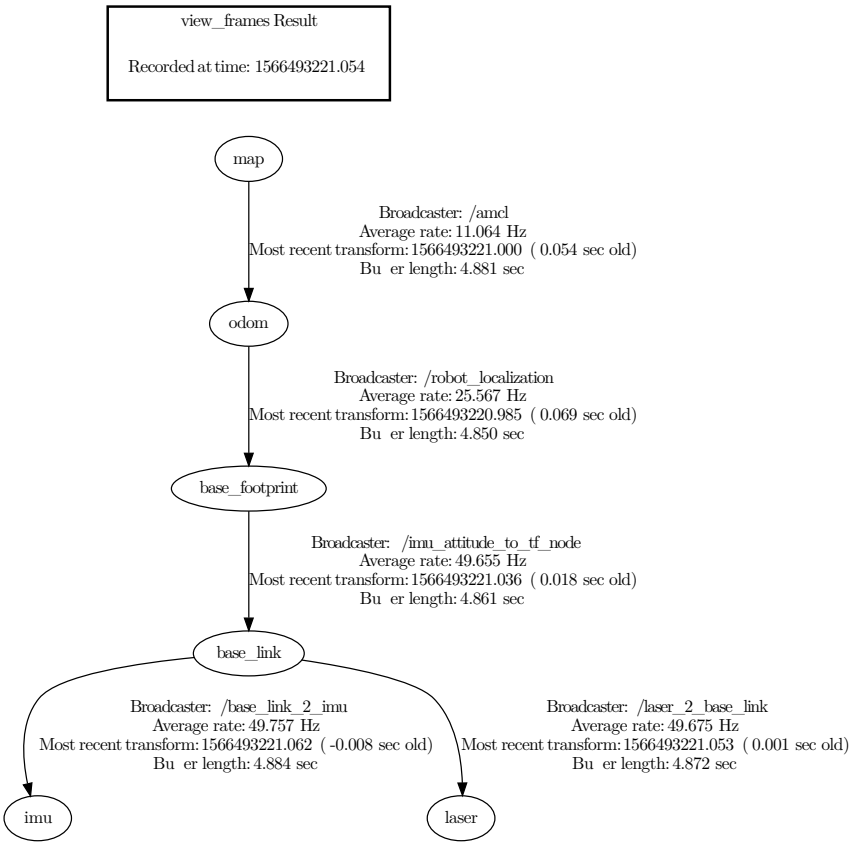


Figure B.2: The tree shows which nodes publish transformations between the coordinate frames and at what average rate the transformations are updated. This diagram is the output of the `view_frames` diagnostics tool from the `tf` package.

Agent Behavior

We implemented the individual components of the behavior of the agent, as it was described in Chapter 2 and visualized in Figure 2.3, with four ROS nodes as part of the `racer` package we implemented:

1. `current_state_node`,
2. `circuit_node`,
3. `planning_node`,
4. `dwa_planning_node`.

The C++17 implementation of the `racer` package is located in `/catkin_ws/-src/racer/` and it contains the source code of the individual ROS nodes as well as the source code of the algorithms which were described in this thesis. The definitions of the messages are located in a separate package `/catkin_ws/src/racer_msgs/`. All the coordinates used by `racer` package nodes are in the global `map` coordinate frame unless stated otherwise.

Current State Node This node listens to the `tf` transformations and converts them into a custom message format `racer_msgs/State`. This message contains the 2D position of the vehicle, its orientation, and its immediate speed in the heading direction in the `map` `tf` coordinate frame.

To achieve the highest possible update rate, we had to manually combine the latest odometry estimate `odom -> base_link` and the latest odometry drift correction `map -> odom` by multiplying their transformation matrices. When we tried obtaining the transformation `map -> base_link` from the `tf` library directly, the update rate was the same as the transform published by AMCL. By manually combining the transformations, we would be able to publish the current state estimate in sync with odometry updates.

If we assume that the fused odometry gives us reasonable movement estimates over short time periods and that we use a very recent drift correction, this workaround is reasonable. The maximum rate at which we can control the actuators is 50 Hz and we are able to achieve vehicle state updates at the rate of 25 Hz (see Figure B.2).

Circuit Node The circuit node analyzes the occupancy grid of the current track based on a circuit definition for the track at the start of the race using Algorithm 4. The circuit definition is a list of checkpoints on the map. The vehicle is supposed to drive along the track in a way such that it passes these checkpoints in the given order. The starting position is considered to be an implicit checkpoint. This list is loaded from a `.yaml` configuration file which contains an array of 2D point coordinates under the key `check_points`. The coordinates are strings with the x and y coordinate separated by a whitespace. An example of such configuration file with two checkpoints follows:

```
check_points:
  - "6.385 -0.521"
  - "-0.639 -4.226"
```

Later during the race, this node monitors the state of the vehicle and it produces a list of the next waypoitns in a custom message format `racer_msgs/Waypoints`. The number of waypoints which are published can be configured using the `lookahead` parameter of the node.

Planning Node The planning node plans a trajectory based on the last known state of the vehicle through the next several waypoints directly ahead of the vehicle and publishes it in the form of a custom message type `racer_msgs/Trajectory`. The planning node uses either the SEHS or the Hybrid A* algorithm based on an parameter passed from a ROS launch file.

Following Node

This node listens to the current vehicle state and the planned trajectory topics and it selects the actions for the vehicle using either the DWA or the Pure Pursuit algorithms. The strategy which the node uses can be changed via a parameter. The node publishes the actions in two standard message types on two topics. These message types are `geometry_msgs/Twist`[29] and `ackermann_msgs/AckermannDrive`[30].. The `Twist` message type mimics the output of the open source `move_base` ROS package[31]. The throttle level is stored in the `linear.x` component ant the steering level is stored in `angular.z`. This message is later translated into a PWM signal for the actuators by the hardware interface component. The `AckermannDrive` message type is necessary to control the vehicle in the simulator.

Hardware Interface

The `Arduino 2` from Figure B.1 has two responsibilities:

(i) subscribe to the commands produced by the agent and transform them into electrical signals for the servo and the DC motor,

(ii) monitor the input from the radio controller and in case that there is a strong signal, disable the autonomous mode and forward the signal to the actuators to allow the human supervisor to control the vehicle directly.

Conversion of Commands to PWM The agent behavior produces actions in the form of two real numbers between -1 and 1 for the steering angle and for the throttle level. These numbers are mapped to PWM duty cycles which are then passed to the actuators through PWM-capable digital pins of the Arduino.

The mapping for the steering servo is straightforward. The $[-1, 1]$ interval is directly mapped to the interval of $[1200\,\mu s, 1800\,\mu s]$ which are the bounds of the servo of our vehicle. The wheels steer to the leftmost position when the duty cycle is set to $1200\,\mu s$, at $1500\,\mu s$ the wheels are aligned with the body of the vehicle, and at $1800\,\mu s$ the wheels are in the rightmost position.

[29] `http://docs.ros.org/melodic/api/geometry_msgs/html/msg/Twist.html`
[30] `http://docs.ros.org/jade/api/ackermann_msgs/html/msg/AckermannDrive.html`
[31] `http://wiki.ros.org/move_base`

The mapping of the signal for the DC motor is a bit more complicated. The range of duty cycles [1400 µs, 1600 µs] does not produce enough torque to set the vehicle into motion. The range of [1000 µs, 1400 µs] corresponds to reversing, where at 1000 µs the motor spins the fastest in the reversing direction. The range of [1600 µs, 2000 µs] corresponds to going forward, where at 2000 µs duty cycle the motor spins the fastest in the forward direction. The boundary values of 1400 µs and 1600 µs are not fixed and they vary based on how much is the battery charged. We therefore use the following mapping:

$$\begin{cases} [-1, -0.05] & \rightarrow [1000\,\mu s, 1400\,\mu s] \\ [-0.05, 0.05] & \rightarrow 1500\,\mu s \\ [0.05, 1] & \rightarrow [1600\,\mu s, 2000\,\mu s] \end{cases}$$

The limits can be adjusted for different needs, e.g. the maximum speed can be limited by lowering the maximum duty cycle.

Manual Override The two signals from the radio receiver are connected to two interrupt pins of the Arduino. By measuring the time difference between the detection of a raising edge and a falling edge of the signal we can determine the width of the duty input cycle. If one of the two signals falls out of the range of [1400 µs, 1600 µs], we consider it to be the supervisors intervention and we will start forwarding the PWM signal directly to the actuators and we will stop processing the commands from the agent. The autonomous mode can be restored by physically pressing the reset button on the board.

Deployment The code of the hardware interface described in Section B.4.3 is located in the `/arduino/steering_node` directory. The code for the motor shaft encoder described in Section B.4.1 is located in the `/arduino/motor_shaft_encoder` directory. In order to upload the code to an Arduino board, use the Arduino IDE[32].

B.4.4 Requirements and Compilation

First, you need to install ROS *Melodic Morenia* Desktop-Full Install according to the instructions provided on the official website of the project[33]. Then, you need to clone the following open-source ROS packages from GitHub into the `/catkin_ws/src` directory:

```
> cd /catkin_ws/src
> git clone --branch 0.4.0 https://github.com/tu-darmstadt-ros-pkg/hector_slam
> git clone --branch 1.8.8 https://github.com/ros-perception/laser_filters
> git clone --branch 2.6.5 https://github.com/cra-ros-pkg/robot_localization
```

Depending on the LIDAR and the IMU you use, add the corresponding ROS packages. In the case of the sensors described in Appendix A, we installed the following packages:

[32] `https://www.arduino.cc/en/Main/Software`
[33] `http://wiki.ros.org/melodic/Installation`

```
> cd /catkin_ws/src
> git clone --branch 1.4.1 https://github.com/YDLIDAR/ydlidar_ros
> git clone https://github.com/yoshito-n-students/bno055_usb_stick
> git clone https://github.com/yoshito-n-students/bno055_usb_stick_msgs
```

Further, our project depends on these packages, which are part of the full installation of ROS:

- `amcl`[34]
- `tf2`[35]
- `map_server`[36]
- `costmap_2d`[37]
- `rosserial_python`[38]

If any of these packages is missing on your computer after you install ROS, you might have to install them manually.

To run the simulator, we also need to clone the repository from GitHub and switch to the state at the time of writing of this thesis. We also need to install the dependencies as they are stated in the installation guide in the `README.md` file of the repository:

```
> cd /catkin_ws/src
> git clone https://github.com/f1tenth-dev/simulator
> cd simulator
> git checkout 140f8bd93b22b59d7cf7bfa753e1fdc73a4a8e59
> sudo apt install -y ros-melodic-navigation \
                      ros-melodic-teb-local-planner \
                      ros-melodic-ros-control \
                      ros-melodic-ros-controllers \
                      ros-melodic-gazebo-ros-control \
                      ros-melodic-ackermann-msgs \
                      ros-melodic-serial \
                      qt4-default
```

With all the dependencies installed and cloned, we need to finish the setup and compile all the packages and nodes in the project:

```
> cd /catkin_ws
> catkin_make install
> source devel/setup.bash
> catkin_make
```

[34] `http://wiki.ros.org/amcl`
[35] `http://wiki.ros.org/tf2/`
[36] `http://wiki.ros.org/map_server`
[37] `http://wiki.ros.org/costmap_2d`
[38] `http://wiki.ros.org/rosserial`

B.4.5 Usage

The parameters for the agent behavior nodes can be changed in the `/catkin_ws/src/racer/launch/agent.launch` file. This file is included from all the other launch files mentioned further in this section.

Simulator

In order to run the SEHS planning algorithm along with the DWA following algorithm, simply run these commands:

```
> source catkin_ws/devel/setup.bash
> roslaunch racer_simulator dwa.launch _use_sehs:=true
```

When the `use_sehs` parameter is set to `false`, the Hybrid A* algorithm is used instead. To run the Pure Pursuit algorithm instead of DWA and with the Hybrid A* planning algorithm, change the name of the launch file:

```
> roslaunch racer_simulator pure_pursuit.launch  _use_sehs:=false
```

The semantics of the `use_sehs` stays the same also for this launch file. After you run the `roslaunch` command, it will take a short while before the *Rviz* and *Gazebo* programs are launched and the vehicle starts moving along the circuit.

Experimental Vehicle

To run the code on the experimental vehicle, we first connect the onboard Nvidia Jetson Nano computer and a laptop to the same Wi-Fi network and we connect to the on-board computer over `ssh`. We then launch the main launch file on *Jetson*:

```
> source catkin_ws/devel/setup.bash
> roslaunch racer race.launch _map:=<map> _circuit:=<circuit>
```

where `<map>` is the name of the map definition file and `<circuit>` is the name of the circuit definition file. Both of these names must not include the `.yaml` suffix and the files must be placed in the `/catkin_ws/src/racer/maps` and the `/catkin_ws/src/racer/circuits` directory.

Since both computers are connected to the same network, we can listen to the ROS topics on the laptop and visualize the data exchanged between these topics using *Rviz*. In order to achieve this, the `ROS_MASTER_URI` and `ROS_IP` environment variables must be set properly on both computers. Follow the *Network Setup* tutorial on the ROS website[39].

B.5 Implementation of Algorithms

In this section, we will provide a high-level overview of the C++ implementation of the racing agent. We will focus on the code of the algorithms which is located in the `/catkin_ws/src/racer/include` directory. The code is organized into two root namespaces: `racer` and `racer_ros`. In this section, we will go over the nested namespaces and we will mention their responsibilities and we will introduce the most notable classes and structures in these namespaces.

[39] `http://wiki.ros.org/ROS/NetworkSetup`

Convention Each `struct`, `class`, or standalone function is located in a separate header file. The name of the type matches the file name. The location of each type in the directory structure corresponds to its namespace.

B.5.1 The `racer` namespace

This namespace contains all the algorithms described in this thesis. This includes the trajectory analysis algorithm, the Hybrid A* and SEHS algorithms, and the DWA and the Pure Pursuit algorithms.

The `racer::math` namespace

This namespace includes several important math primitives, operations between them, and helper functions which are used throughout the codebase:

- `angle` is a simple typed representation of an angle in radians. The structure contains several helper functions for conversion between degrees and radians and for manipulation with an angle value.
- `vector` is a two dimensional representation of a vector with several methods for manipulation with vectors. This structure has a type alias `point`.
- `rectangle` is a representation of a rectangle in a two dimensional plane. It contains a function to rotate the whole object around the origin of the coordinate system and to check if two rectangles intersect each other. These functions are necessary mostly for collision detection.
- `circle` is a representation of a circle in a two dimensional plane. It contains functions for generating points on the circumference of a circle and for checking if one circle overlaps with another one. This type was created mostly for the needs of the SEHS algorithm, but it is used also in other parts of the codebase.

The `racer::track` namespace

This namespace contains code related to the race track and to the collisions between the vehicle and the track:

- `analysis` is a class which contains the implementation of the trajectory analysis algorithm for detecting race track corners in an occupancy grid described in Section 3.4.
- `centerline` is a simple class which contains an algorithm for finding an approximation of a center line in an occupancy grid.
- `occupancy_grid` is a class which implements all the necessary functionality for accessing the occupancy grid data. The class contains functions to find an approximate distance to the closest obstacle to a given point of the grid and to check if there is a clear line of sight between two points of the grid.
- `collision_detection` is a class which checks for collision in a given occupancy grid as described in Section 3.2.2.

- `circuit` is a class which implements functions for monitoring the waypoints of the circuit and whether a vehicle passed a waypoint or not.

The `racer::vehicle` namespace This namespace contains classes and structures related to vehicle movement modeling:

- `action` is the input for the vehicle. It contains a factory method which creates a list of actions from a predefined range of throttle and steering angle values.
- `configuration` is a simple structure which contains the position and the orientation of a vehicle.
- `trajectory` is a structure which represents a trajectory found by the planning algorithm. It contains functions to find the closest step of the trajectory to any given vehicle configuration which is used both in the DWA algorithm implementation and in Pure Pursuit.
- `motor_model` and `steering_servo_model` are classes which model the behavior of a DC motor and the steering servo. They both contain factories to create an instance of the model with the fitted values of parameters. `chassis` is a class which holds all the parameters of the vehicle.
- `base_vehicle_model` is a pure abstract class which is meant as a base to any vehicle model implementation (e.g., kinematic or dynamic vehicle model).
- `kinematic::state` and `kinematic::model` are an implementation of the kinematic bicycle model described in Section 3.5.2.

The `racer::space_exploration` namespace

This namespace contains classes related to the SEHS algorithm for the `space exploration` part of the algorithm as described in Section 3.3.3. The rest of the SEHS algorithm is located under the `racer::astar` namespace.

- `space_exploration` is a class contains the `explore_grid` function which contains an implementation of the A* algorithm and which finds the path of circles through a series of waypoints.

The `racer::astar` namespace

This namespace contains all the code of the generic A* algorithm implementation.

- `open_set` and `closed_set` are wrappers around the standard C++ containers `std::priority_queue` and `std::unordered_set` with an interface tailored for the A* algorithm.
- `search` is a function which implements the A* algorithm itself as described in Section 3.3.2 and Section 34. The function finds a solution to an instance of a search problem in the form of a class derived from `base_search_problem`.

- `discretized` is a nested namespace which contains a generic implementation of a search problem in a discretized state space.
  - `discretized::search_problem` is an implementation of `base_search_problem` designed for autonomous racing. This problem requires an implementation of the `discretized::base_discretization` pure abstract class. By injecting a specific discretization, this problem can represent either the Hybrid A* or the SEHS algorithm.
- `sehs` is a nested namespace which contains the specific discretization for the SEHS algorithm.
  - `sehs::discretization` is a class which implements the discretization of the state space based on the path of circles obtained from from *space exploration*
  - `sehs::nearest_neighbor` is a class which implements linear search to find the nearest circle to any vehicle configuration.
- `hybrid_astar` is a nested namespace which contains the specific discretization for the Hybrid A* algorithm.
  - `hybrid_astar::discretization` is a class which implements the discretization of the state space into a uniform grid.

The `racer::following_strategies` namespace

This namespace contains code for the DWA and Pure Pursuit trajectory following algorithms:

- `following_strategy` is a pure abstract class which contains the definition of the API of a trajectory following strategy. A following strategy must implement the `select_action` function which selects an appropriate action based on the current state of the vehicle and the planned trajectory.
- `target_error_calculator` is a class which implements scoring a trajectory considered by `dwa_strategy` based on weight parameters as described in Section 4.2.
- `dwa_strategy` is a class which implements the `following_strategy` and the DWA algorithm.
- `pure_pursuit` is a class which contains the implementation of the Pure Pursuit algorithm for selecting the steering angle.
- `pure_pursuit_strategy` is a class which implements the `following_strategy` interface and it combines the steering angle obtained from the `pure_pursuit` class and the throttle based on the reference trajectory.

B.5.2 The `racer_ros` namespace

This namespace contains the bindings between the code in the `racer` namespace and the ROS libraries and our custom ROS nodes.

The most interesting set of functions is in the `include/racer_ros/utils.h` file. This file contains functions for converting our types into ROS message types.

- `discretized_planner` is an abstract class which observes map updates, vehicle state updates, and waypoint updates and triggers the discretized planning algorithm based using a discretization object implemented by a derived class.
- `hybrid_astar_planner` extends `discretized_planner` and it provides the Hybrid A* discretization.
- `sehs_planner` extends `discretized_planner` and it provides the SEHS discretization.
- `follower` is a class which observes the map updates, vehicle state updates, waypoint updates, and planned trajectory updates and it can select the next action from the last known vehicle state based on the injected trajectory following strategy.
- `circuit_progress_monitoring` is a class which detects corners in observed circuit map and which publishes the coordinates of the next `n` corners ahead of the vehicle and updates this list whenever the vehicle reaches the next waypoint ahead of it.