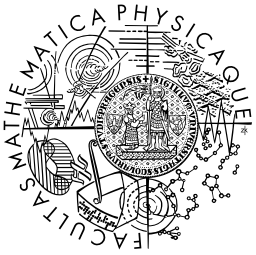


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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 2018

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

Dedication.

Title: Trajectory planning for fast moving cars

Author: Šimon Rozsíval

Department of Theoretical Computer Science and Mathematical Logic: Department of Theoretical Computer Science and Mathematical Logic

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

Abstract: Abstract.

Keywords: key words

Contents

Todo list

Introduction

Autonomous vehicles are slowly making their way from research facilities to the public roads around the world and it seems inevitable that they will soon become a part of our everyday lives. Autonomous vehicles could have great 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 seems to be 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 not new. Prototypes of autonomous vehicles were demonstrated on public roads as early as in the 1980s. 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 could see that self-driving cars are not sci-fi anymore. 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 of other vehicles.

These commercially available assistants provide the user only with partial autonomy. The driver is required to pay attention to the road at all times 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 no matter what the weather and traffic is like outside. 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. 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.

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 will describe an architecture of an artificial agent for the task of autonomous racing.

Next, we examine the individual subproblems we must solve in order to create the racing agent. We will study how the vehicle reacts to steering commands and we describe two vehicle models with different levels of complexity and accuracy in the chapter Vehicle Model. With a model of the vehicle defined, we formulate the trajectory planning problem for an autonomous racing car, and we describe several methods of finding a feasible plan in the chapter Trajectory Planning Problem. In the chapter Trajectory Following we will describe algorithms for following a predefined reference trajectory.

Finally, we implement the racing agent and we conduct several experiments to verify the performance and success rate of our algorithms on a physical robot in the chapter Experimental Verification, and summarize our findings.

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 form 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 would 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 intrinsic measurement unit (IMU) with gyroscopes and accelerometers which measures the linear and angular acceleration of the vehicle. The combination of 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 Requirements

In this thesis, we will design and develop an autonomous vehicle for the task of autonomous racing. The racing scenario is a 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 relaxed problem allows us to ignore any traffic rules. We do not have to consider any speed limits, crossroads, or stop signs.

This problem is inspired by the F1/10 competition organized by the University of Pennsylvania and the University of Virginia [9]. 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.

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.

1.2 Related Work

In this section we will go through several interesting approaches to path and trajectory planning from the area of robotics and autonomous cars.

Shakey the Robot was a project at the Stanford Research Institute in the 1960s. For the purposes of efficient route finding the A* algorithm was designed by Nils J. Nilsson and his coworkers [10]. This algorithm is widely used for solving various graph search problems thanks to its simplicity, completeness, and optimality. The NavLab autonomous vans of the Carnegie Mellon university was one of the first autonomous vehicles tested on public roads in the 1980s and 1990s. One of the interesting results of their work is the Pure Pursuit control algorithm [11].

Steven M. LaValle designed a randomized path planning algorithm for vehicles with kinodynamic driving constraints and high-dimensional configuration spaces called Rapidly-Exploring Random Trees (RRT) [12]. This algorithm randomly samples the configuration space and steers the vehicle towards the sampled point in the space. A tree of feasible paths in the configuration space rooted in the starting position is constructed until a path which ends in the goal position is found. The algorithm is probabilistically complete, but it is not optimal. The algorithm was extended by many times in order to converge faster and to be optimal. The optimal variant of the algorithm, RRT* [13], is unfortunately hard to use for car-like vehicles.

In 2007, the DARPA Urban Challenge took place in the USA which 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 [14]. The Stanford team used an algorithm called Hybrid A* in their vehicle called Junior [15]. 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 [16].

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 a behavior of an autonomous agent and decompose the problem 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 a time period as short as possible. 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 their different properties such as maximum speeds, acceleration, braking power, or the grip of the tires.

For a turn with a steady 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 commanded by the driver. This situation is called understeer or oversteer, depending on whether it was the front or rear wheels which lost 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 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 higher lap time.

The trajectory of the vehicle as it goes through a corner can be split into several stages. First, the vehicle aligns itself to the outer edge of the track. Second, the vehicle must adjust its speed so it can safely stay on the curve. It

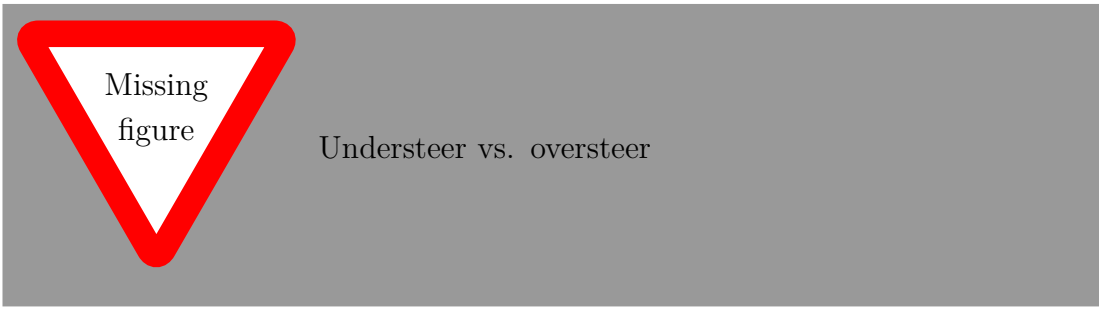


Figure 2.1: This figure shows the difference between oversteer and understeer.

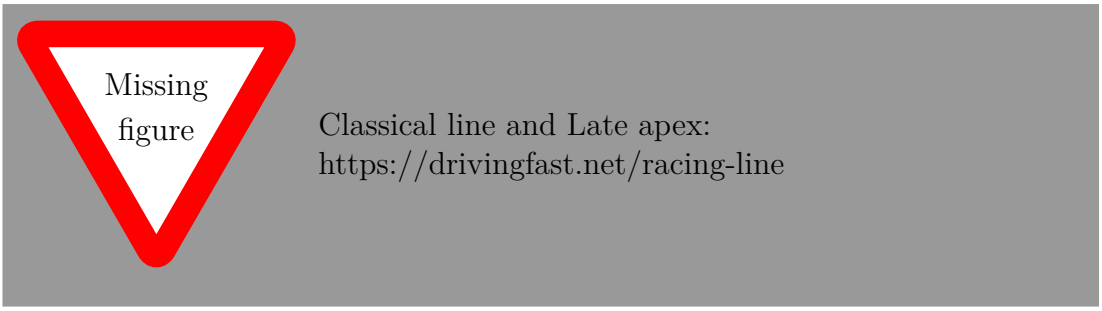


Figure 2.2: Classical line vs. Late apex

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

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 some 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 Inertial Meassurement Unit (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 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 for 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 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 process which is visualized in Figure 2.3.

There is one more reason to decompose the task into several sub-problems. The rate at which the nodes produce outputs is different and they work asynchronously. While the trajectory following node 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 node will have 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

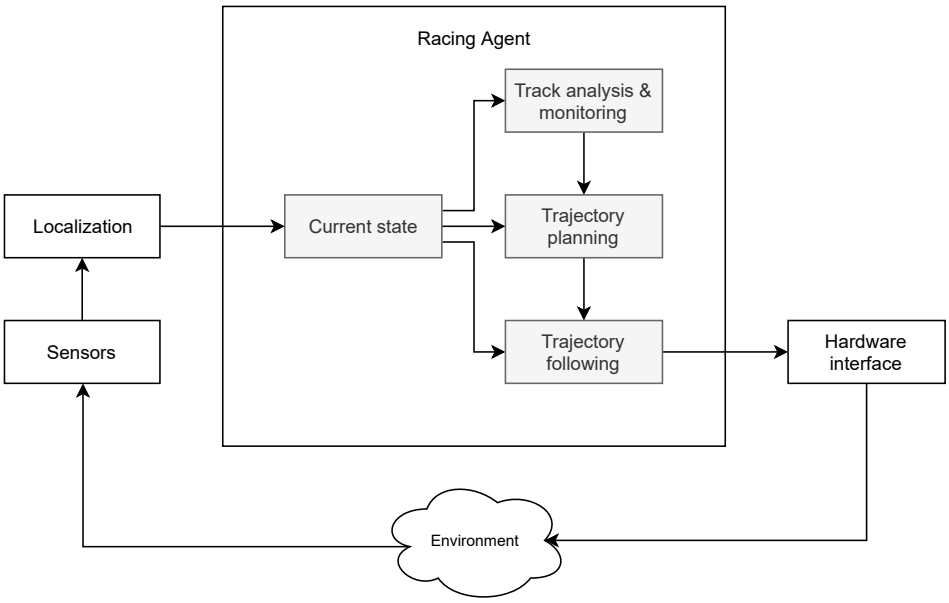


Figure 2.3: A diagram of the decision process of the racing agent.

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 process 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 node 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 search. This directly affects the update rate of the trajectory planning node.

Trajectory planning The trajectory planning algorithm finds a feasible trajectory from the last state of the vehicle estimated by the localization node through the selected waypoints. The trajectory planning node 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 node will send the commands to the hardware at the maximum possible rate which the hardware is capable of. In practice we are limited by the rate at which we receive the state estimates from the localization node. The trajectory following node 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.

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 of the planning problem will be a trajectory. A trajectory is the description of both a path the vehicle should follow and also a speed profile. 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* [17]. This book describes all of 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 some 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 of 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 a vector of an n-dimensional vector of real numbers. All of the 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 of the 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 condition function $g : X^* \to \{T, F\}$.*

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:*

$$\forall i \in \{0, 1, \ldots, k\} : u_i \in U(x_i) \wedge x_{i+1} = f(x_i, u_i)$$
$$g(\langle x_0, x_1, \ldots x_{k+1} \rangle) = T.$$

Determining if a plan meets the goal condition is often performed by checking if the final state x_{k+1} of the plan is in some subset of goal states $X_g \subset X$. Often there are many feasible plans for some task, but it is not sufficient to find just any of these plans. We might be tasked to find an optimal plan, which minimizes some cost function.

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 that the states form vertices of a graph $G = (V, E)$ and the applications of actions through the state transition function form 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 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 states vertices. Finding a path in a graph is 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 spaces 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 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 be 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 see, that the state space of all of the 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 of the combinations of the state variables of the world.

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 a 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* [17, 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 we described it 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 **??**.

For a car with a wheelbase length $L \in \mathbb{R}$, its 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 3.2.3.

Obstacles Additionally, we must be able 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.

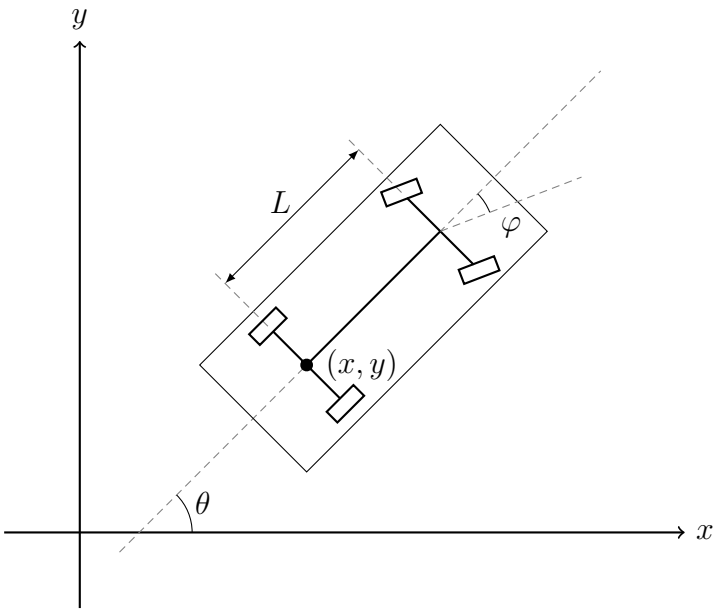


Figure 3.1: The simple car from the example has three state variables. The speed and the steering angle are action variables and they can change at any moment, so the vehicle can stop on a spot and change the steering angle instantaneously, which is of course not possible in a real world car.

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 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* and satisfies a goal condition. 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 some 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*:* $\dot{x} = f(x, u)$*.*
3. *A goal condition function* $g : \tilde{X} \to \{T, F\}$*, where* $\tilde{X}$ *is the set of all state trajectories obtained by integrating all possible action trajectories over* 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* $\tilde{u}$ *such that:*

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

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 some non-trivial time period. This effectively discretizes time into time intervals, which usually have a fixed length Δt or the duration depends depend on the action which is used ("a motion primitive" [17, 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 some action from some state over some 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 of the 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.

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) [17, 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?

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, whose solution is a time-optimal state trajectory and the vehicle does not at any point in time exceed its handling limits. The state trajectory is expected to describe both the poses of the vehicle, as well as a speed profile.*

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 late 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 any 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 capture 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.

3.2.1 Problem Formulation

The Configuration Space and the State 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 of the 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 will consist 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.

The complete *state space* X of the vehicle will be a combination of the configuration space C and additional variables representing the kinematics and dynamics required by the differential constraints. We will discuss the choice of appropriate vehicle models in Section 3.2.3 and we will define a concrete state space for each of the vehicle models.

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 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 in mind 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.

Collision Detection

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

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

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.

The planning algorithm does not work directly with the configuration space, but rather with a state space. An extension of the partitioning into C_{free} and C_{obs} into the state space is very straightforward because a collision is dependent only on the configuration of the vehicle body, not on the kinematics and dynamics of the vehicle. For a state $x \in X$, which is an extension a configuration $c_x \in C$, it holds $x \in X_{obs} \iff c_x \in C_{obs} \wedge x \in X_{free} \iff c_x \in C_{free}$.

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: either it can be traversed freely, or there is some obstacle in the tile and so the tile must be avoided. This is represented by the values 0 (free space) and 1 (an obstacle) in our definition.

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 can 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 can be:

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

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), especially when a robot is equipped with a LIDAR. 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.

(a) The relative coordinates to tiles occupied by the footprint of the vehicle when the heading angle is close to θ.

(b) The obstacles are "inflated" by the radius of the vehicle.

Figure 3.2: Photos of the experimental vehicle.

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 **??** 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 f 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.

The second idea is simpler. First, we determine the minimum safe distance of the vehicle. We will then change all free tiles in the occupancy grid to obstacle tiles if they are closer to an original obstacle tile than the minimum radius and "inflate" the obstacles. To check for collisions of a vehicle at position (x, y), only a single invocation of c_1 is required, which has very little performance requirements. This method is depicted in Figure **??**.

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. In principle, we could fix this by "inflating" the occupancy grid for different heading angles and mark the footprints of rotated rectangles directly into the occupancy grid. The problem we had with this approach is an increased cost of pre-processing before the planning algorithm can start. The footprints of the vehicle can be pre-calculated ahead of time and they do not take any extra computation at runtime.

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 for all of the heading angles. In the end, the second approach was sufficient for the racing task and we used the simple implementation. The resulting function is sometimes "to cautious" and it reports collisions even if there is still some gap between the actual vehicle and the obstacle, but that can be considered as an advantage for high speed maneuvers and it might make sense to increase the radius even more to force the trajectory to be further from the obstacles in practice.

Action Space

Our vehicle has two actuators which can be controlled:

- 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 Rotations 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:

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

U is a set of tuples of a steering angle proportion δ_t, and a throttle position τ_t. These actions are an abstraction of the signals which would be 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 some direction and the action specifies a throttle position in the opposite direction, the vehicle might engage brakes, if it is equipped with any. 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 to integrate the velocity in the state space over a constant time period Δt with a constant action $u \in U(x(t))$ 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 $\forall \tau \in [t, t + \Delta t] : x(\tau) \in X \wedge u \in U(x(\tau))$. 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

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

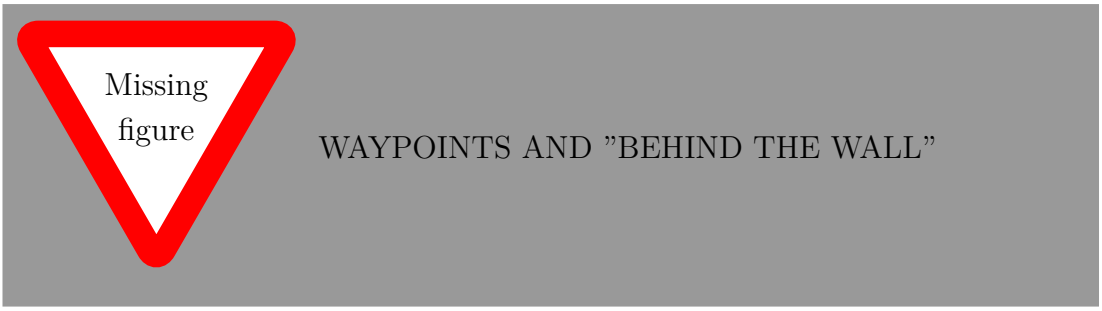
Executing some actions for a time period Δt might result in a collision with an obstacle. To avoid this problem, we will limit the action set of a vehicle state by incorporating the collision detection function and the system simulator $f_{\Delta t}$ to allow only safe actions:

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

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.

Goal Condition

In Definition 2 we used a set of goal states to check if a plan is a solution to the problem or not. When we analyzed the problem earlier in this chapter, we decided to plan a trajectory for the next $n > 0$ 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. 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 a the given order. The concept of "passing a sequence of waypoint" 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 shortest path between these two points. The set* $D \subseteq \mathbb{R}^2 \times \mathbb{R}^2$ *captures this relation:*

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

Definition 11 (Waypoint)**.** *Waypoint goal region is a set*

$$G_r(\boldsymbol{w}) = \{x \in X \mid \|\boldsymbol{p_x} - \boldsymbol{w}\| < r \wedge (\boldsymbol{p_x}, \boldsymbol{w}) \in D\},$$

where p_x *is the position vector of the state* x *and* $r \in \mathbb{R}$ *is the radius of the goal region.* $G_r(\boldsymbol{w})$ *is a set of all states which are considered to pass the waypoint.*

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 **??**.

Definition 12. *A sequence of states* $\langle x_0, x_1, x_2, \ldots, x_k\rangle$ *passes a sequence of waypoints* $\langle \boldsymbol{w_1}, \boldsymbol{w_2}, \ldots, \boldsymbol{w_l}\rangle, l > 0$ *when a sequence of indexes* $\langle i_1, i_2, i_3, \ldots i_l\rangle$ *exists such that:*

$$\begin{aligned} \forall m, n \in \{1, \ldots, l\} : m < n &\implies i_m < i_n \\ \forall m \in \{1, \ldots, l\} : x_{i_m} &\in G_r(\boldsymbol{w_m}) \\ i_l &= k \end{aligned}$$

Definition 13. *The goal condition* $g_{\hat{w}_l} : X^* \to \{T, F\}$ *is a function sutch that for a given sequence of waypoints* $\hat{w}_l$ *and a state trajectory* $\hat{x_k} = \langle x_0, x_1, x_2, \ldots, x_k\rangle, k > 0 : g_{\hat{w}_l}(\hat{x_k}) = T \iff \hat{x_k}$ *passes* $\hat{w}_l$*.*

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

- *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 $n \in \mathbb{N} : n \geq 3$ state variables and it contains the configuration of the vehicle in a two dimensional plane and other state variables as defined by the vehicle model which is used.*
- *A non-empty finite set U_f of actions, such that $\forall x \in X, \forall u \in U_f(x) : c(f_{\Delta t}(x, u)) = F$, 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 condition function $g_{\hat{w}} : X^* \to \{T, F\}$ for some fixed sequence of waypoints $\hat{w} = \langle w_0, w_1, w_2, \ldots, w_k \rangle, k > 0$.*

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 15 (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 state x_0 when for a fixed time interval $\Delta t > 0$:*

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

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 $U(x)$.

This sequence contains all the information we need to reconstruct the full continuous trajectory by finding the corresponding sequence of actions. In practice, this is not very important to us though. 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 16 (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 $\hat{x_k}$ is a time-optimal feasible solution of the problem if:*

- *the goal condition is met: $g_{\hat{w}}(\hat{x_k}) = T$,*
- *and every other feasible state trajectory $\hat{y_l}$, which meets the goal condition, takes longer or the same time to reach the goal (i.e. $l \geq k$).*

State Trajectory Sub-sampling The choice of the sampling time interval Δt greatly impacts the performance of finding solutions to the planning problem. If the optimal time to reach the goal is t seconds, then a state trajectory for a sampling interval Δt_1 will have more elements than a state trajectory with a sampling rate of $\Delta t_2 > \Delta t_1$. The planning algorithm would therefore have to make more decisions for Δt_1 and it would take longer 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.

Instead, we can uniformly subdivide the time interval of length Δt into $n \in \mathbb{N}$ smaller intervals of $\frac{\Delta t}{n}$ and apply the state transition function n times for every action in the sequence. 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.

3.2.2 Track Segmentation

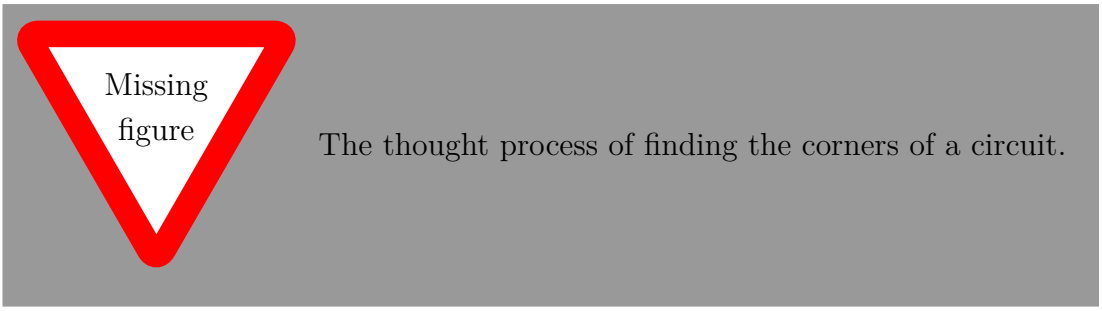
A natural way of splitting the track into smaller segments is to find the corners of the track. We can then plan the trajectory for the very next turn in front of the vehicle and for one or two consecutive ones, and imitate the behavior of a human racing driver, as we described it in Section 2.1. To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

Actually test this.

In this section we will describe an algorithm we chose to find the corners of a racing circuit. We will use this algorithm once just before the start of the race, to split the circuit into a series of short segments. We expect to be given the definition of a racing circuit as an occupancy grid, initial pose of the vehicle with respect to the occupancy grid, and at least two more checkpoints which define the direction in which the vehicle must drive along the circuit. An occupancy grid can be formalized with the following definition:

Definition 17. *Occupancy grid* $G \in \{0,1\}^{m \times n}$ *of resolution* $r \in \mathbb{R}$ *is a two dimensional table of* m *rows and* n *columns which corresponds to a rectangular area of the environment of the width of* $m*r$ *meters and the length of* $n*r$ *meters. The cells of the table fill the area as square tiles of the side length of* r *meters. The value of a cell* G_{ij} *reflects on the state of its corresponding area:*



The thought process of finding the corners of a circuit.

Figure 3.3: The thought process of finding the corners of a circuit.

$$G_{ij} = \begin{cases} -1, & \textit{if } i < 0 \vee j < 0 \vee i \geq m \vee j \geq n \\ 0, & \textit{or if the corresponding tile contains an obstacle} \\ 1, & \textit{otherwise.} \end{cases}$$

The goal of the track analysis algorithm is to find interesting points of the track which split the track into smaller segments corresponding to stretches between the corners of the track. In order to achieve this, we can make a simple observation and derive an algorithm which will produce good approximate solutions.

We can inflate imaginary rubber walls with the thickness of a given safety radius of the vehicle along the edges of the track and loosely lay an imaginary string through the whole circuit. We can then start tightening the string and eventually it will take a form of alternating straight segments and parts, where it touches the rubber wall at an inner edge of a corner of a track. The string represents the shortest path for the vehicle around the circuit. We can then remove the imaginary rubber walls and start walking from the initial position of the vehicle along the string. We will mark the furthest point which is directly visible from the place where we're standing. By directly visible we mean that it is possible to draw a line between the two points in the occupancy grid and it will not intersect a cell containing an obstacle between the two points. This is the first corner ahead of us. We will then walk to this point and repeat the process, until we can see the first point we marked again. This thought process is visualized in Figure **??** on different track layouts.

We will implement this process with a three step algorithm which will identify the corners and points along long winding bends. The first step will be to find the shortest path through the grid which starts at the initial position of the vehicle and which goes through the checkpoints in the correct order and at any point it does not come closer to an obstacle than to a distance of the safety radius. The second step will simply traverse the path once and select a sub-sequence of the points such that for two consecutive points A and B, B was the last point of the points immediately following A on the original track which are directly visible from A. In the third step, we will merge points, which are close together (e.g., in a 180° hairpin turn).3

The shortest path can be found in several different ways. The simplest approach would be to use a simple grid search on the occupancy grid. An interesting alternative is the Space Exploration algorithm described by Chao Chen [18] which uses a search algorithm to explore the grid using circles of variable radii which depend on the distance to the closest obstacle. The expansion of a circle is achieved

by calculating $k \in \mathbb{N}$ points on the circumference of the expanded circle and calculating maximum possible a radius for the given point as a distance to the closest obstacle. We will add the child circle to the open set if its radius is larger than some minimum radius (i.e., we will avoid points too close to obstacles) and if the circle has not been closed yet. A circle will be considered closed if the center of the circle lies inside of an already closed circle. This allows us to avoid exploring some regions of the occupancy grid multiple times. To search the space efficiently, we will use the A* algorithm as the search method. The cost to come to a circle will equal to the distance traveled from the initial position to the circle and the estimate of the cost to go will be equal to the euclidean distance to the goal position. We will stop searching at the moment when we expand a circle which contains the goal position. The outline of the algorithm is described in Algorithm 1 and a visualization of the algorithm is shown in Figure **??**.

Algorithm 1: Space Exploration

Input: Occupancy grid G, starting position $\boldsymbol{x}_0$, goal position $\boldsymbol{g}$
Output: Sequence of circles
parameter: Number of expanded children k, minimum radius r_{min}

$r_0 \leftarrow$ `MaxRadius(`G, $\boldsymbol{x}_0$`)`
$O \leftarrow \{(\boldsymbol{x}_0, r_0)\}$ ▷ `Open set`
$C \leftarrow \emptyset$ ▷ `Closed set`
$P \leftarrow \emptyset$ ▷ `Set of transitions`
while $O \neq \emptyset$ **do**
  $(\boldsymbol{x}, r) \leftarrow$`Top(`$O$`)`
  $O \leftarrow O \setminus \{(\boldsymbol{x}, r)\}$
  if $\|\boldsymbol{x} - \boldsymbol{g}\| \leq r$ **then**
    return `ReconstructPath(`$(\boldsymbol{x}, r)$, P`)`
  end
  for $\boldsymbol{p'}$ *in* `PointsOnCircumference(`$(\boldsymbol{x}, r)$, k`)` **do**
    $r' \leftarrow$ `MaxRadius(`G, $\boldsymbol{p'}$`)`
    if $r' \geq r_{min} \wedge \nexists(\boldsymbol{x_c}, \boldsymbol{r_c}) \in C : \|\boldsymbol{x_c} - \boldsymbol{p'}\| \leq r_c)$ **then**
      $O \leftarrow O \cup \{(\boldsymbol{p'}, r')\}$
      $P \leftarrow P \cup \{\Big((\boldsymbol{x}, r), (\boldsymbol{p'}, r')\Big)\}$
    end
  end
  $C \leftarrow C \cup \{(\boldsymbol{x}, r)\}$
end

To find the path from the initial position through the check points and to the finish line position, we will simply find the path from the initial point to the first checkpoint and then starting from the last circle of the previous path to the next check point. We repeat this until we close the circuit by reaching the initial position again. The path of circles we find might not be optimal. We can further improve it by iterating over the path it and smoothing it as shown in algorithm **??**.

With the path of circles around the circuit, we can now proceed the second step of finding the corners of the circuit. We will traverse the path and mark the centers of circles which are the last directly visible points from the previous point as shown in algorithm 2

Algorithm 2: Waypoint Selection

Input: Occupancy grid G, sequence of n points $(\boldsymbol{x_i})_{i=0}^{n}$
Output: Sequence of points

3.2.3 Differential Constraints

3.2.4 Search Algorithm

4. Vehicle Controller

5. Experiments

Conclusion

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] University of Pennsylvania. F1/tenth.

[10] Nils J Nilsson. *The Quest for Artificial Intelligence*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

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

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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 [19], 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

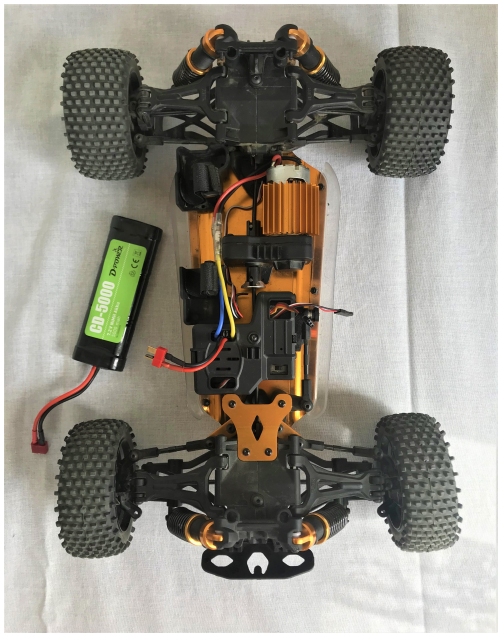
We used chassis, steering servo, Direct Current (DC) motor, Electronic Speed Controller (ESC), and a ratio transmitter and receiver from an off-the-shelf RC car. We removed the plastic cover and some unnecessary parts attached to the chassis and we attached a thin plywood board on top of the chassis. We later mounted all of 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 of the 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 **??**.

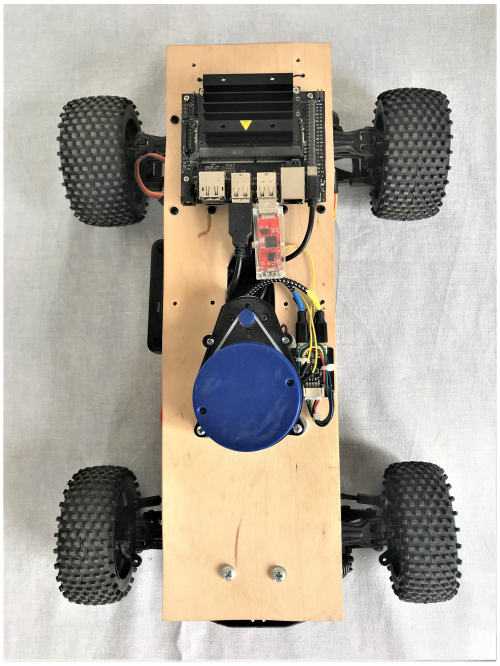
A.2 Sensors

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, 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.

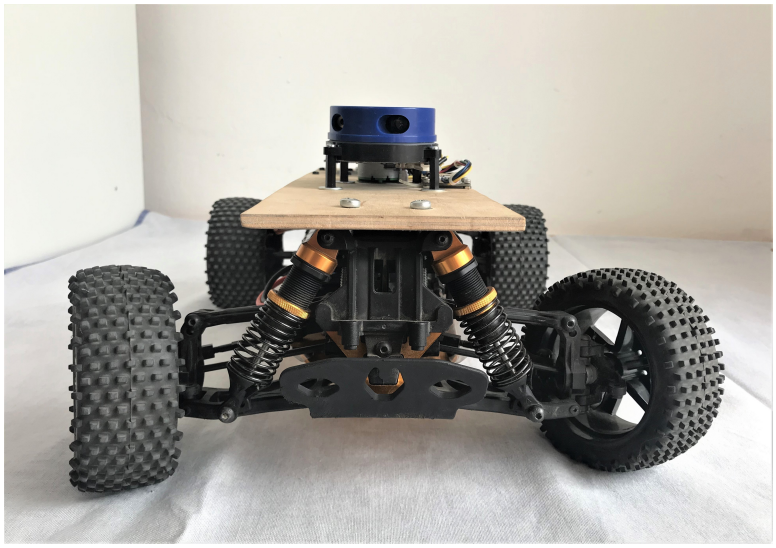
[1] https://mit-racecar.github.io/
[2] https://autorally.github.io/



(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 of 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.

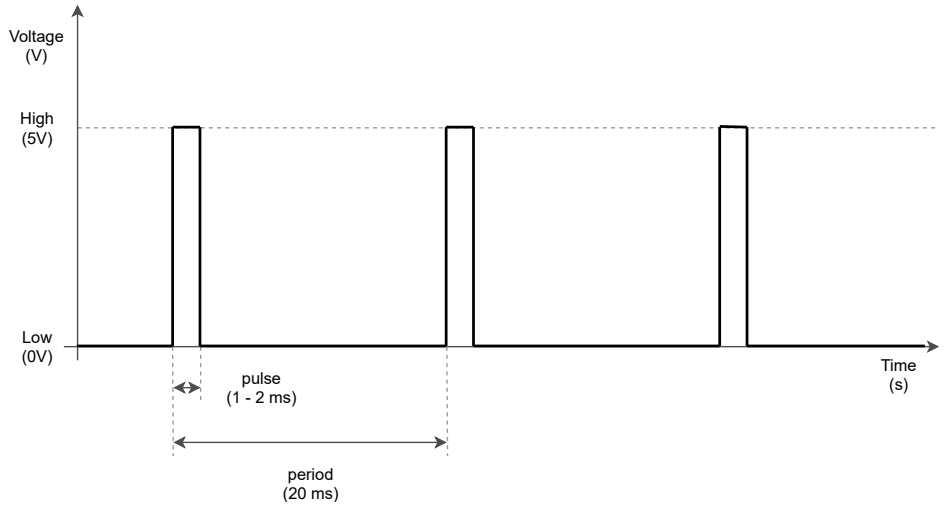


Figure A.2: The PWM signal for the steering servo and the ESC form a square wave with the period of 20 ms and pulse width between 1 ms and 2 ms.

A.3 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 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.

[3]https://www.sparkfun.com/products/12629
[4]https://www.bosch-sensortec.com/bst/products/all_products/bno055

A.4 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.4.1 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.5 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. 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.

[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

B. Technical Documentation

The implementation of our autonomous racing agent relies on a complex set of libraries and tools. In this chapter, we will describe the libraries and tools we used and we will briefly explain how some of them work. We will also discuss our implementation the behavior of the autonomous racing agent. The installation instructions and instructions on how to deploy and use the software used and developed in this thesis are covered in the following chapter C.

B.1 Robot Operating System

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 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. The configuration we used during our experiments is included in the attached files as `/ubuntu/mapping.rviz` and `/ubuntu/race.rviz`.

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. Further information about ROS and tutorials which describe how to work with it are available on the official website of the ROS project[1].

[1] https://www.ros.org

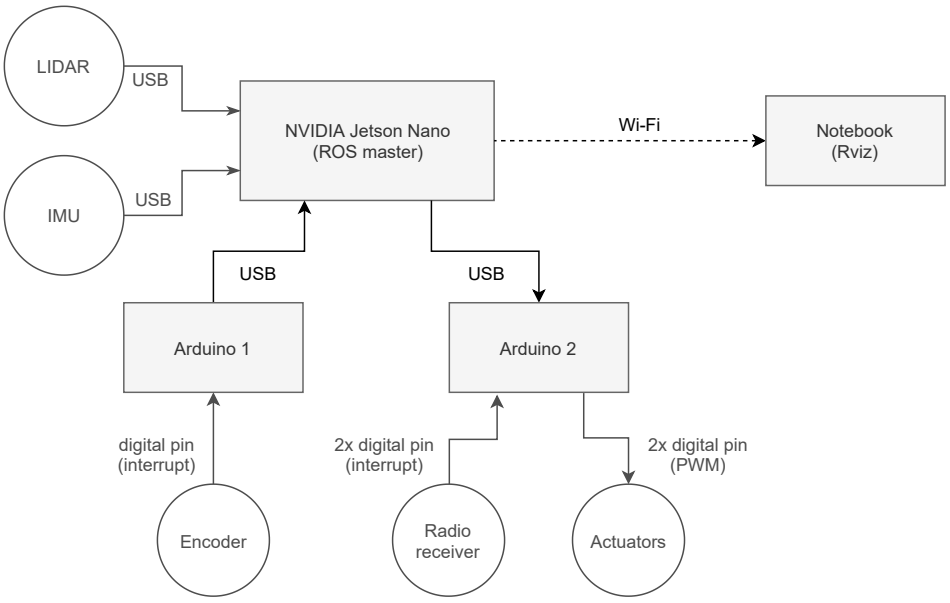


Figure B.1: This figure shows how the computers and sensors which make up the experimental vehicle are connected with each other and the direction of data flow between the devices.

B.2 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 over USB. This is possible thanks to the `rosserial` package[2]. 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.

B.2.1 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`[3]. `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`[4] type. The `tf` package provides libraries for both Python and C++ which make accessing the current state of the whole system and manipulating with the linear transformations easy.

[2]http://wiki.ros.org/rosserial
[3]http://wiki.ros.org/tf
[4]http://docs.ros.org/kinetic/api/tf/html/msg/tfMessage.html

Our implementation adheres to the standard naming convention of coordinate frames as it is defined in ROS Enhancement Proposal (REP) 105[5]. 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[6] 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.

B.2.2 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`[7] and `sensor_msgs/Imu`[8] messages. The data from the motor shaft encoder is published in the form of `std_msgs/Float64` message and it contains the number of revolutions since the Arduino was last powered or reset. The source code for the Arduino is available in the attached source files in `/controller-arduino/wheel_encoders/wheel_encoders.ino`.

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 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[9] 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 `/racer-jetson/catkin_ws/srcracer/launch/ydlidar.launch`.

2. The coordinate frame of the IMU is not ideally 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 `bnp055_usb_stick`[10] does not set them correctly. The configuration for the IMU is stored in `/racer-jetson/catkin_ws/srcracer/launch/imu.launch`.

[5] https://www.ros.org/reps/rep-0105.html
[6] https://github.com/tu-darmstadt-ros-pkg/hector_slam/tree/catkin
[7] http://docs.ros.org/melodic/api/sensor_msgs/html/msg/LaserScan.html
[8] http://docs.ros.org/melodic/api/sensor_msgs/html/msg/Imu.html
[9] http://wiki.ros.org/laser_filters
[10] https://github.com/yoshito-n-students/bno055_usb_stick

B.3 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 [20]:

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

where $S_i(\xi)$ 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 topic of type `nav_msgs/OccupancyGrid`[11]. 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[12]. 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 `/racer-jetson/catkin_ws/src/racer/launch/create_map.launch`. The proper usage of this launch file is described in Chapter C.

B.4 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 a hardware interface. We prepared a ROS launch file with the configuration of all nodes necessary for the racing task in the attached files in
`/racer-jetson/catkin_ws/src/racer/launch/race.launch`. This launch file requires two arguments: a map definition file and a circuit definition file. The proper usage of this launch file is described in Chapter C.

[11] http://docs.ros.org/melodic/api/nav_msgs/html/msg/OccupancyGrid.html
[12] http://wiki.ros.org/map_server#map_saver

B.4.1 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`[13] 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. 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.

B.4.2 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

[13] http://wiki.ros.org/costmap_2d

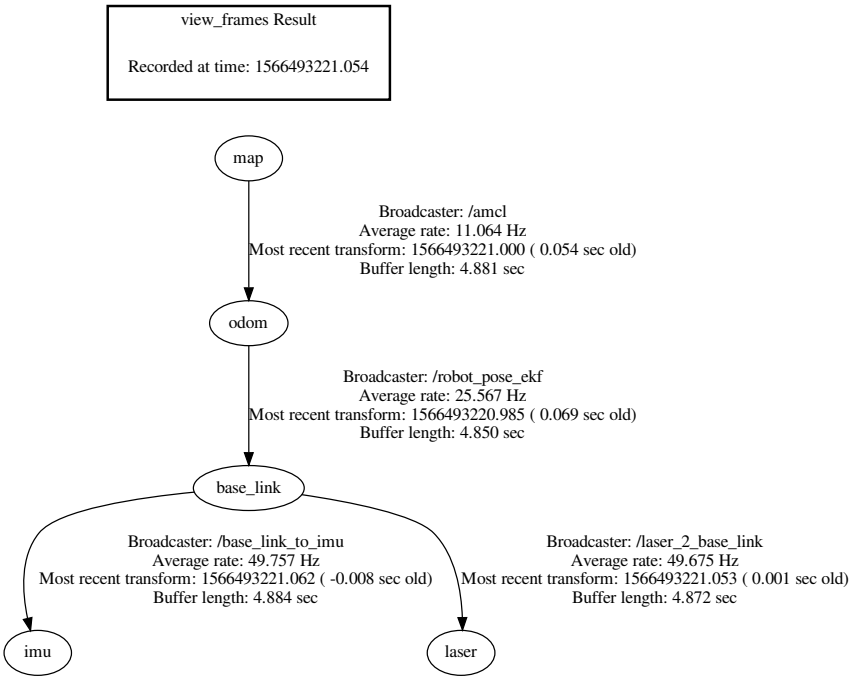


Figure B.2: The tree shows which nodes publish transformations between the coordinate frames and at what average rate the transformations are updated. This diagram is the output of the `tf view_frames` diagnostics tool.

MIT RACECAR VESC odometry ROS node[14]. Source code of the odometry node is located in the attachments in `/racer-jetson/catkin\_ws/src/odometry`.

The data from the odometry node and the IMU are then fused into a single estimate using the `robot_pose_ekf` node[15]. 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 a newer ROS package `robot_localization` 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[16]. 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

[14]https://github.com/mit-racecar/vesc
[15]https://wiki.ros.org/robot_pose_ekf
[16]http://wiki.ros.org/amcl

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 `/racer-jetson/catkin_ws/src/racer/launch/amcl_localization.launch`.

B.4.3 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 `/racer-jetson/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 `/racer-jetson/catkin_ws/src/racer_msgs/`. All of 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 2. 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 would look like this:

```
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 list of next waypoints and publishes it in the form of a custom message type `racer_msgs/Trajectory`. The planning node uses the SEHS algorithm we described in this thesis because it is more efficient than Hybrid A*. The source code of the node can be easily modify to switch to Hybrid A* if necessary.

This node will try to re-plan the trajectory as often as possible, but it is limited by a maximum updated frequency which is configurable. This is necessary because sometimes the vehicle passes close to an obstacle while following the last published trajectory and the localization algorithm briefly fails and determines the location of the vehicle in a way that the planning node detects an collision right in the initial state. The localization algorithm will recover with a new measurement of the sensors, but in the meantime, the planning algorithm could report tens or hundreds of failures because of the incorrect location. By limiting the maximum update frequency for example to 2 Hz we will still get frequent updates of the trajectory and the node will be more stable.

Most of the time during a race, the vehicle drives forward. and it does not come to a full stop or even require driving in reverse. Therefore, we prefer planning just with actions for driving forward only. If it is not possible to find a trajectory like this, we will try again now also with actions which allow driving in reverse. This way we can generate trajectories faster than if we always allowed going in reverse most of the time by limiting the number of actions and therefore we decrease the size of the nodes we open.

Following Node

This node finds the best control input for the actuators in order to follow the planned trajectory based on the current state of the vehicle using the Dynamic

Window Approach (DWA) algorithm. DWA has a set of actions it chooses from. The algorithm simulates how the state of the vehicle would change if a given action was used for a given period of time and then it selects the one, which allows it to track the reference trajectory better than the others. We give the algorithm many different actions including actions for driving in reverse.

The node publishes a standard message type `geometry_msgs/Twist`[17]. By using this message type, we mimic the output of the open source `move_base` ROS package[18]. 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.

There is an alternative following node which implements the Pure Pursuit algorithm for steering and PID controller for regulating the speed. This node is called `geometric_following_node` and it listens to and publishes the same topics as the `dwa_following_node`. We included this node even though it is not used by default because it does not have the ability to avoid previously unknown obstacles and it does not support going in reverse. Both of the algorithms simple yet interesting and the node can be useful in certain scenarios.

B.4.4 Hardware Interface

The `Arduino 2` from Figure **??** 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.

The implementation of the behavior for `Arduino 2` is included in the attached `*.ino` files in `/controller-arduino/steering_node/`.

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.

The mapping of the signal for the DC motor is a bit more complicated. The range of duty cycles $[1400\,\mu s, 1600\,\mu s]$ does not produce enough torque to set the vehicle into motion. The range of $[1000\,\mu s, 1400\,\mu s]$ corresponds to reversing, where at $1000\,\mu s$ the motor spins the fastest in the reversing direction. The range

[17] http://docs.ros.org/melodic/api/geometry_msgs/html/msg/Twist.html
[18] http://wiki.ros.org/move_base

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.

C. User Documentation

Write this chapter once the source code is stable.