Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Dynamic obstacle avoidance for autonomous F1/10 car

Bc. Jaroslav Klapálek

Supervisor: Ing. Michal Sojka, Ph.D.
Field of study: Cybernetics and Robotics
May 2019

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Klapálek Jaroslav**	Personal ID number:	**434668**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Control Engineering**		
Study program:	**Cybernetics and Robotics**		
Branch of study:	**Cybernetics and Robotics**		

II. Master’s thesis details

Master’s thesis title in English:

Dynamic obstacle avoidance for autonomous F1/10 car

Master’s thesis title in Czech:

Vyhýbání se dynamickým překážkám s autonomním autem F1/10

Guidelines:

1. Make yourself familiar with F1/10 competition, ROS project and software that was used in previous competitions.
2. Review dynamic obstacles avoidance algorithms. Consider situations like overtaking of different car or crossing an intersection.
3. Design and implement an algorithm for path planning with dynamic obstacle avoidance. Prove its functionality in simulation where the location of all cars (dynamic obstacles) is known.
4. Test the algorithm with F1/10 models. Perform experiments that demonstrate algorithm behaviour in various situations.
5. Document everything thoroughly.

Bibliography / sources:

[1] F1/10 – The Rules version 1.0, http://f1tenth.org/misc-docs/rules.pdf
[2] Urmson, C. , Anhalt, J. , Bagnell, D. et al. (2008), Autonomous driving in urban environments: Boss and the Urban Challenge. J. Field Robotics, 25: 425-466. doi:10.1002/rob.20255

Name and workplace of master’s thesis supervisor:

Ing. Michal Sojka, Ph.D., Embedded Systems, CIIRC

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **29.01.2019** Deadline for master's thesis submission: **24.05.2019**

Assignment valid until:
by the end of summer semester 2019/2020

Ing. Michal Sojka, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Ing. Pavel Ripka, CSc.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student’s signature

Acknowledgements

I would like to express my deep gratitude to my supervisor, Ing. Michal Sojka, Ph.D., for his guidance and valuable advice regarding this thesis. Also, I would like to offer my special thanks to Ing. Jiří Vlasák for his insights into the autonomous driving and his support within the F1/10 project. Finally, I would also like to extend my thanks to my parents, siblings, beloved ones, friends, and colleagues for their support throughout my study.

Declaration

I declare that the submitted thesis was developed individually and that I listed all used information sources in accordance with Methodical Guideline on Ethical Principles for College Final Work Preparation.

In Prague, 24. May 2019

Abstract

This thesis focuses on the problem of controlling intersections with autonomous cars. Upon reviewing control methods, distributed intersection control was selected. An algorithm for solving these traffic situations is described, implemented, tested in simulation, and evaluated on F1/10 platform. F1/10 platform is an RC-based car model, that is used for competitions. Also, it is used as an autonomous vehicle model for testing the algorithms.

Keywords: intersection control, intersection, mixed traffic, traffic situation, autonomous car, F1/10, Stage, ROS

Supervisor: Ing. Michal Sojka, Ph.D.

Abstrakt

Tato diplomová práce je zaměřena na problematiku průjezdu křižovatek ve spojení se samoříditelnými automobily. Za pomocí rešerše literatury bylo vybráno distribuované řízení křižovatek. Algoritmus pro řešení průjezdu automobilu křižovatkou je popsán, implementován a otestován na simulátoru a na platformě F1/10. Tato platforma je model auta, sestavená pro účast na soutěži autonomních modelů a která je dále využívána jako model skutečného autonomního automobilu pro testování algoritmů.

Klíčová slova: řešení křižovatek, křižovatka, kombinovaná doprava, dopravní situace, samořiditelný automobil, F1/10, Stage, ROS

Překlad názvu: Vyhýbání se dynamickým překážkám s autonomním autem F1/10

Contents

Figures

Tables

Chapter 1
Introduction

In the age of rising automatization, autonomous cars are slowly driving their way into our lives. With developing systems for autonomous vehicles, many problems come up. Research teams across the world try to find solutions to them. One of these problems is related to solving intersections.

The goal of this thesis is to design, implement, and verify an algorithm for intersection control.

Reviewing the problem in the literature, three possible ways for control design come up. The solution that does not require additional devices is selected and closely described.

Implementation of the algorithm is leaning towards ROS framework, that is used on robots. One of these robots is also a *scaled-down autonomous vehicle*, RC-based car model that is participating in F1/10 Autonomous Racing Competition. As an autonomous car, it is equipped with an onboard computer, camera, and LiDAR.

Verification of the implemented algorithm is not done only using the simulator, but it is also deployed on the model car, and, therefore, tested on the target platform.

The outcome of this thesis could be used for upcoming rounds of mentioned F1/10 Autonomous Racing Competition.

Chapter 2

Literature review

The problem of intersection control in relation to autonomous cars is quite discussed in recent years. Creating a reliable solution for navigating through the intersections is necessary for broad deployment of driver-less vehicles. However, focusing on the safety of traffic participants is not the only topic.

Intersections represent a part of the traffic infrastructure that induces delays to the overall transportation. Efficient navigation through these zones would reduce travelling times.

Considering the traffic full of autonomous cars lead to solutions which are, however, not usable at this moment. Therefore it is required to deal with *mixed* traffic which contains both autonomous and manually driven vehicles.

While focusing on mixed traffic, two terms are tightly connected with solving of intersections: *centralized* and *decentralized* (also *distributed*) control.

2.1 Centralized intersection control

Centralized intersection control is done by a *server*, *framework*, or *intersection manager*. These terms describe a device that is communicating with vehicles within a specific range from the intersection and assigns them specific instructions, e.g., path (which way to drive), and trajectory (how fast to drive). This approach was suggested as a *reservation-based system* in [11] and improved later in [12], [13] and [14].

The main challenges with this kind of control strategy "are associated with the heavy communication requirements and the possible occurrence of deadlocks" [54]. One has to consider security and privacy issues as well as high infrastructure costs [23]. As another noticeable drawback, only intersections with the overwatch system can be used for this strategy.

On the other hand, centralized control is mixed-traffic friendly as instructions for manually-driven vehicles can be sent via traffic lights [33].

2.2 Decentralized intersection control

In contrast to the previous method, decentralized (or distributed) control does not require any additional infrastructure device. Therefore the problem with higher cost is eliminated here [70]. Distributed control can be divided into two groups – active control and passive control. Both are designed to be used specifically for intersections with low traffic density since the peer-to-peer approach is not suited for a large number of vehicles.

2.2.1 Active decentralized control

Active decentralized control depends on V2V (vehicle to vehicle) communication to actively negotiate on intersection reservation between cars. This method was described in [70] along with the proposal of communication protocol. Messages can be one of two types – `CLAIM` and `CANCEL`. The first one is continually published to notify other vehicles of car's intentions and to "make a reservation" (which is the same technique as the one used for centralized control). Message `CANCEL` is used to cancel a reservation, e.g., when negotiated terms cannot be met.

2.2.2 Passive decentralized control

To avoid the requirement for V2V communication, passive decentralized control is suggested in [23]. This method relays on sensing (data from sensors) with the possibility to receive data from the intersection (one-way transmission). A signal from the intersection contains information about vehicles within certain zones as well as synchronization timestamp. In case this information is not used, cars ought to synchronize their clocks from GPS. This feature is along with known durations of traffic lights states used to predict them.

Chapter 3
Control design

To design an intersection control algorithm, several choices have to be made. As stated in the literature review (Chapter 2), there are three main routes: centralized, decentralized active, and decentralized passive. Each of these strategies has its benefits and drawbacks. Since we are not dealing with complex intersections and adding an intersection server is not the desired step (as it would make all experiments location dependent), *decentralized control* was selected.

Among the main reasons for this decision are:

- location independent solution (no additional device required),
- the number of cars on the track is low, therefore, centralized control is not necessary.

3.1 Scenario definition

Since designing intersection control for the real world is a difficult problem, let us narrow the problem by specifying which scenarios are the target. From these scenarios, we derive requirements for control design and upcoming implementation. These rules take into account the platform (see Chapter 4).

1. The scenario is a track (road limited by tall borders/walls) placed on a flat surface.
2. The testing track is closed and fully controlled environment (only selected objects are allowed to enter).
3. Only two cars are allowed on the track at once.
4. Each car has to broadcast its position and velocity with minimal frequency 10 Hz.
5. Both cars have to obey traffic rules (explained below).
6. Only 4-way cross-shaped intersections are allowed.
7. When driving through the intersection, cars are not allowed to turn.

Traffic signs. We consider the following two traffic signs. They are complementary – if one of them is used, the other one has to be placed on the perpendicular road.

- YIELD sign, which forces the car to yield and let the other car pass.
- PRIORITY sign, which gives the car priority and, therefore, allows it to pass the intersection as first.

Traffic rules. Taking the traffic signs into account, all cars that are in the scenario have to obey the following rules. When applying the rules, we proceed from the top. When a condition is met, the corresponding rule is applied. Otherwise, we continue to the next one.

- When no other car is detected, the car can pass the intersection.
- When both cars are driving through the intersection from the opposite directions (they do not intersect each other paths); the car can cross the intersection.
- When YIELD traffic sign is detected in front of the intersection; the car has to let the other one pass.
- When PRIORITY traffic sign is detected in front of the intersection; the car can pass the intersection.
- When the other car is approaching from the right side; the car has to let the other one pass.
- When no rule is used; the car can pass the intersection.

3.2 Marking convention

Before the algorithm description let us explain used terms and notation. Marking is noted in Table 3.1. All used angles are represented by numbers in an interval $\langle -180°; 180°)$.

Car A, B. Letter A denotes the car currently running the algorithm. Letter B denotes the other car on the track. Each car is represented as a circle with a defined radius. For some computations, we consider only its origin, therefore, it is the same as representing the car with a mass point.

Car coordinate frame. Each car has its coordinate frame: the y-axis corresponds to the axis of steering wheels, and it points to the left; the x-axis is placed along the longer side of the car, and it is pointing forward. Frames are labeled with the same letter as the car which it belongs to. For the global coordinate frame, letter O is used. When a variable is expressed with relation to a selected coordinate frame, its marking is used as a top index. For variables without the top index, the global coordinate frame is assumed.

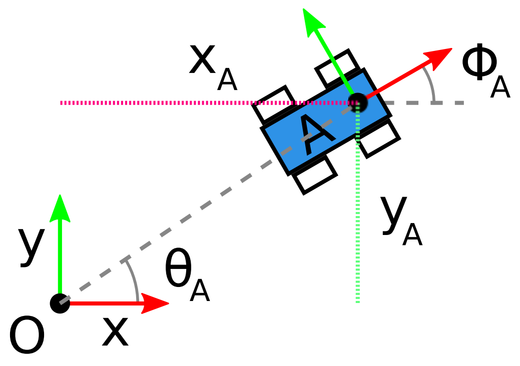


Figure 3.1: Car A shown in a global coordinate frame

Path p. The path is a line created by connecting past, present, and future positions of a car. However, when referring to this term later, only future positions are meant.

Trajectory. By a term trajectory control action for reaching the next point in a path is meant. This applies for reactive (i.e., without knowledge of past action) planners.

Collision point C. Collision point is an intersection of paths of both cars. Therefore, it is a point where the two cars would collide if each one of them were represented by a mass point (their radius is zero).

Collision area C^+. Collision area is a zone where the two cars might collide if they continue in their directions. It is defined as a common space for two "inflated" paths – the inflated path is created by applying the car's radius in each point of a path. Therefore it is defined as a Minkowski sum of a path and car's radius. Collision area has a center (collision point), size and with relation to the cars, *entry* and *exit* points are used. The entry point is the first position on a car's path that belongs to the collision area; the exit point is the last.

Heading ϕ. Heading is an angle between x-axis of one coordinate frame and x-axis of another coordinate frame, e.g., ϕ_A^B stands for heading of car A in the coordinate frame of car B. Therefore, ϕ_A^A is zero.

Collision angle. When two cars are about to collide, the angle between their headings is called collision angle. When referring to the collision angle with respect to the car A, it is the heading of the car B in car A coordinate frame (i.e. ϕ_B^A) and vice versa.

Orientation θ. Term orientation is used for the angle between x-axis of one coordinate frame and a segment from the origin of this frame to the origin of another coordinate frame – θ_A^B stands for orientation of the car A with respect to the car B.

Term	Description
O	Origin of the used map; Global coordinate frame
A, B	Cars; Coordinate frames of the cars
x, y	Coordinates
v	Velocity
r	Radius
ϕ	Heading
θ	Orientation
p	Path
V_A^B	Variable of car A in coordinate frame of car B
d_{AB}	Distance between cars A and B

Table 3.1: Used symbols

3.3 Algorithm description

In [42] design called *hierarchical Fuzzy Rule-Based System* was presented. This strategy is used as a reference for our algorithm. This particular decision system was selected because it is straightforward, and it does not require a map. However, our design is stripped of fuzzy sets [80] that are in the paper used for optimization (which is outside the scope of this thesis).

3.3.1 Input

At first, the algorithm receives information about both cars. These data contain their poses and velocities. Poses are referred to the origin of the map O and contain the position of the cars and their headings. Velocities consist of two parts – longitudinal speed (forward/backward) and lateral steering (right/left).

It means that the algorithm requires multiple information about each car. However, it is possible to work only with the position – heading and velocities might be computed from two consecutive locations.

As mentioned in Section 3.1, traffic signs are also considered in the scenario specification. Therefore the last input of the algorithm contains detected traffic signs. However, this input is optional.

3.3.2 Detection of situation

Data about car B are transformed into the reference frame of car A:

$$\begin{aligned} x_B^A &= d_{AB} \cdot \cos \theta_B^A, \\ y_B^A &= d_{AB} \cdot \sin \theta_B^A, \\ \phi_B^A &= \phi_B^O - \phi_A^O, \end{aligned} \tag{3.1}$$

where coordinates of car B are expressed with respect to the car A – x_B^A is an x-axis position, y_B^A is a y-axis position, and ϕ_B^A is a heading. Variables ϕ_A^O, and ϕ_B^O are, respectively, headings of car A and car B in the global reference frame. d_{AB} is the distance between cars; θ_B^A is the orientation of car B from the reference frame of car A. These last two variables are computed using equations:

$$\begin{aligned} d_{AB} &= \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}, \\ \theta_B^A &= 180 \cdot \frac{atan2(y_B - y_A, x_B - x_A)}{\pi} - \phi_A^O, \end{aligned} \tag{3.2}$$

where x_A, y_A are x, y positions of car A and x_B, y_B are x, y positions of car B.

Transformation of coordinates allows us to describe the current situation (with respect to the car A) using this method:

- $x_B^A > 0$ means that car B is ahead of car A,
- $x_B^A < 0$ means that car B is behind car A,
- $x_B^A = 0$ means that car B is on the side of car A,
- $y_B^A > 0$ means that car B is on the left from car A,
- $y_B^A < 0$ means that car B is on the right from car A,
- $y_B^A = 0$ means that car B is in front of/behind the car A.

Let us note that using strict equality is not mandatory – for real-world usage, an interval would be used instead. On the other hand, for (at least) the simulation, these two conditions may be completely omitted as they describe only four corner cases.

Using combinations of relations above lead to eight possible situations. This number can be increased to a higher one by adding extra conditions (e.g., far ahead).

Collision angle may be used to discard certain situations. If this angle is low (close to the x-axis, in other words, close to integer multiples of 180°) intersection is not being solved, and therefore algorithm can stop.

3.3.3 Collision detection in space

Collision detection is a term that covers methods for predicting collisions. Collision is a situation in which both vehicles touch/crash into each other.

For collision detection, we assume that both cars will continue in driving with the same heading and speed. This assumption allows us to predict their paths and find possible collision point.

In [42], the differentiation of more situations is done. However only cases with collision angle close to ±90° are considered there. Let us loosen this constraint and check for possible collisions in an unlimited angular area.

At first, collision points are used for this. Collision point approach is not able to detect certain collision situations. Considering the cars as mass points makes it fail in predicting collisions caused by their size. However, these collisions are not intersection related, as the cars are not both driving towards the collision point. This is also shown in Figure 3.2.

The collision of two cars is possible (in space) if and only if:

$$\phi_B^A < 0 \implies \hat{\theta}_B^A < \phi_B^A < 0, \tag{3.3a}$$

$$\phi_B^A > 0 \implies \hat{\theta}_B^A > \phi_B^A > 0, \tag{3.3b}$$

where ϕ_B^A is heading of car B with respect to car A and $\hat{\theta}_B^A$ can be described as "inverse orientation" – orientation from car B to car A but in the coordinate frame of car A. This situation is also shown in Figure 3.2. If conditions are not met, the algorithm can end.

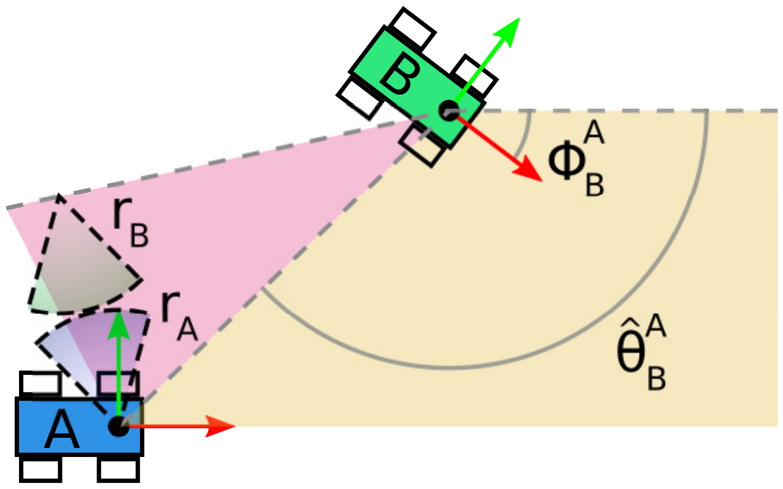


Figure 3.2: Visualization of Equation 3.3a. Orange area corresponds to possible collisions that are detected, red area contains possible collisions that are not intersection related, and, therefore, not detected.

Since the collision between the cars is now marked as "possible", the location of the collision point can be computed. For future calculations, the point is expressed using polar coordinates. The phase of the collision point is already known (it equals to car's heading) and, therefore, only the magnitude is missing. However, magnitude equals to the distance, which is computed by these equations:

$$\begin{aligned} d_{BC} &= -\frac{y_B^A}{\sin\phi_B^A}, \\ d_{AC} &= x_B^A + d_{BC}\cdot\cos\phi_B^A, \end{aligned} \tag{3.4}$$

where d_{BC} is the distance between the car B and the collision point C, d_{AC} is the distance between the car A and the collision point C, x_B^A, y_B^A and ϕ_B^A are, respectively, x, y, and heading of car B in the coordinate frame of car A. In case that computed distance would be negative, the car is heading away from the collision point, and, therefore, no collision may occur.

However, to address one particular corner case (from a planner's point of view), we continue with the algorithm if the distance between the cars is less then the sum of their sizes. By this, behaviour during waiting on the intersection is resolved.

Since collision points do not take vehicle dimensions into account, a collision area is used from now on. Size of this zone guarantees that the car inside the area is not endangered by any vehicle waiting outside. As mentioned before, the center of the zone is the collision point.

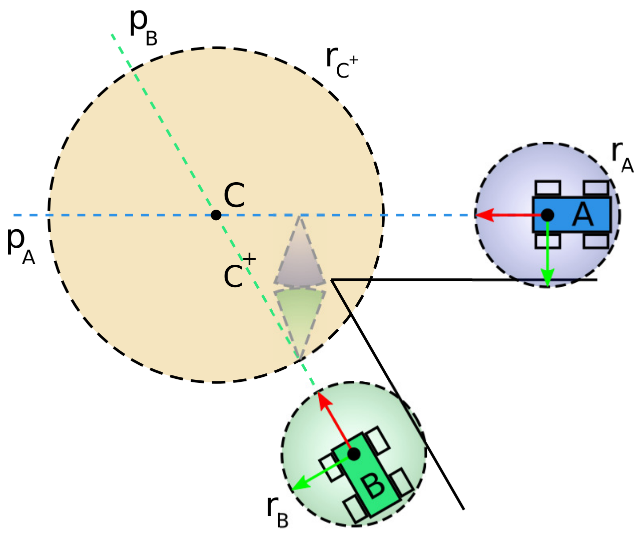


Figure 3.3: Two cars approaching the intersection. Circle sectors in the middle illustrate the smallest distance between the cars, thus, the reason for computing r_{C+} with collision angle.

The size of the collision area is defined by sizes of the cars and the collision angle. Using only dimensions of the cars lead to detecting certain collisions too late, as they would be caused by sharing part of the path when driving through the area. Adding collision angle into the equation solves this issue, as shown in Figure 3.3. The approximate radius of the collision area can be computed using this equation:

$$r_{C+} = \frac{r_A}{\sin|\phi_B^A|} + \frac{r_B}{\sin|\phi_B^A|}, \tag{3.5}$$

where r_{C+} is a radius of collision area, r_A and r_B are radiuses of cars A and B, ϕ_B^A is heading of car B with respect to car A (also collision angle). By combining this value and the collision point, distances to the entry and exit points of the collision area may be computed:

$$\begin{aligned} d_{AC+}^{in} &= d_{AC} - r_{C+}, \quad & d_{AC+}^{out} &= d_{AC} + r_{C+}, \\ d_{BC+}^{in} &= d_{BC} - r_{C+}, \quad & d_{BC+}^{out} &= d_{BC} + r_{C+}, \end{aligned} \tag{3.6}$$

where distances marked with *in* are distances between cars and points where they would enter the collision area, distances marked with *out* are distances between cars and points where they would exit the collision area, therefore freeing up the intersection.

3.3.4 Collision detection in time

Variables computed in Equation 3.6 can be used for checking whether the collision can happen. Until now, only headings and relative positions were considered. By adding velocities collision can be confirmed or discarded.

Let us calculate times for entry/exit points of the collision area using these equations:

$$t_{AC+}^{in} = \frac{d_{AC+}^{in}}{v_A}, \quad t_{AC+}^{out} = \frac{d_{AC+}^{out}}{v_A}, \quad t_{BC+}^{in} = \frac{d_{BC+}^{in}}{v_B}, \quad t_{BC+}^{out} = \frac{d_{BC+}^{out}}{v_B}, \tag{3.7}$$

where t denotes times until the cars arrive at the entry/exit points of the collision area, v_A and v_B are, respectively, velocities of cars A and B. Combining *in* and *out* times for a car creates an interval in which the car is occupying the collision area. Using just the raw times can be used to distinguish between these situations:

- $t_{AC+}^{in} > 0 \iff$ car A is driving towards the collision area,
- $t_{AC+}^{out} < 0 \iff$ car A is past the collision area,
- $t_{AC+}^{in} < 0 \;\; \& \;\; t_{AC+}^{out} > 0 \iff$ car A is within the area,
- $t_{AC+}^{in} < t_{BC+}^{in} \iff$ car A arrives in the area before car B,
- $t_{AC+}^{out} < t_{BC+}^{out} \iff$ car A leaves the area before car B,
- $t_{AC+}^{out} < t_{BC+}^{in} \iff$ car A leaves the area before car B arrives,

while this last one means that the cars are not sharing time inside the area, and, therefore, the collision does not occur.

3.3.5 Trajectory planning

As a matter of trajectory planning, this algorithm relies on using an external planner. Trajectory generated by this planner is modified by intersection control. In case the car is going to yield, there are three possible ways how to resolve ongoing collision:

Stop before the area. As a first and basic strategy, stopping the car is used. It is also used later within the implementation part. If the car is about to yield to the other car, it will stop at the entry point of the collision area.

Speed up the car. Another strategy is to speed up the car to pass the collision area before the other vehicle. In this case, target velocity is set to:

$$v_A \geq \frac{d^{out}_{AC+}}{t^{in}_{BC+}}. \tag{3.8}$$

Slow down the car. In contrast to the previous strategy, the car can be slowed down to arrive at the collision area after the other vehicle leaves. In this case, target velocity is set to:

$$v_A \leq \frac{d^{in}_{AC+}}{t^{out}_{BC+}}. \tag{3.9}$$

3.4 Localization

As mentioned in the algorithm description (Section 3.3), positions, headings, and velocities of both vehicles are required. In this section, a quick review of methods for obtaining this information is provided. Only solutions that are considered for the implementation are mentioned.

3.4.1 Particle filter

As a source of self-localization, a particle filter [9] can be used. It is a Monte Carlo class algorithm that is used to localize robot – estimate its pose (i.e., position and heading) – in a predefined map.

Particle filter works with particles (also pose hypotheses) that are randomly spread across the map of the environment. The method is split into three parts (which are repeated in the same order):

- *Prediction phase* uses a motion model of the car and detected movement to change position and orientation of the particles.
- *Update phase* uses a measurement model of the sensors and received readings to update belief in the individual particles. Each measurement is compared to its counterpart generated by the particle. The basic method for this generation is called *ray casting*.
- Last part resamples particles (generates a new set of particles) according to updated beliefs.

As the algorithm proceeds, the particles are gradually gathering around the real position of the vehicle. Only requirements on the used method are (besides the best possible accuracy): common origin and map rotation for all traffic participants.

For outdoor usage, the particle filter would be replaced by absolute position received from, e.g., GPS.

3.4.2 Intervehicle communication

Since the particle filter (Subsection 3.4.1) can only estimate the position of one car, another solution needs to be deployed to retrieve data about the other vehicle.

If the particle filter is used on both cars, estimated poses may be shared between them. As mentioned in the literature review (Chapter 2), intervehicle communication can be used to receive/send instructions (then we are dealing with an intersection manager) or to exchange car information with other vehicles.

Also, terms *V2V* and *vehicle-to-vehicle communication* are used when referring to this.

3.4.3 Object detection

As an alternative for retrieving data about the other car, object detection approach can be used. Described algorithm (Section 3.3) relies only on relative coordinates between the cars. Therefore, there is no need for the vehicle to self-localize, since it does not affect the detection of the other car.

For object detection various sensors can be used, e.g., monocular camera [79], LiDAR [52], RaDAR, SoNAR, infrared camera [10].

3.4.4 Wheel odometry

Movement estimation of a vehicle is called odometry. By measuring the rotation of wheels, velocity, and relative position can be retrieved. Even though this method is not very accurate (as there are papers that solve this problem [5]), it can be still used for quite an accurate velocity estimation and particle filter correction [37].

Since the position of the cars is estimated by particle filter, wheel odometry is used for estimating the velocity on the platform.

Chapter 4
Platform

Intersection control strategy described in Chapter 3 is designed for implementation and verification on F1/10 platform. Vehicles F1/10 are RC-based models in ratio 1:10 with regular cars.

The project of F1/10 cars was started to compete in *F1/10 Autonomous Racing Competition* – an international event which encourages student teams across the world to pursue autonomous driving. This contest is being held on conferences (namely CPSWeek and ESWeek) twice a year.

Attending the racing competition is not the only usage for these models. Thanks to their real autonomous car-like design, they can be used for implementing, verifying, and testing of algorithms that can be later ported to a full-scale autonomous car.

Advantage of using a model car instead of a real one is not just its price. All tested scenarios are much more controllable as they can be created on a 1/10 scale. Also, all crashes with a model car are much less severe.

4.1 F1/10 competition

F1/10 Autonomous Racing Competition that was already mentioned earlier, is an international competition, where teams of students complete a predefined task. At first, this task was only related to building a car and driving it through the track as fast as possible. Latest competition added also racing of two cars.

Even though the assignment of each competition may be a little bit different, the goal stays the same – challenge the student teams appropriately according to the overall level of all contestants. This means that over time, the competition may evolve to include also some traffic situations like intersection solving.

We have attended the 2nd and 3rd F1/10 Autonomous Racing Competitions that were held, respectively, in Porto (April 2018) and Torino (October 2018). Our team was called "Řeřicha". We have won the Porto race and later we were third in Torino.

4.1.1 Past assignments

As said, assignments of the competition are evolving. To show the challenges that had to be met over time, a list of them follows.

Building the car. At first, each student team has to build their car. To create a competition approved model, several conditions have to be met (the list is updated to the current version [17]):

- A 1/10 scale rally car chassis equivalent to the Traxxas model 74054 type is allowed.
- Only the use of stock tires, or equivalent - in size and profile, is allowed.
- NVidia Jetson TX2 or an equivalent capability processor or anything of a lower spec is allowed.
- Hokuyo UST-10LX or an equivalent LiDAR range sensor or anything with a lower spec is allowed.
- Multiple LiDARs are allowed, as long as they are all equivalent to, or lower spec than, the Hokuyo UST-10LX.
- No restrictions on the use of cameras, encoders, or custom electronic speed controllers.
- Brushless DC motor equivalent to Velineon 3500 or anything of a lower spec is allowed.

On the other hand, since the last competition, even the cars that do not follow these rules are able to compete in a so-called *Open Class.* Cars that compete in the main part (called *Restricted Class*) may attend this as well. This side-competition can be used for testing better equipment or other not allowed methods.

Algorithm implementation. Along with running the car, installing required packages, etc., the selected algorithm has to be implemented to be eligible to participate. For this, lectures [18] from the organizers can be used. As early competitions were only about driving through the track, implementing a simple reactive algorithm was enough. This part of the contest is still a part of the event named as a *Time Trial Race.*

Time Trial Race. The goal of the first racing task is to complete as many laps as possible within a specified time. In case that two teams get the same number of laps, time of the fastest lap determines the winner. During this race, only one car is allowed on the track.

Obstacle avoidance. Avoiding the obstacles was not a rated part of the competition, but it served more like a preparation for the next task. We were the first to demonstrate quite reliable static obstacle avoidance during the competition in Porto. Dynamic obstacles were avoided as well, but the success rate was lower.

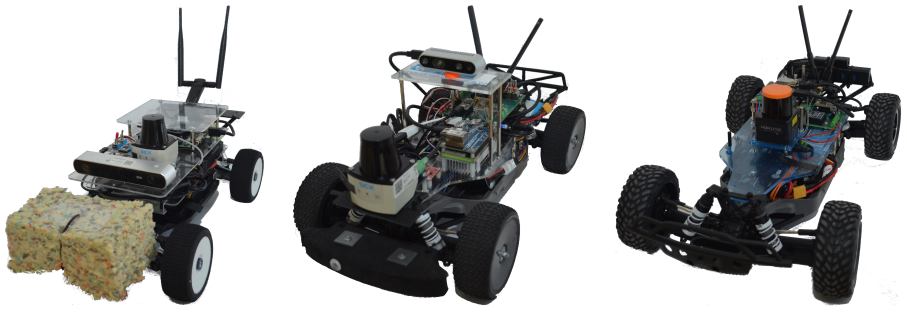

(a) : First model **(b) :** Second model **(c) :** Fourth model

Figure 4.1: Our fleet of F1/10 cars

Head to Head Race. After the 3rd competition, a new racing task was revealed. Since multiple teams were able to demonstrate obstacle avoidance racing of multiple cars on the track became possible. Head to Head Race is a modification of Time Trial Race with an exception that two cars are racing at once.

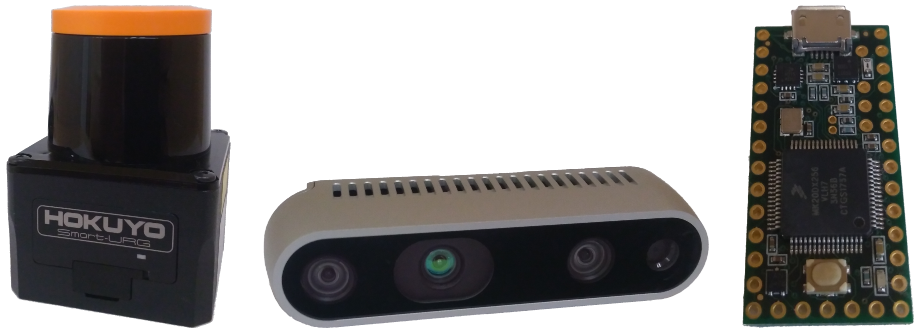
4.2 Hardware

Currently, we have three cars built (see Figure 4.1), and the fourth is being created. The first model (Figure 4.1a) was created as a part of a master's thesis in 2017 [69]. The second model (Figure 4.1b) was created a year later to attend the competition in Porto. The third model was created as a part of thesis [16] this year. The fourth model (Figure 4.1c) is based upon the third model, and it was created at the beginning of this year. Therefore we are able to do experiments with multiple vehicles on the track. Since the essential components of all cars are quite similar, only the latest model is closely described. Its hardware list is in Table 4.1.

Part	Model
CPU	nVidia Jetson TX2 on Orbitty Carrier
CPU for engine	Teensy 3.1
Chassis	Traxxas Slash 1:10 4WD VXL TQi TSM OBA RTR
Engine	Velineon 3500
Batteries	2S LiPo (for engine only), 3S LiPo
Controller	FOCBOX VESC (Electric speed controller)
Transceiver	Traxxas TQi 5CH TSM (manual)
LiDAR	Hokuyo UST-10LX
Camera	Intel RealSense D435

Table 4.1: Hardware list of the car

Chassis selected for this model does not satisfy the requirement from organizers (Subsection 4.1.1). However, this was discussed with them, and they allowed it to attend the competition.



(a) : Hokuyo UST-10LX **(b) :** Intel RealSense D435 **(c) :** Teensy

Figure 4.2: Selected components from the newest model

4.2.1 Camera

Currently, we are using RealSense D435 (Figure 4.2b) manufactured by company Intel. Its additional features can be useful when solving future assignments.

Along with color camera with a wide field of view and global shutter, a depth camera is provided. It can be used as another measurement device than relying only on LiDAR. To improve depth perception, active infrared projectors are used to illuminate objects in the environment.

4.2.2 LiDAR

LiDAR is an abbreviation for Light Detection and Ranging. It is a device that is able to measure distances to nearby obstacles using laser rays that are reflected to the environment by a rotating mirror.

We are currently using Hokuyo UST-10LX (Figure 4.2a). It is a two-dimensional LiDAR (it can measure in only one plane) with a field of view 270°. Its frequency is 40 Hz, which means that it can measure distances to all surrounding obstacles every 25 ms.

On large-scale autonomous cars, LiDARs are used as well. However, in that case, it is mostly a three-dimensional one.

4.2.3 Teensy board

The Teensy board (Figure 4.2c) is a microcontroller that is used for controlling the engine and servo of the car by PWM-modulated signal. Separation of this functionality provides certain reliability that it is done in time.

Using the Teensy board also comes with another benefit – the car can be remotely stopped by triggering the transmitter (manual control). This can be useful when testing new algorithms as it limits the damage caused by a crash.

4.3 Software

The main computer on the car – nVidia Jetson TX2 – is running Ubuntu 16.04 as an operating system. Therefore, all packages compiled for an *arm* version of this distribution can be used during the development.

Selection of Ubuntu 16.04 is not a random choice. This version is supported by nVidia company, and, moreover, supported by Robot Operating System, which is a main requirement for the platform. Also, using ROS is encouraged by competition organizers [18].

4.3.1 Robot Operating System

Robot Operating System (or ROS) is "an open-source, meta-operating system" [63] which contains "collection of tools, libraries, and conventions" [45]. It can also be described as a distributed framework.

Master. The heart of the ROS is called ROS Master [59] or ROSCore. It is used to register other nodes in the system, to track their subscriptions and publications. Role of the Master is to provide a way for the nodes to locate one another. After this, they can communicate outside the Master. It is also responsible for providing a Parameter Server which stores parameters for the nodes.

Node. The main building component of ROS is called node. Node is any application which can be run under ROS. Nodes can be designed, implemented and tested individually. Grouping them, to complete complex tasks, is done at runtime.

Package. Packages are containers that gather up nodes. They were created for easier distribution of code [63]. Each package has its name, list of dependencies, and other required files. Packages are organized in workspaces (folders).

Message. Messages are simple data structures composed of primitive types (e.g., strings, integers) and arrays of these types [60]. These messages are used to communicate between nodes.

Topic. Topics are named buses for exchanging messages. Each topic has specified one type of message that is used there. Nodes subscribe to topics (receive messages from them), publish to them (send messages), or both [64].

Launch file. Special status has a command `roslaunch` that works with launch files [7]. These files contain instructions which nodes to run and what are their parameters and arguments.

4.3.2 Functional architecture

As a part of building third car functional architecture for the platform was designed and created. It is based on architecture proposed in [1]. Project is split into three main parts as shown in Figure 4.3. Each part can be designed and implemented on its own. Therefore each member of the team can create a new package that is only limited by constraints on its input/output. Putting parts together creates a *control chain* for the vehicle.

Since this is still a work in progress, it is possible that the architecture will be changed over time as more experience with the platform is gained.

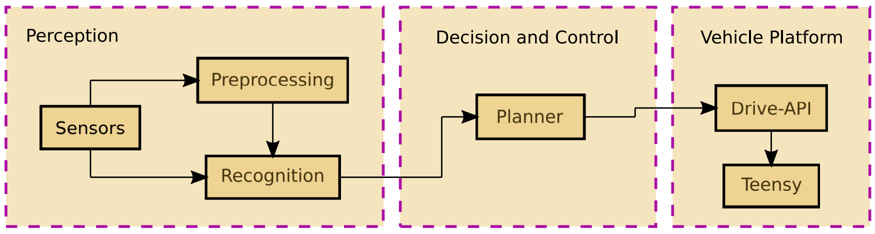


Figure 4.3: Functional architecture of the platform

Perception

The first part of the architecture is called Perception. It is responsible for observing and understanding the environment using sensors. Perception is split into three blocks. Even though our models are quite similar, differences in their equipment can be found. Design of the Perception takes this into account. Therefore it is possible to substitute a certain model of a sensor with a different one of the same type.

Sensors. The first block is responsible for retrieving information from the environment and other vehicles. Packages within the Sensors block are mostly establishing a connection with the sensors, reading data from them, and converting these data to ROS compatible messages. All packages are organized in folders by the type of the sensor they are communicating with – e.g., LiDAR, camera. Output messages are standardized for each sensor type.

Preprocessing. The second block is used for converting, editing, precomputing, cleaning, of data received from the sensors. Packages from Preprocessing should help maintain certain compatibility between various sensors of the same type and recognition algorithms. Also, using these packages should ease up the implementation of other nodes since some of the required operations may be already solved here.

Recognition. Third block of the Perception contain packages that are used, e.g., processing received data, detecting obstacles, understanding environment, estimating location. Results of these algorithms are stored inside several messages which describe estimated position (`geometry_msgs/Point` [56]), estimated pose along with the velocity (`nav_msgs/Odometry` [62]) and detected obstacles (`obstacle_msgs/Obstacles`). Last message is a custom one (it is not a part of standardized messages) and it is defined as follows:

```
# Obstacle message
obstacle_msgs/SegmentObstacle[] segments
    └── geometry_msgs/Point[2] points
obstacle_msgs/TriangleObstacle[] triangles
    ├── geometry_msgs/Point edge_a
    ├── geometry_msgs/Point edge_b
    ├── geometry_msgs/Point edge_c
    └── geometry_msgs/Vector3 velocity
obstacle_msgs/CircleObstacle[] circles
    ├── geometry_msgs/Point center
    ├── float64 radius
    └── geometry_msgs/Vector3 velocity
```

where triangles and circles also contain estimated velocity of the obstacle (`geometry_msgs/Vector3` [58]) which consists of speeds along three axes. This velocity is applied to the center of the object.

Decision and Control

Decision and Control is the name of the second part of the architecture. It is responsible for trajectory planning using data received from Perception. Packages inside can be theoretically chained or put in parallel to create more complex behaviours. In fact, behaviour trees [6] along with the state machines are planned to be here in the future.

Created plans are published using custom command messages that are being caught by Drive-API, which is described later.

Vehicle Platform

Differences in the equipment of the models were already solved in Perception. However, as stated in Subsection 4.2.3 engine and servo are controlled via PWM signals. Parameters of these signals differ between the cars. To overcome this issue, the third part called Vehicle Platform was designed to be hardware independent (from a software point of view).

Both parts of Vehicle Platform are subscribed to topic `/eStop` (boolean messages). This topic is used as an *emergency stop* which forces the car to stop immediately.

Drive-API. The first part of Vehicle Platform is Drive-API. It is a framework that is used to make planners hardware independent. Drive-API can be launched as a ROS node or imported as a Python module to allow controlling the car outside of ROS. Hardware related constants are stored inside ROS Parameter Server, and they are used by Drive-API to convert received values to duty cycles.

Currently, custom `drive_api/drive_api_values` messages are used to send commands to this application. This message type is defined as follows:

```
# Values for drive_api
float64 velocity
bool forward
float64 steering
bool right
```

where `velocity` and `steering` are allowed to contain only values from interval $\langle 0; 1\rangle$.

This message design allows selecting precisely the lowest speed possible by setting `velocity` to 0. Stopping and resetting the steering is done by sending a negative number. However, this approach is subject to change when PWM-to-speed identification is made, which allows specifying required speed in SI units.

Teensy. The second part of Vehicle Platform, and, also, the last part of the whole control chain is Teensy. It is a package with only one node that communicates with the Teensy board that is attached to the cars. As said in Subsection 4.2.3, this board is responsible for generating PWM signals to the engine and servo. Duty cycles of the signals are sent from the Drive-API via `teensy/drive_values` custom message which is defined as follows:

```
int16 pwm_drive
int16 pwm_angle
```

where `pwm_drive` is used for controlling the engine and `pwm_angle` is used for controlling the servo. Their values from $\langle 0; 65535\rangle$ interval correspond to $\langle 0\%; 100\%\rangle$ of the duty cycle.

The Teensy board itself is strongly hardware dependent as it is flashed with parameters of the car. However, this could be resolved in the same way like Drive-API – receive these values from the Parameter Server.

Chapter 5
Implementation

Having designed an algorithm for solving the intersections (Section 3.3) and having introduced the target platform (Chapter 4), implementation can be done.

As a programming language Python 2 was selected. Its alternative – Python 3 – is not officially supported by ROS. Using `rospy` module's API, one can easily interface with ROS Topics, Services, and Parameters (see Subsection 4.3.1). Also, using Python/`rospy` combination is well suited for quick prototyping and testing within ROS [8] since compilation is not required. On the other hand, using Python has lower performance than the same algorithm written in C++/`roscpp`, which is designed to be a high-performance library [53].

5.1 Intersection solver

Implementation of the algorithm from Section 3.3 is provided as a single package placed within Decision and Control (see Subsection 4.3.2). It is designed to run in series with another trajectory planner. If a planner manager (node that decides between multiple plans) were available, these two planners would be running in parallel instead. This approach is also suggested in Subsection 4.3.2 and associated paper [1].

Standing as another planner within the ROS chain allows reusing the plan created by its predecessor. Therefore, the created package serves more like a *plan modifier*. However, it is still categorized as a trajectory planner as it outputs a trajectory.

This design comes with a huge benefit. Planning a whole new trajectory (thus ignoring the other planner) could be counterproductive to the global plan. Even though modifying the speed affects the plan, the other planner does not lose (in a specific way) its control of the vehicle. Also, this makes the planner responsible for navigating through the intersection as the algorithm does not influence the steering. However, this could be also considered as a drawback.

5.1.1 Inputs

The algorithm description (Section 3.3) starts with specifying the input variables. Their adaption to ROS framework is explained in the following lines.

Car A. Position, heading, and velocity of car A is provided via `/odom1` topic which uses Odometry messages defined by `nav_msgs/Odometry` [62]. This type contains all the required information. The velocity of the vehicle is specified in a "*car-like* style" – its longitudinal speed (forward/backward) and its lateral (angular/steering) speed.

Car B. Data about the other car are received on topic `/odom2`. The message type is identical to car A.

Traffic signs. As later explained in Section 5.3, AprilTags are used instead of traffic signs. Detected tags are received from `/tag_detections` topic (message `apriltags2_ros/AprilTagDetectionArray` [36]).

Planner. Trajectory planned by the other planner is obtained using a remapped `/drive_api/command` topic, which is used for communicating with the Vehicle Platform (Subsection 4.3.2). It means that the output of the other planner is redirected to the intersection controller, which sends modified instructions to the Drive-API.

5.1.2 Outputs

The output of the algorithm can be one of two types – the first one is standardized and used as a default method. The other one is left as a compatibility layer and as a backup solution in case that Drive-API is not an option (i.e. it is not available).

Drive-API command. The first type of output is created for ROS chain described in the functional architecture (Subsection 4.3.2). It is triggered by messages received from the other planner. An obtained trajectory is appropriately modified and published to a topic for Drive-API. This method supports all three strategies described in Section 3.3 – stopping, speeding up and slowing down.

eStop command. The second type is created for older planners that were designed before the introduction of the Drive-API. These planners are feeding instructions directly to the Teensy board. Since using `/eStop` can only stop the car, this method supports only one strategy – stopping the vehicle before the collision area.

5.2 Localization methods

In Section 3.4, several ways of obtaining poses and velocities of both vehicles were introduced. Following subsections change the focus on their platform integration.

5.2.1 Particle filter

As mentioned in Subsection 3.4.1 particle filter is an algorithm used for self-localization in a known map. This method can be used for estimating position, and heading.This estimate is published together with velocity from wheel odometry (Subsection 5.2.4) published as an Odometry message (`nav_msgs/Odometry` [62]).

Particle filter implementation that is used for autolocalization is based upon *Compressed Directional Distance Transform* (CDDT) [76]. In contrast to the ordinary ray casting method, this novel approach significantly lowers the computation time with an only minor increase in memory usage.

This solution was designed for MIT RACECAR [39], autonomous car platform that is very similar to ours. Implementation of this particle filter localization algorithm is ROS compatible and freely available on GitHub [38]. Integration to F1/10 platform is closely described in [32].

Mapping the environment. Since using the particle filter requires knowledge of the environment, a map has to be created. Package `hector_mapping` [30] is used for this task. It requires a stream of measurements from LiDAR. These data are processed by *scan matching* – a function which aligns laser scans with an already existing map and stores new data into it [31].

5.2.2 Intervehicle communication

Particle filter from the previous subsection estimates the position of only one vehicle. As proposed in Subsection 3.4.2 intervehicle communication can be used to exchange estimated data between two (or generally multiple) cars.

Full-scale autonomous cars would be using specialized V2V-enabled devices. Because our platform is not equipped with such hardware, this functionality has to be solved differently.

As a way of communicating between cars, TCP/IP packets over WiFi were selected. Each car launches its server for incoming transmissions and one client for sending the messages to the other car. Even though this solution may not be ideal for large-scale scenarios, its simplicity, and hardware undemandingness (as it does not require additional components) overcome its drawbacks.

For this purpose, two nodes were created – server node and client node. Each one of them requires several arguments:

- target IP address,

- target port,
- name of topic – server for publishing, client for subscribing,
- type of message on this topic,
- (server only) size of the buffer for received data,

where the type of the message is not required if the topic is already running. In that case, nodes inherit used message.

5.2.3 Object detection

For certain scenarios usage of V2V might be problematic (e.g., long distances, jammed environment). In that situation, method of detecting the other car would be handy. For this, obstacle detection using LiDAR data can be used. Even though the object detection package is imported to the platform, this part is more theoretic as this behaviour is not implemented, yet.

Object detector algorithm, which is described in [52], takes measurements from LiDAR as its input. Distance readings are subject to these procedures:

1. *Grouping* – at first points are split into groups according to the distance between two consecutive measurements; if it is too large (which is defined by a variable) new group is started,
2. *Splitting* (recursive) – groups larger then defined value are examined for possible splitting, which occurs if a point is too far away from a line, that is fitted through extrema of the group,
3. *Segmentation* – modified groups are approximated with a segment, parameters are computed via least squares regression,
4. *Segments merging* – extracted segments are tested for possible merging, segments close to each other are temporarily merged and if this merged segment is collinear to extrema fitted line (same technique as for splitting) new segment is created,
5. *Circles extraction* – each segment is converted to triangle (one side is the segment, rest of the triangle is placed away from the origin) and a circumscribed circle is created, which is not discarded if its radius is within a specified interval.

ROS compatible implementation of this algorithm is publicly available on GitHub [51]. The published message `obstacle_detector/Obstacles` [50] contains lists of circles and segments. They are later converted to our message type that is described in Subsection 4.3.2. This particular implementation also features obstacle tracking – a method which assigns velocity to the circle obstacles.

5.2.4 Wheel odometry

Wheel odometry (mentioned in Subsection 3.4.4) can be used for estimating the velocity of the vehicle. This is also used within the platform.

The platform is using VESC (Electric speed controller) which can be connected to the computer via USB. This connection is made by `vesc_driver` [3] package which receives various data from VESC internal sensors. This information is stored within `vesc_msgs/VescStateStamped` [4] messages.

Since raw data cannot be directly merged with the estimated position from particle filter (Subsection 5.2.1), they are converted to Odometry message using `vesc_ackermann` [2] package.

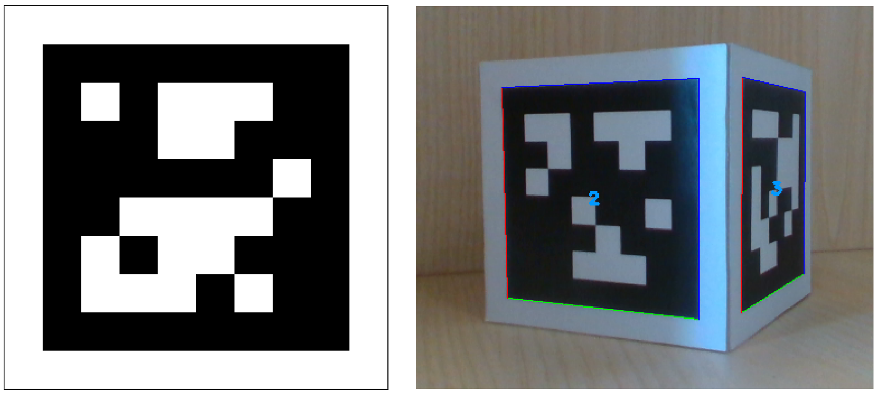
5.3 Traffic signs

Scenario definition from Section 3.1 takes into account the possibility of traffic signs on the track.

In real life scenarios, traffic signs are used to alter priority on the intersections. However, traffic sign recognition is prone to many difficulties, e.g., quality of image sensor, lighting conditions, color fading, scene complexity, similarity, and so forth, as described in [65]. Even though our scenario is limited, and, therefore, almost fully controllable (so problems with the sign recognition may be avoided or at least limited), using *fiducial markers* seems like a better choice.

Fiducial markers are objects placed within predefined area, designed to be easily recognized and distinguished from one another [41]. These markers are commonly used for localization [49], object tracking [77], or passing commands to the robot [15].

Let us note that this traffic sign substitution does not affect the designed algorithm. In case that real signs are required only relevant parts of the implementation have to be adjusted (e.g., used message type).



(a) : AprilTag with ID 27 **(b) :** Detected IDs 2 and 3

Figure 5.1: Examples of AprilTags from 36h11 family. It means that it is using 36-bit encoding with a minimum Hamming distance of 11.

Proposed solution utilizes AprilTags [41], monochrome markers that are designed to have large minimum Hamming distance (the difference between two tags) which allows to detect and correct possible bit errors. Example of an AprilTag is in Figure 5.1a.

The procedure of reading the AprilTags is as follows (described in [35]):

1. *Grayscaling* – high contrast parts of the image are binarized into black and white regions while the rest of the image is discarded,
2. *Segmentation* – union-find algorithm produces white and black segments,
3. *Boundary detection* – every black/white edge is recorded,
4. *Quad detection* – recorded boundaries (segments) are processed to find every four-sided shape,
5. *Payload decoding* – each detected quad is tested for a possible payload – quad is split into white and black parts (using adaptive threshold), while both parts are processed individually.

For detecting and reading AprilTags, AprilTags 2 algorithm was used. This algorithm is open-source, ROS compatible, and available on GitHub [34]. Example of an output from this algorithm is in Figure 5.1b.

Chapter 6

Simulation

This chapter describes the simulation of the implemented algorithm. At first, the selection of simulator application is discussed in Section 6.1, followed by an explanation of its usage and configuration. Further sections describe the creation of vehicle (Section 6.3) and environments models (Section 6.4). Adjustments that had to be done to make the functional architecture (Subsection 4.3.2) able to work with the simulator are described in Section 6.5. In the end (Section 6.6), scenarios are introduced and solved in simulation.

6.1 Simulator overview

This section is focused on the short review of existing simulators. Good selection of used simulator is important – dealing with unnecessary features can be counterproductive, and having interface incompatible with current design might require excessive adjustments. To at least avoid the latter one as much as possible, only ROS-compatible simulators were considered.

6.1.1 Gazebo

Gazebo [43] is a 3D dynamic simulator for populations of robots which can be accurately and efficiently simulated. It supports complex indoor and outdoor environments as well as precise physics simulation. As also written in [44], key features include “multiple physics engines, a rich library of robot models and environments, a wide variety of sensors and convenient programmatic and graphical interfaces”.

6.1.2 STDR Simulator

STDR Simulator [66] (which is an abbreviation for Simple Two Dimensional Robot Simulator) is a simple 2D simulator with an intention to make the simulation of single robots or a robot swarm as simple as possible. On the other hand, it is not aimed to be the most realistic simulator, nor to have many features. STDR (also sometimes mentioned as S2DR) is created to be a total ROS compliant [67].

6.1.3 Stage

Stage simulator is a 2D simulator with support for a third dimension. As said on their website [48], Stage is an application capable of simulating multiple robots along with their sensors. The simulation takes place within 2D bit-mapped environment. This “world” is occupied by simulated robots as well as other custom objects. The goal of Stage is to support research into multi-agent systems.

Stage itself supports communication with real robots. Connecting with the server side, which is known as Player [47], allows interacting between real and simulated robots.

ROS package *stage_ros* [21] enables to use Stage with ROS framework.

6.1.4 Simulator selection

In the end, Stage simulator described in Subsection 6.1.3 was selected as a simulator application for the implemented algorithm. In upcoming paragraphs reasons for this choice are explained.

Gazebo simulator was rejected because of its overall complexity. Also, it is mostly focused on the modelling of simulated robots, which is not the main focus of this thesis. Therefore, a more straightforward solution is appreciated.

STDR Simulator does fulfill requirements on the complexity. The two-dimensional world is much more suited for the topic of this thesis because the car’s movement is limited to one plane. On the other side, it does not allow to change the appearance of simulated robots and, more importantly, according to the ROS website [68], STDR is not able to provide detection of other robots during the simulation. Having the latter in mind, this simulator was rejected as well.

Since the other two simulators were dismissed, Stage simulator that does not suffer from the shortcoming of the other simulators was selected. Its simplicity and reliable two dimensional (with partial third dimension) simulation makes it a good choice. However, as explained later, even this simulator has its downsides.

6.2 Configuration reference

As mentioned in Subsection 6.1.4, even selected Stage simulator has its drawbacks. They are mostly related to insufficiently documented creation of configuration files. This section covers the basics of creating vehicle and world models for Stage.

Stage for ROS requires only one configuration file called *world*. Information about the simulation is stored inside. As a reference unofficial *How to use Player/Stage* documentation [40] is used within this whole section.

6.2.1 Model

Model [72] is a basic building block for all other model types. It simulates an object with basic properties, e.g., position, size, and color. These properties can be used for derived models as well. Only those that were used during the thesis are mentioned here.

Size. Parameter `size` is used for specifying the dimensions of the created model along all three axis – $[x, y, z]$. All variables are in meters.

Position. Parameter `pose` is used for placing the robot at the desired position during startup of the simulation. It is specified by four numbers – position in 3D space and heading. Both parts are relative to the parent coordinate system. The position is specified in meters like `size`; the heading is in degrees.

Color. Color of the created shape can be adjusted using the `color` parameter. Passed string parameters refer to the colors defined by X11 [19].

Name. Each model can be assigned a label to make it easily distinguishable. Parameter `name` has only one string argument which contains this marking.

Bitmap. When using a complex shape for a model `bitmap` parameter can be used. Through its only parameter path to a bitmap file can be specified. This is commonly used for loading a map.

Collidability. Ability to collide with other models is defined by parameter `obstacle_return`. If set to 1, this model can collide with other models with this property.

Reflectance. The ability of a model to reflect incoming rays from the simulated sensor is controlled by the parameter `ranger_return`. Setting this value to a negative number makes the model invisible to ranger sensors. Otherwise, it is detected.

Model stacking. When a model is attaching other models on itself, they are stacked on the top of the parent model. This can be turned off by setting `stack_children` parameter to zero, which sets the origin of each child model to the origin of the parent model.

6.2.2 Position model

Position model (object `position`) simulates a mobile robot base [73]. Since *position* has many parameters, only those that are relevant to our scenario are mentioned.

Steering model. Parameter `drive` defines model type used for the simulation. Possible models are: `diff` – differential-steer model (two wheels), `omni` – omnidirectional model, or `car` – which means that model is using forward velocity and steering angle.

Localization type. Parameter `localization` defines a model for reporting the position of the robot. Possible options are: `gps` – which means perfect accuracy, or `odom` – which adds overtime drift to the position (that is defined by `odom_error`).

Localization origin. Parameter `localization_origin` sets an origin for localization coordinate frame. It is specified by four numbers – $[x, y, z, \theta]$ where first three are translations in meters along one of three axes; fourth is used for ground rotation and it is specified in degrees. When left at zeros and `gps` model is selected, a perfect global position is returned for the robot.

Distance between wheels. Parameter `wheelbase` is used only for `car` steering model. It defines the distance between wheels (center to center on one side) in meters. This value is used to determine turning for the whole model.

6.2.3 Block model

Block model (object `block`) defines a form of a robot or obstacle using 2D shapes with a constant third dimension. Therefore, required design has to be split into basic 2D shapes which are then formed together in Stage.

Since blocks are mostly used within *position*, used coordinates are not specified in meters. Instead, they are rescaled to fit into the size of their parent.

Points array definition. Using the parameter `points`, an array of points is defined. The only passed argument is a number that specifies size of the array (i.e., number of points).

Points definition. Using parameters `point[X]` (where `X` is an index of a single point), points are filled into 2D space. Each point is specified by two numbers – $[x, y]$ coordinates along two axes.

Third dimension. Parameter `z` is used to add z coordinate to created shape. Two numbers are passed with this parameter – starting and ending z coordinate.

For an easier understanding of this part, a code snippet is added:

```
block
(
    # number of points
    points 4

    # coordinates of all points
    point[0] [0 0]
    point[1] [0 1]
    point[2] [1 1]
    point[3] [1 0]

    # third dimension of the block
    z [0 1]

    # optional color
    color "black"
)
```

This code creates a black cube with dimensions $1 \times 1 \times 1$ (meters in this case), starting in the origin of coordinates and spreading along all axes.

6.2.4 Ranger model

Ranger model (object `ranger`) simulates an array of sonar and infrared range sensors [74]. Because in our case, only one sensor is used, ranger has a single parameter – model `sensor` which is described in the next subsection.

6.2.5 Sensor model

Sensor model (object `sensor`) simulates sonar and infrared range sensors [74]. Besides `pose` and `size` parameters that are very important here, the following parameters can be used.

Field of view. Using the parameter `fov` field of view is specified. It expects a single argument which sets the angle span – minimal and maximal detection angle. This area is centered on the pose of the sensor.

Sensor range. Parameter `range` defines minimal and maximal detection range for the sensor.

Number of samples. The angular resolution of the sensor is determined by `samples` parameter. Its only argument corresponds to the number of samples that sensor measures during one sweep. This parameter is not officially documented but it can be found within the implementation [75]. It is also mentioned in the unofficial guide [40].

Fully specified sensor might look like this:

```
ranger
(
    sensor
    (
        pose [ -0.12 0 -0.0175 0 ]
        size [ 0.05 0.05 0.035 ]
        range [ 0.0  10.0 ]
        fov 270.0
        samples 3243
    )
    color "orange"
)
```

6.2.6 Camera model

Stage can even simulate an image stream from the camera. Therefore AprilTags can be used even in this scenario. Camera model (object `camera`) is used for this purpose [71].

Camera resolution. Resolution of a generated stream is specified via parameter `resolution`. Its arguments are *image width* and *image height*.

Camera range. Parameter `range` can be used to specify a minimal and maximal distance (in meters) of detectable obstacles – for this, two arguments are used.

Field of view. Similarly to the sensor model, a field of view can be set via `fov` parameter. In this case two parameters are used – one for *horizontal* and the other one for *vertical* field of view.

Camera tilt. Because camera does not need to be attached parallel to the ground, `pantilt` parameter is used to express this. The first argument sets an angle of left-right rotation while the second one up-down rotation.

6.3 Vehicle model

This section describes the modelling of the vehicle and its sensors. Parameters and their arguments are explained in Section 6.2.

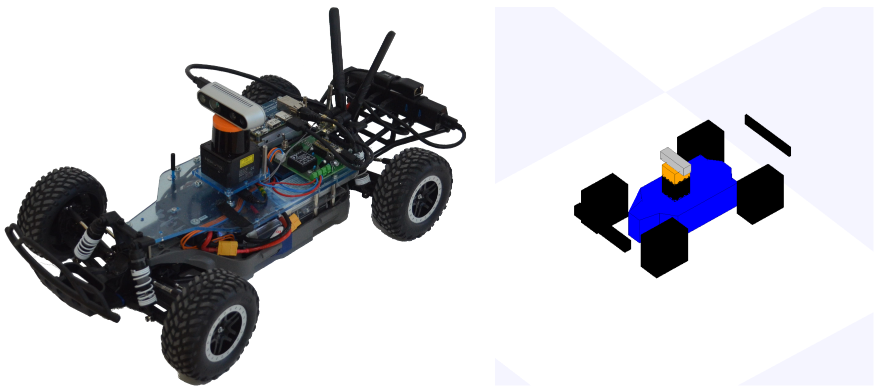

(a) : Real model **(b) :** Stage model

Figure 6.1: Comparison of vehicle models

Due to the complexity of the real model, only significant features are considered for designing the Stage model. As a minimal model bounding box would be enough. However, to make the simulation a little bit more realistic, these parts were created: front bouncer, back bouncer, chassis, and wheels. All parts were created using block models from Subsection 6.2.3.

From the devices mounted on the vehicle, only LiDAR (along with its base) and camera were created. Their parameters match with the devices used on the F1/10 car.

Comparison of the real model and its Stage counterpart is in Figure 6.1.

6.4 Environment model

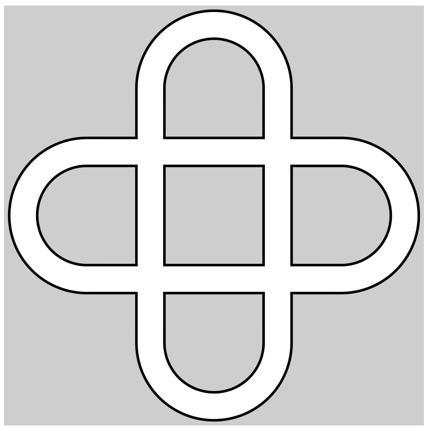
This section briefly explains how are the maps represented in ROS. Then a simple walkthrough for creating the map for the simulator is provided.

Basic ROS map is a 2D array that holds integer values in interval $\langle 0; 255\rangle$. This structure is called *occupancy grid*, and it is also a part of a ROS message of the same name [61]. The number within each cell determines the probability of an obstacle in that cell. For this only values in interval $\langle 0; 100\rangle$ are used. Other numbers are not used – except 255 which stands for unknown.

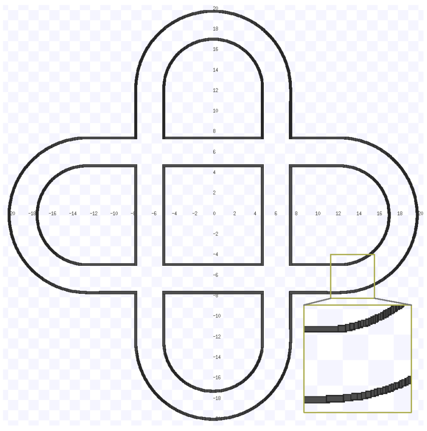
Nevertheless, the interpretation of the map as an image is different. It is using a grayscale palette. Usually, black color is used to represent obstacles, and the white color is used to represent free spaces. A configuration *yaml* [20] file is associated with this image. It holds information like thresholds for contrast intensity of free and occupied spaces, resolution of the map and the location of the origin.

Since the map can be drawn, it only takes a simple paint application for modelling the environment. Comparison of a drawn map and its simulated counterpart is in the Figure 6.2.

The downside of this map representation lies in the raster graphics. Using an occupancy grid reduces the accuracy of a modelled environment to the resolution of the original image.



(a) : Original map



(b) : Simulated map (with a zoomed part)

Figure 6.2: Comparison of maps

6.5 Implementation adjustments

When modelling of all required parts is finished, we can proceed to run the algorithm with the simulator. This section covers all adjustments that had to be made to simulate the environment for the algorithm. It is organized from the perspective of functional architecture.

6.5.1 Perception

As for the first part of the architecture, no changes were required. In fact, most of the packages from Perception are not used with the simulation.

Stage publishes simulated sensor data (LiDAR sweeps, camera images); therefore, connecting to these devices is not necessary. Also, these data are not preprocessed in any way which leaves only with the packages from Recognition. Packages from this part are used only for processing data for planners and for reading the AprilTags – no localization or intervehicle communication is done because this information is already generated by the simulator.

As mentioned in Subsection 4.3.2, planners should be subscribed to the topic with detected obstacles. Unfortunately, at this moment, there is not a planner that supports planning with obstacles. To at least follow the rules given by the architecture, "obstacle substitution" is done. It simply converts each LiDAR measurement into a small circle obstacle.

6.5.2 Decision and Control

Packages within Decision and Control are also not required to be internally modified in any way. On the other hand, a slight external touch is necessary. Basically, the planners are not prepared to be chained with some other package. Therefore topic remapping is a mandatory adjustment.

Planners are usually subscribed to `/obstacles` topic with message type `obstacle_msgs/Obstacles`, and they are sending commands to Drive-API via `/drive_api/command` topic (`drive_api/drive_api_values`). Since we need to redirect published messages to the intersection solver, remapping of the topic has to be done. This can be specified when starting the node:

```
rosrun [PACKAGE_NAME] [NODE_NAME] \
    /original_topic_name:=/new_topic_name
```

To avoid remembering names of all nodes throughout the workspace, each package has its own `start.launch` launch file. This file is specifically written to support topic remapping in the same way, as running a node. Therefore, this is an identical way of launching the node:

```
roslaunch [PACKAGE_NAME] start.launch \
    remap:=true \
    /original_topic_name:=/new_topic_name
```

Additional argument `remap` is used to ensure that remapping will be done only when explicitly required.

During the simulations, only one reactive planner was used. It is placed within `scan_regression` package. This planner is using first order polynomial curve fitting (*polyfit*) – it intersects a straight line through all detected obstacles (measured points). The slope of this line is used as a target steering angle for the Drive-API.

6.5.3 Vehicle platform

The physical platform is using Teensy board (Subsection 4.2.3) to control the vehicle. However, this is not possible to use with the simulator. Also, as stated in Subsection 4.3.2, the second part of the Vehicle platform is Drive-API, which is platform dependent. Since simulated cars do not have any control-related parameters and the Teensy is not available for simulation, there are two possible ways of resolving this: either create a new package for compatibility between platform interface and the simulator or modify Drive-API to support simulated environments. The latter approach was selected as it requires only minor modification to the existing ROS node.

Stage is using for controlling the robots standardized message (as it is used on many robots – e.g., [46] or [78]) on the topic `/cmd_vel` with message type `geometry_msgs/Twist` [57]. Publishing of these messages was added to the Drive-API.

6.6 Testing

This section concludes the simulation chapter. It starts with a list of commands that are necessary for running the simulation. Then particular testing scenarios are described.

6.6.1 Running the simulation

To run all the packages that are used within the simulation of a car, the following commands have to be used:

```
roslaunch obstacle_substitution start.launch \
    remap:=true \
    /scan:=/robot_0/base_scan \
    /obstacles:=/robot_0/obstacles \
    anonymous:=true

roslaunch scan_regression start.launch \
    remap:=true \
    /obstacles:=/robot_0/obstacles \
    /drive_api/command:=/robot_0/command \
    anonymous:=true
```

```
roslaunch intersection_solver start.launch \
    remap:=true \
    /odom1:=/robot_0/odom /odom2:=/robot_1/odom \
    /eStop:=/robot_0/eStop /command:=/robot_0/command \
    /drive_api/command:=/robot_0/drive_api/command

roslaunch drive_api start.launch \
    remap:=true \
    /drive_api/command:=/robot_0/drive_api/command \
    /cmd_vel:=/robot_0/cmd_vel /eStop:=/robot_0/eStop \
    anonymous:=true
```

These commands are similar for the other car; only `robot_0` is replaced by `robot_1` and vice versa. Besides this, certain nodes require additional argument `anonymous`. These nodes are usually running in a *one instance mode* which ensures that specific functions are not handled by multiple instances of the same node. This behaviour is favourable when using the real platform to avoid unexpected reactions. However, it is not desirable during the simulation. Therefore by setting the argument `anonymous`, multiple instances become allowed.

To start the simulation itself, the following command needs to be run:

```
rosrun stage_ros stageros [WORLD_FILE]
```

Running this command opens Stage GUI, which shows the environment along with simulated cars. However, to make the things a little bit easier, special launch file for running the nodes with proper remapping was created. It also opens rViz application [22], which visualizes all simulated data. Thus `rviz` package is required. Also for making the rViz able to display a map of the environment, `map_server` [20] package is necessary as well.

However, during the testing phase, it was discovered that running the simulator along with all required nodes by one launch file sometimes leads to the failing of simulator start. Therefore it is split into two files which can be launched via the following commands:

```
roslaunch launchers stage_sample_intersection.launch \
    map:=true rviz:=true \
    map_name:=[NAME_OF_MAP_FILE]

roslaunch launchers stage.launch \
    world:=[NAME_OF_WORLD_FILE]
```

Since the folder for keeping the world/map files is explicitly defined in the launch files, only names of these files are required.

6.6.2 Testing scenarios

For testing the algorithm in the simulator map shown in the Figure 6.2 was used. This map was designed to make the cars drive without human interaction (e.g., to reset their position) and to be able to test multiple intersection layouts at once. Images included in each scenario are focused on the area of interest.

Scenario 1

In the first scenario, cars are driving towards the intersection that is not modified by any traffic signs. Therefore right-side rule should be applied here. Their starting position is in the same distance from the center of the intersection, and their speeds are equal as well. Simulation of this scenario is shown in Figure 6.3. A recording of the simulation is available on server Youtube [25].

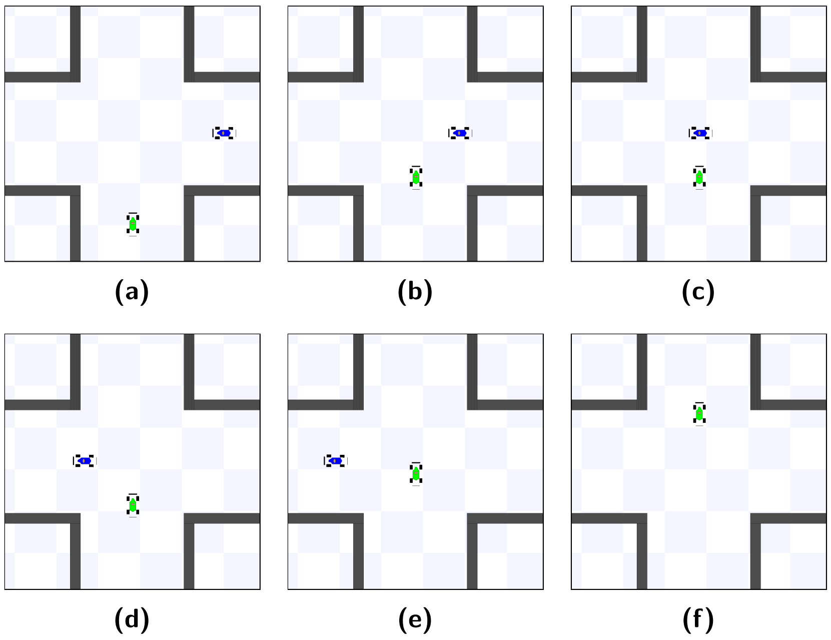


Figure 6.3: Simulation for Scenario 1 – apply the right-side rule. Image **(a)** shows the situation before the intersection; in **(b)** green car yields to the blue car; in **(c)** blue car is inside the intersection collision area; in **(d)** blue car leaves the intersection; in **(e)** green car is inside the intersection collision area; and in **(f)** green car leaves the intersection.

Scenario 2

In the second scenario, the intersection does not contain any traffic signs as well. The distance of the green car to the intersection center is twice the distance of the blue car. On the other hand, green car is twice as fast as blue car. Simulation of this scenario is shown in Figure 6.4. A recording of the simulation is available on server Youtube [26].

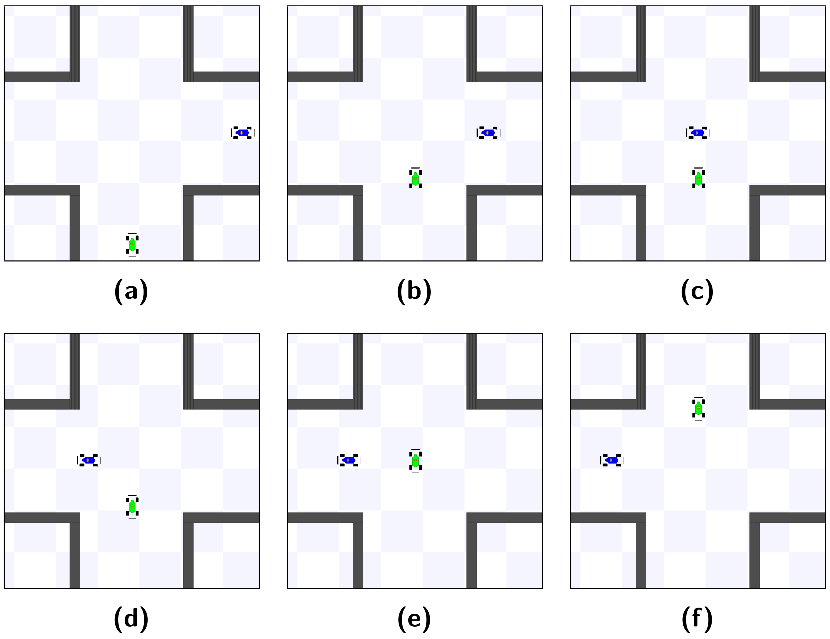


Figure 6.4: Simulation for Scenario 2 – apply the right-side rule on a slower car. In **(a)** the situation before the intersection is shown; in **(b)** green car yields to blue car; in **(c)** blue car is inside the intersection collision area; in **(d)** blue car leaves the intersection; in **(e)** green car is inside the intersection collision area; and in **(f)** green car leaves the intersection.

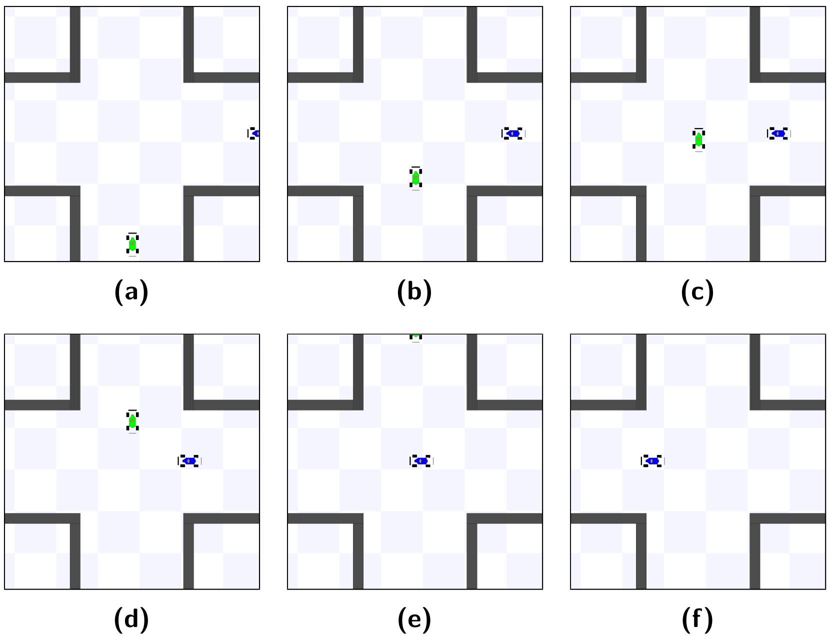


Figure 6.5: Simulation for Scenario 3 – faster car does not have to apply the right-side rule. In **(a)** the situation before the intersection is shown; in **(b)** green car enters the collision area; in **(c)** green car is inside the intersection collision area; in **(d)** green car leaves the intersection; in **(e)** blue car is inside the intersection collision area; and in **(f)** blue car leaves the intersection.

Scenario 3

The third scenario is very similar to Scenario 2; as an only difference, green car is a little bit closer to the intersection center. Therefore, it can drive through without stopping and without endangering blue car. Simulation of this scenario is shown in Figure 6.5. A recording of the simulation is available on server Youtube [27].

Scenario 4

The fourth scenario introduces AprilTags (see Subsection 5.3). Because of the modification of the intersection, blue car has to yield this time. Distances from the intersection and speeds of the cars are equal. Simulation of this scenario is shown in Figure 6.6. A recording of the simulation is available on server Youtube [28].

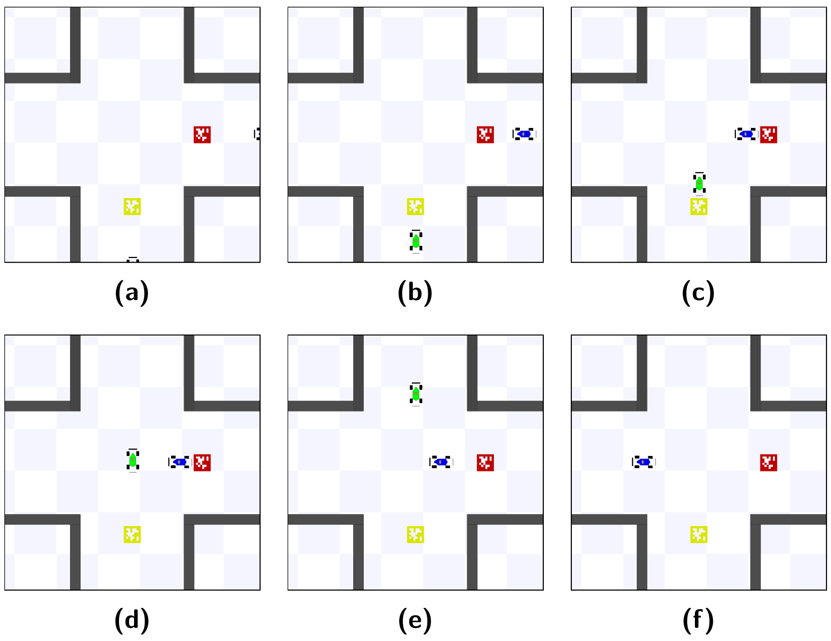


Figure 6.6: Simulation for Scenario 4 – intersection modified by AprilTags. AprilTags are colored to make them easily recognizable – red AprilTag imitates YIELD sign, yellow AprilTag imitates PRIORITY sign. In **(a)** the situation before the intersection is shown; in **(b)** traffic signs are detected by both cars; in **(c)** blue car yields to green car; in **(d)** green car is inside the intersection; in **(e)** blue car enters the intersection collision area; and in **(f)** blue car leaves the intersection.

Scenario 5

The fifth scenario shows the ability to solve a skewed intersection. Both cars are driving at the same speed, and their distance from the intersection center is approximately identical. The right-side rule is applied in this

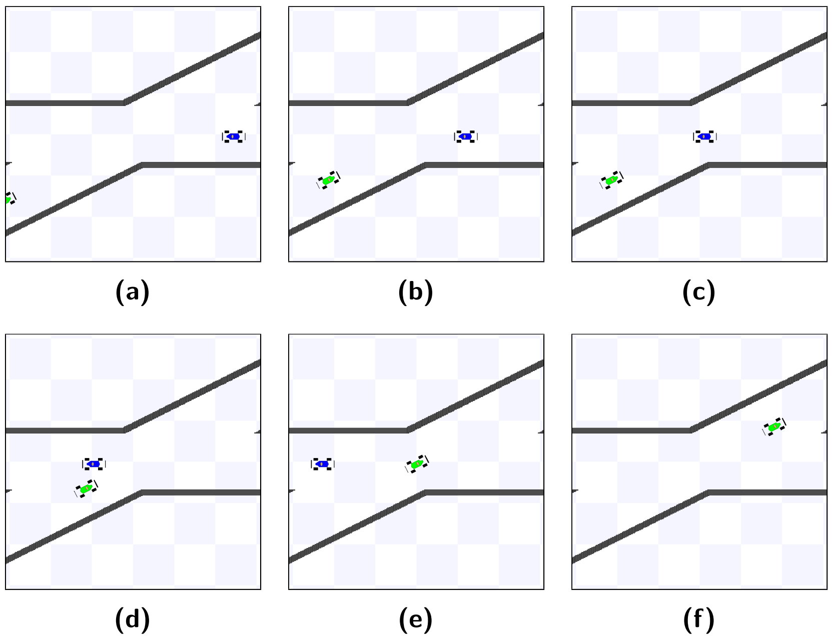


Figure 6.7: Simulation for Scenario 5 – skewed intersection. In **(a)** the situation before the intersection is shown; in **(b)** green car yields to blue car; in **(c)** blue car leaves the intersection collision area; in **(d)** green car approaches blue car; in **(e)** green car is inside the intersection collision area; and in **(f)** green car leaves the intersection.

case. Simulation of this scenario is shown in Figure 6.7. A recording of the simulation is available on server Youtube [29].

Let us note that this scenario was simulated without the reactive planner. Unlike previous scenarios, reactive action does not secure driving straight through the intersection. Instead of planner, static steering and velocity data were published.

The proximity of both cars in Figure 6.7d does not lead to the collision. This situation is not prohibited by the algorithm (Section 3.3) because it expects that both vehicles will continue in their way using the same heading and speed. Cases which include changing of one (or both) of these variables within the intersection would be solved by the underlying planner.

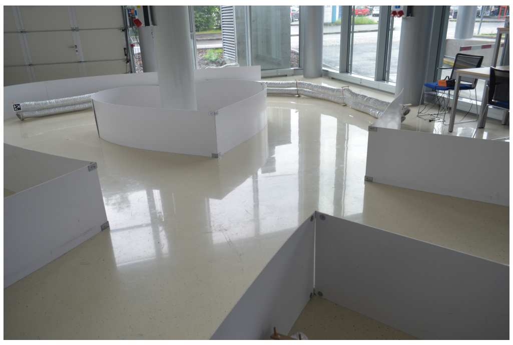
Chapter 7
Experiments

This chapter describes experiments performed on the F1/10 platform (Section 4.2). At first, a testing environment is introduced in Section 7.1. Then, in Section 7.2, the required adjustments between the simulation and the experiments are written down. The last section describes field testing.

7.1 Scenario description

Testing track was built in an office that is located on the ground floor of the building A of Czech Institute of Informatics, Robotics, and Cybernetics (CIIRC). Photo of the environment is in Figure 7.1a. Map created by `hector_mapping` (see Subsection 5.2.1) is shown in Figure 7.1b.

Unfortunately because of the limited environment, larger testing track could not be built. However this scenario is sufficient for testing.


(a) : Photo of the real track

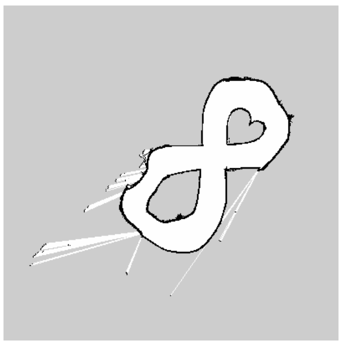

(b) : Map of the track

Figure 7.1: Track used for experiments

7.2 Adjustments

Even though the algorithm was implemented with the real platform taken into account, during the experiments certain problems were discovered. This section describes these issues.

7.2.1 Vehicle configuration

Preparing the experiments, one of the testing cars was not responding properly to the used `scan_regression` planner (see Subsection 6.5.2). Therefore older *Follow the Gap* planner had to be used instead. Follow the Gap is a method of reactive planning which detects obstacles ahead of the vehicle, finds the largest gap between obstacles available, and computes action towards its center. Referring to this as an "older" planner means that it is not compatible with Drive-API, therefore, this car could be only stopped.

7.2.2 Hardware limitation

During performing the experiments hardware related problem was found. Used TX2 board (Section 4.2) is not powerful enough to run all required nodes reliably – i.e. localization, intersection control, AprilTags detection, intervehicle communication and platform control (via Teensy board). Turning all of these nodes led to slowing down the commands from the planner to the Teensy board, which means that the car started to crash into the walls. This was partially solved by adjusting the priorities of the processes (lowering priority of AprilTags reader) and by lowering the camera frame rate. Even though the commands were sent in time, AprilTags were most of the time missed. Because of that, AprilTags reading was disabled for the experiments.

This issue is closely related to the problem of real-time computing. Having a real-time operating system (and support of real-time within the ROS framework), one could ensure that certain processes are done in time and, therefore, theoretically make it possible to do the experiments. However ROS does not support that. Support of real-time programming is planned for ROS 2 [55].

7.3 Testing

Due to real-time computing related problem described in Subsection 7.2.2 only the first simulated scenario (Subsection 6.6.2) was tested on the real platform.

In this scenario, intersection is not modified by any traffic signs. Therefore, the right-side rule should be applied. Both cars are in an approximately equal distance from the intersection center and their speeds are also similar. Pictures from the experiment are in Figure 7.2. A recording of the experiment is available on server Youtube [24].

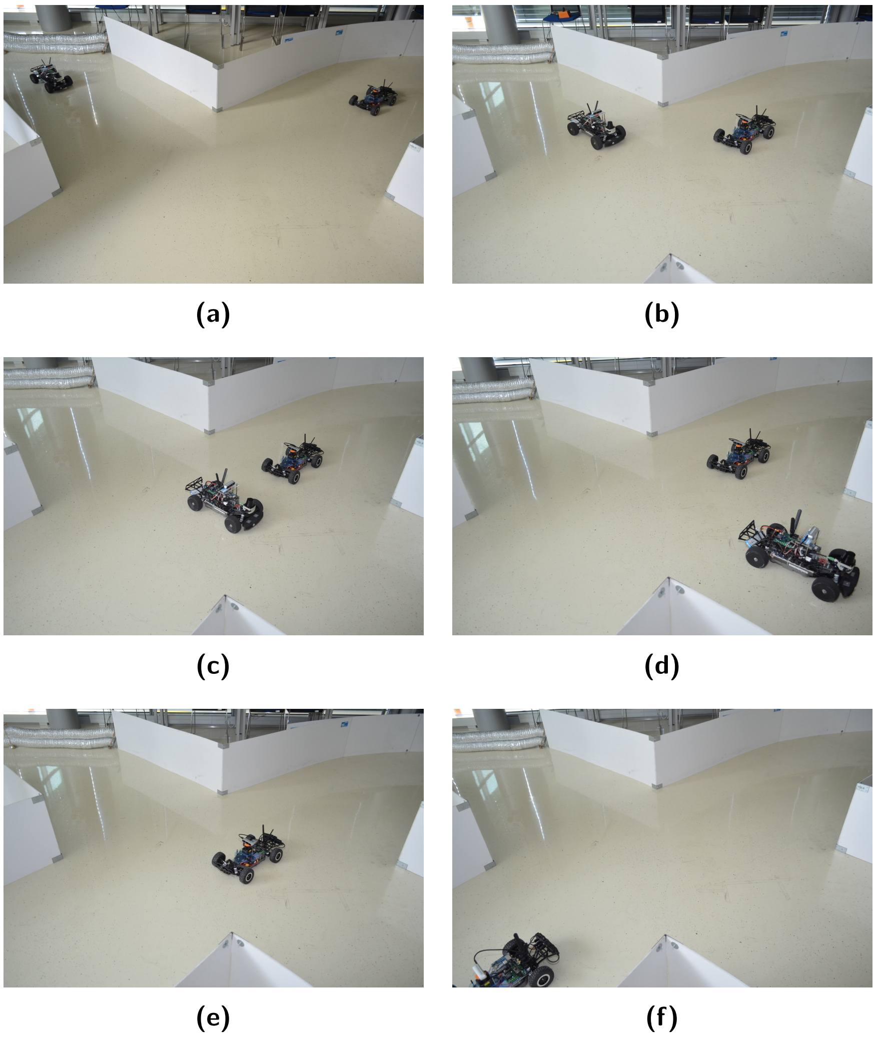


Figure 7.2: Experiment for Scenario 1. In **(a)**, both cars are approaching the intersection. The car that starts on the left side of the picture is referred to as "First", the other one is referred to as "Second". In **(b)**, Second car applies the right-side rule. In **(c)**, First car is inside the collision area. In **(d)**, First car leaves the intersection. In **(e)**, Second car is inside the collision area. In **(f)**, Second car leaves the intersection.

Chapter 8

Future work

Throughout the thesis, certain directions for future work were suggested. In this chapter, these ideas are put together.

Object detection

One of the possible improvements was suggested in Subsection 5.2.3. Sometimes it may happen that the intervehicle communication is not available (e.g., not supported on target vehicle, car in failure state). To avoid relying on the V2V, one could detect other vehicles using LiDAR, camera, or both.

Data from these sensors will be processed by an object detection algorithm to find other vehicles in their field of view, and also, to assign them an estimation of their velocity. Therefore, the requirement for the intervehicle communication (Section 3.1, rule 4) will not be necessary, since the required data can be obtained through object detection.

Behaviour reliability

The topic of reliability is quite tightly connected to the previous paragraph. Having multiple sources of the same data can be used for detecting faults in the sensors, which can contribute to implementing sensor redundancy.

Trajectory planning

Another possible improvement comes up from the algorithm description (Subsection 3.3.5). This algorithm does not implement its trajectory planning. Adding a custom planner could lead to more complex solutions of intersection control since instead of just heading, the whole path will be returned.

However, to make this work, an interface between the planners have to be created. This interface could allow the nodes to exchange information (e.g., goals, constraints).

Behaviour tree

As a part of the group of traffic rules, intersection control could be implemented as a part of a behaviour tree [6], which allows having all the rules at one place, along with resolving other scenarios (e.g., parking).

Hardware improvements

Another possible direction for future work comes up from the experiments (Subsection 7.2.2). During the experiments, it was found out that the hardware of the platform is not powerful enough to run all necessary nodes.

Therefore, a new platform design could be created in a way to boost performance. There are at least two ways how to do it: either buy a more powerful CPU or move some nodes (e.g., AprilTags detection) to a dedicated board to free up the computation time on the main computer.

Chapter 9
Conclusion

In this thesis, the algorithm for intersection control was designed, implemented, simulated, and tested on F1/10 platform.

The design is based on *passive intersection control* that allows solving the intersections without the need to develop a dedicated device. This solution is suited for low traffic intersection control.

F1/10 platform, RC-based autonomous car model, was introduced and described.

Upon reviewing the available simulators, Stage simulator was selected for algorithm verification. Model of the platform was created in this simulator, for use in multiple scenarios that were designed for evaluation. One of these scenarios was also tested on the F1/10 car.

Ideas for the improvements, based on encountered issues, were suggested in the last chapter.

Appendix A

Bibliography

[1] S. Behere and M. Törngren. A functional architecture for autonomous driving. In *Proceedings of the First International Workshop on Automotive Software Architecture*, WASA '15, pages 3–10, New York, NY, USA, 2015. ACM.

[2] M. T. Boulet. *vesc/vesc_ackermann – mit-racecar/vesc – GitHub*. `https://github.com/mit-racecar/vesc/tree/master/vesc_ackermann`.

[3] M. T. Boulet. *vesc/vesc_driver – mit-racecar/vesc – GitHub*. `https://github.com/mit-racecar/vesc/tree/master/vesc_driver`.

[4] M. T. Boulet. *vesc/VescStateStamped.msg – mit-racecar/vesc – GitHub*. `https://github.com/mit-racecar/vesc/blob/master/vesc_msgs/msg/VescStateStamped.msg`.

[5] M. Brossard and S. Bonnabel. Learning Wheel Odometry and IMU Errors for Localization. In *International Conference on Robotics and Automation (ICRA)*, Montreal, Canada, May 2019.

[6] M. Colledanchise and P. Ogren. *Behavior Trees in Robotics and AI: An Introduction*, July 2018.

[7] K. Conley. *roslaunch/XML – ROS Wiki*. `http://wiki.ros.org/roslaunch/XML`.

[8] K. Conley. *rospy – ROS Wiki*. `http://wiki.ros.org/rospy`.

[9] F. Dellaert, D. Fox, W. Burgard, and S. Thrun. Monte carlo localization for mobile robots. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, volume 2, pages 1322–1328 vol.2, May 1999.

[10] A. Discant, A. Rogozan, C. Rusu, and A. Bensrhair. Sensors for obstacle detection - a survey. In *2007 30th International Spring Seminar on Electronics Technology (ISSE)*, pages 100–105, May 2007.

[11] K. Dresner and P. Stone. Multiagent traffic management: a reservation-based intersection control mechanism. In *Proceedings of the Third*

International Joint Conference on Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004., pages 530–537, July 2004.

[12] K. Dresner and P. Stone. Multiagent traffic management: An improved intersection control mechanism. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '05, pages 471–477, New York, NY, USA, 2005. ACM.

[13] K. Dresner and P. Stone. Turning the corner: improved intersection control for autonomous vehicles. In *IEEE Proceedings. Intelligent Vehicles Symposium, 2005.*, pages 423–428, June 2005.

[14] K. Dresner and P. Stone. A multiagent approach to autonomous intersection management. *J. Artif. Intell. Res. (JAIR)*, 31:591–656, 01 2008.

[15] G. Dudek, P. Giguère, and J. Sattar. *Sensor-Based Behavior Control for an Autonomous Underwater Vehicle*, pages 267–276. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[16] J. Dusil. Slip detection for F1/10 model car. Bachelor thesis, Czech Technical University in Prague, May 2019.

[17] F1tenth organizers. *F1/10 Competition Rules*. `http://f1tenth.org/race/rules-v2.pdf`.

[18] F1tenth organizers. *F1/10 Lectures*. `http://f1tenth.org/lectures`.

[19] J. Fulton. *X colorname to RGB mapping database*. `https://cgit.freedesktop.org/xorg/app/rgb/tree/rgb.txt`.

[20] B. Gerkey, T. Pratkanis, et al. *map_server – ROS Wiki*. `http://wiki.ros.org/map_server`.

[21] B. Gerky. *ros-simulation/stage_ros: Package which contains ROS specific hooks and tools for the Stage simulator*. `https://github.com/ros-simulation/stage_ros`.

[22] D. Hershberger, D. Gossow, and J. Faust. *rviz – ROS Wiki*. `http://wiki.ros.org/rviz`.

[23] J. Khoury and J. Khoury. Passive, decentralized, and fully autonomous intersection access control. In *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pages 3028–3033, Oct 2014.

[24] J. Klapálek. *Intersection control experiment – Scenario 1*. `https://www.youtube.com/watch?v=3BN59KcW-mA`.

[25] J. Klapálek. *Intersection control simulation – Scenario 1*. `https://www.youtube.com/watch?v=bp_URoOO_Ac`.

[26] J. Klapálek. *Intersection control simulation – Scenario 2.* `https://www.youtube.com/watch?v=uEyJQk4jui4`.

[27] J. Klapálek. *Intersection control simulation – Scenario 3.* `https://www.youtube.com/watch?v=P2OTDXbW3jM`.

[28] J. Klapálek. *Intersection control simulation – Scenario 4.* `https://www.youtube.com/watch?v=_pU0--7Pm7Q`.

[29] J. Klapálek. *Intersection control simulation – Scenario 5.* `https://www.youtube.com/watch?v=FyH31lA6MJs`.

[30] S. Kohlbrecher. *hector_mapping – ROS Wiki.* `http://wiki.ros.org/hector_mapping`.

[31] S. Kohlbrecher, O. von Stryk, J. Meyer, and U. Klingauf. A flexible and scalable slam system with full 3d motion estimation. In *2011 IEEE International Symposium on Safety, Security, and Rescue Robotics*, pages 155–160, Nov 2011.

[32] D. Kopecký. Localization and advance control for autonomous model cars. Master thesis, Czech Technical University in Prague, June 2019.

[33] P. Lin, J. Liu, P. J. Jin, and B. Ran. Autonomous vehicle-intersection coordination method in a connected vehicle environment. *IEEE Intelligent Transportation Systems Magazine*, 9(4):37–47, winter 2017.

[34] D. Malyuta. *AprilRobotics/apriltag_ros: A ROS wrapper of the AprilTags 2 visual fiducial detector.* `https://github.com/AprilRobotics/apriltag_ros`.

[35] D. Malyuta. Guidance, Navigation, Control and Mission Logic for Quadrotor Full-cycle Autonomy. Master thesis, Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, CA 91109, USA, December 2017.

[36] D. Malyuta and W. Merkt. *apriltags2_ros/AprilTagDetectionArray Documentation.* `http://docs.ros.org/kinetic/api/apriltags2_ros/html/msg/AprilTagDetectionArray.html`.

[37] P. Merriaux, Y. Dupuis, P. Vasseur, and X. Savatier. Wheel odometry-based car localization and tracking on vectorial map. *2014 17th IEEE International Conference on Intelligent Transportation Systems, ITSC 2014*, pages 1890–1891, 10 2014.

[38] MIT RACECAR team. *mit-racecar/particle_filter: A fast particle filter localization algorithm for the MIT Racecar.* `https://github.com/mit-racecar/particle_filter`.

[39] MIT RACECAR team. *RACECAR – A Powerful Platform for Robotics Research and Teaching.* `https://mit-racecar.github.io/`.

[40] K. Nickels and J. Owen. *Ch3 – Building a World – How to use Player/Stage*. https://player-stage-manual.readthedocs.io/en/latest/WORLDFILES/.

[41] E. Olson. Apriltag: A robust and flexible visual fiducial system. In *2011 IEEE International Conference on Robotics and Automation*, pages 3400–3407, May 2011.

[42] E. Onieva, V. Milanés, J. Villagrá, J. Pérez, and J. Godoy. Genetic optimization of a vehicle fuzzy decision system for intersections. *Expert Systems with Applications*, 39(18):13148 – 13157, 2012.

[43] Open Source Robotics Foundation. *Gazebo*. http://gazebosim.org/.

[44] Open Source Robotics Foundation. *Gazebo: Tutorial: Beginner: Overview*. http://gazebosim.org/tutorials?cat=guided_b&tut=guided_b1.

[45] Open Source Robotics Foundation. *ROS.org | About ROS*. http://www.ros.org/about-ros/.

[46] J. Pages. *Robots/TIAGo/Tutorials/motions/cmd_vel – ROS Wiki*. http://wiki.ros.org/Robots/TIAGo/Tutorials/motions/cmd_vel.

[47] Player Project contributors. *Player Project – Player cross-platform robot device interface & server*. http://playerstage.sourceforge.net/index.php?src=player.

[48] Player Project contributors. *Player Project – Stage 2D multiple-robot simulator*. http://playerstage.sourceforge.net/index.php?src=stage.

[49] C. Potena, D. Nardi, and A. Pretto. Effective target aware visual navigation for uavs. In *2017 European Conference on Mobile Robots (ECMR)*, pages 1–7, Sep. 2017.

[50] M. Przybyła. *obstacle_detector/Obstacles.msg – tysik/obstacle_detector – GitHub*. https://github.com/tysik/obstacle_detector/blob/master/msg/Obstacles.msg.

[51] M. Przybyła. *tysik/obstacle_detector: A ROS package for 2D obstacle detection based on laser range data*. https://github.com/tysik/obstacle_detector.

[52] M. Przybyła. Detection and tracking of 2d geometric obstacles from lrf data. In *2017 11th International Workshop on Robot Motion and Control (RoMoCo)*, pages 135–141, July 2017.

[53] M. Quigley, J. Faust, B. Gerkey, and T. Straszheim. *roscpp – ROS Wiki*. http://wiki.ros.org/roscpp.

[54] J. Rios-Torres and A. A. Malikopoulos. A survey on the coordination of connected and automated vehicles at intersections and merging at highway on-ramps. *IEEE Transactions on Intelligent Transportation Systems*, 18(5):1066–1077, May 2017.

[55] ROS 2 Community. *Real-Time Programming in ROS 2*. `https://index.ros.org//doc/ros2/Tutorials/Real-Time-Programming/`.

[56] ROS Community. *geometry_msgs/Point Documentation*. `http://docs.ros.org/kinetic/api/geometry_msgs/html/msg/Point.html`.

[57] ROS Community. *geometry_msgs/Twist Documentation*. `http://docs.ros.org/kinetic/api/geometry_msgs/html/msg/Twist.html`.

[58] ROS Community. *geometry_msgs/Vector3 Documentation*. `http://docs.ros.org/kinetic/api/geometry_msgs/html/msg/Vector3.html`.

[59] ROS Community. *Master – ROS Wiki*. `http://wiki.ros.org/Master`.

[60] ROS Community. *Messages – ROS Wiki*. `http://wiki.ros.org/Messages`.

[61] ROS Community. *nav_msgs/OccupancyGrid Documentation*. `http://docs.ros.org/kinetic/api/nav_msgs/html/msg/OccupancyGrid.html`.

[62] ROS Community. *nav_msgs/Odometry Documentation*. `http://docs.ros.org/kinetic/api/nav_msgs/html/msg/Odometry.html`.

[63] ROS Community. *ROS/Introduction – ROS Wiki*. `http://wiki.ros.org/ROS/Introduction`.

[64] ROS Community. *Topics – ROS Wiki*. `http://wiki.ros.org/Topics`.

[65] Š. Toth. Difficulties of traffic sign recognition. *MICT 2012, 7-th Winter School of Mathematics Applied to ICT*, pages 6–11, February 2012.

[66] M. Tsardoulias, C. Zalidis, and A. Thallas. *STDR Simulator*. `http://stdr-simulator-ros-pkg.github.io/`.

[67] M. Tsardoulias, C. Zalidis, and A. Thallas. *STDR Simulator – ROS robotics news*. `http://www.ros.org/news/2014/02/stdr-simulator-simple-two-dimensional-robot-simulator.html`.

[68] M. Tsardoulias, C. Zalidis, and A. Thallas. *stdr_simulator – ROS Wiki*. `http://wiki.ros.org/stdr_simulator`.

[69] M. Vajnar. Model car for the F1/10 autonomous car racing competition. Master thesis, Czech Technical University in Prague, June 2017.

[70] M. Van Middlesworth, K. Dresner, and P. Stone. Replacing the stop sign: unmanaged intersection control for autonomous vehicles. In Lin Padgham, David C. Parkes, Jörg P. Müller, and Simon Parsons, editors, *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal, May 12-16, 2008, Volume 3*, pages 1413–1416. IFAAMAS, 2008.

[71] R. Vaughan et al. *Stage Manual: Camera model*. `http://rtv.github.io/Stage/group__model__camera.html`.

[72] R. Vaughan et al. *Stage Manual: Model*. `http://rtv.github.io/Stage/group__model.html`.

[73] R. Vaughan et al. *Stage Manual: Position model*. `http://rtv.github.io/Stage/group__model__position.html`.

[74] R. Vaughan et al. *Stage Manual: Ranger model*. `http://rtv.github.io/Stage/group__model__ranger.html`.

[75] R. Vaughan et al. *Stage/model_ranger.cc – rtv/Stage*. `https://github.com/rtv/Stage/blob/master/libstage/model_ranger.cc`.

[76] C. H. Walsh and S. Karaman. Cddt: Fast approximate 2d ray casting for accelerated localization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8, May 2018.

[77] J. Wang, C. Sadler, C. F. Montoya, and J. C. L. Liu. Optimizing ground vehicle tracking using unmanned aerial vehicle and embedded apriltag design. In *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 739–744, Dec 2016.

[78] M. Wise and D. Lim. *turtlebot3_teleop – ROS Wiki*. `http://wiki.ros.org/turtlebot3_teleop`.

[79] K. Yamaguchi, T. Kato, and Y. Ninomiya. Vehicle ego-motion estimation and moving object detection using a monocular camera. In *18th International Conference on Pattern Recognition (ICPR'06)*, volume 4, pages 610–613, Aug 2006.

[80] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338 – 353, 1965.